

Search-based Test Generation for Automated Driving Systems:

From Perception to Control Logic

by

Cumhur Erkan Tuncali

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2019 by the
Graduate Supervisory Committee:

Georgios Fainekos, Chair
Heni Ben Amor
James Kapinski
Aviral Shrivastava

ARIZONA STATE UNIVERSITY

May 2019

ABSTRACT

Automated driving systems are in an intensive research and development stage, and the companies developing these systems are targeting to deploy them on public roads in a very near future. Guaranteeing safe operation of these systems is crucial as they are planned to carry passengers and share the road with other vehicles and pedestrians. Yet, there is no agreed-upon approach on how and in what detail those systems should be tested. Different organizations have different testing approaches, and one common approach is to combine simulation-based testing with real-world driving.

One of the expectations from fully-automated vehicles is never to cause an accident. However, an automated vehicle may not be able to avoid all collisions, *e.g.*, the collisions caused by other road occupants. Hence, it is important for the system designers to understand the boundary case scenarios where an autonomous vehicle can no longer avoid a collision. Besides safety, there are other expectations from automated vehicles such as comfortable driving and minimal fuel consumption. All safety and functional expectations from an automated driving system should be captured with a set of system requirements. It is challenging to create requirements that are unambiguous and usable for the design, testing, and evaluation of automated driving systems. Another challenge is to define useful metrics for assessing the testing quality because in general, it is impossible to test every possible scenario.

The goal of this dissertation is to formalize the theory for testing automated vehicles. Various methods for automatic test generation for automated-driving systems in simulation environments are presented and compared. The contributions presented in this dissertation include (i) new metrics that can be used to discover the boundary cases between safe and unsafe driving conditions, (ii) a new approach

that combines combinatorial testing and optimization-guided test generation methods, (iii) approaches that utilize global optimization methods and random exploration to generate critical vehicle and pedestrian trajectories for testing purposes, (iv) a publicly-available simulation-based automated vehicle testing framework that enables application of the existing testing approaches in the literature, including the new approaches presented in this dissertation.

To my lovely wife Gözde and my dear son Bulut.

ACKNOWLEDGMENTS

I was lucky to have the opportunity to do research in one of the most interesting areas to me in a friendly environment in the Cyber-Physical Systems Lab. at Arizona State University. I would like to thank my colleagues in CPS-Lab, Joe Campbell, Shakiba Yaghoubi, Bardh Hoxha, Adel Dokhanchi, Kangjin Kim, Houssam Abbas, for their close and hopefully long-lasting friendships, for our quality discussions and for our research collaborations.

Special thanks to my advisor Georgios Fainekos for providing a healthy environment for doing high-quality research. I feel lucky for having the opportunity to work with him. His attitude has always been encouraging to explore new ideas. I could easily see the highest level of respect he shows to his students and their work. I also greatly appreciate the independence he provided in the way I have worked. His qualities made my Ph.D. life enjoyable while being challenged with difficult research problems every day.

I would like to thank my committee members Heni Ben Amor, James Kapinski and Aviral Shrivastava for their insights and invaluable guidance.

I am grateful to Ken Butts, Hisahiro Ito, James Kapinski and Danil Prokhorov from Toyota for their friendship, support, and guidance throughout my internship at Toyota, and for the close collaboration during my Ph.D. studies. Jim, I deeply appreciate all your help and guidance during my stay in snowy Ann Arbor. I would like to thank Christoph Gladisch from Bosch, Xiaoqing Jin, Jyotirmoy Deshmukh, Sriram Sankaranarayanan, Theodore Pavlic and Yann-Hang Lee for our collaborations.

My parents, my brother and all my family members have always believed in me, and their support kept me walking toward my goals. Thank you, Mom and Dad, for

all kinds of endless support that enabled me to take risks, try and achieve without the fear of losing.

Finally, and most importantly, thank you, my dear love, Gözde! This dissertation wouldn't be possible without your endless love, support and friendship during all these years. You have always walked with me at all the good and tough times. Leaving my professional career for a Ph.D. in a different country was a difficult decision, and I wouldn't be able to do this without having you on my side. Thank you for giving me the best gifts in my life, your love and our dear son Bulut. I love you! Bulut, my cheerful son, you have taught me the most important thing, Life!. I understand the world and life much better by learning a lot from you every single day. A smile on your face gives me all I need to get over any difficulty I can face. I love you, Son!

My dissertation work was partially supported by the National Science Foundation under Grants NSF CNS-1350420, NSF CNS-1319560, NSF CNS-1446730, NSF IIP-1361926 and the NSF I/UCRC Center for Embedded Systems.

TABLE OF CONTENTS

	Page
LIST OF TABLES	xi
LIST OF FIGURES	xii
CHAPTER	
1 INTRODUCTION	1
1.1 Automated Driving Systems	1
1.1.1 Levels of Automation	1
1.1.2 Automated Driving Systems Architecture	2
1.2 Approaches and Challenges in Testing and Verification of Auto- mated Driving Systems	5
1.2.1 Summary of Contributions	12
1.2.1.1 Optimization-guided Automatic Test Generation for Automated Driving Systems	13
1.2.1.2 Requirements-driven Testing of Autonomous Vehicles with Machine Learning Components	15
1.2.2 Other Contributions	16
2 SYSTEMS AND REQUIREMENTS	20
2.1 Systems and Simulation	20
2.2 System Requirements	21
2.2.1 Temporal logic requirements and robustness	22
2.2.1.1 Linear Temporal Logic	22
2.2.1.2 Metric Temporal Logic / Signal Temporal Logic	23
2.2.1.3 Robustness for MTL / STL	25
2.3 Testing and Falsification methods	27

CHAPTER	Page
3 OPTIMIZATION-GUIDED AUTOMATIC TEST GENERATION FOR AUTOMATED DRIVING SYSTEMS	30
3.1 Introduction	30
3.2 Problem Definition	32
3.3 Falsification-based Approach to Test Generation	34
3.3.1 S-TALiRo	35
3.3.2 Simulation Configuration	36
3.3.3 Simulation Engine	37
3.3.4 Cost Function to Detect Boundary-case Collisions	37
3.3.5 Case Study	40
3.3.5.1 Simulation Configuration for the Case Study	40
3.3.5.2 Simulation Engine for the Case Study	41
3.3.5.3 Sensor Setup	41
3.3.5.4 Vehicle Controller with Collision Avoidance	42
3.3.5.5 Motion Controller for the Agent Vehicle	43
3.3.5.6 Experimental Results	44
3.4 Rapidly-exploring Random Trees for Testing Automated Driving Systems	46
3.4.1 Overview of the Approach	47
3.4.1.1 Initializing the Search	48
3.4.1.2 Information Stored on RRT Tree Nodes	50
3.4.1.3 Sampling a Target Path Segment	51
3.4.1.4 Selecting the Best Node from the Search Tree	53
3.4.1.5 Simulating the System	55

CHAPTER	Page
3.4.1.6 Cost Function	55
3.4.1.7 Transition Check Function	57
3.4.1.8 Novelty Function	58
3.4.1.9 Termination Condition	59
3.4.2 Case Studies	61
3.4.2.1 Case Study 1	61
3.4.2.2 Case Study 2	65
3.5 Functional Gradient Descent-based Approach	71
3.5.1 Definitions	72
3.5.2 Solution	73
3.5.3 Case Study	76
3.5.3.1 Gradient Descent Computation	78
3.5.3.2 Stop and Go Scenario	81
3.5.3.3 Experiment Results	81
3.6 Related Work	85
3.7 Review and Discussion on Optimization Methods	87
3.8 Conclusions and Future Work	92
4 REQUIREMENTS-DRIVEN TESTING OF AUTONOMOUS VEHI- CLES WITH MACHINE LEARNING COMPONENTS	96
4.1 Introduction	96
4.2 Preliminaries	100
4.2.1 Robustness-Guided Model Checking (RGMC)	100
4.2.2 Falsification and Critical System Behaviors	101
4.2.3 Covering Arrays	101

CHAPTER	Page
4.3 Requirements	103
4.3.1 STL Requirements.....	103
4.3.2 Development Process Support	109
4.4 Framework.....	112
4.5 Testing Application.....	118
4.6 Related work.....	132
4.7 Conclusions	135
5 A TUTORIAL ON SIM-ATAV	137
5.1 An Overview of Sim-ATAV.....	137
5.2 Installation Instructions.....	139
5.3 Reference Manual	144
5.3.1 Simulation Entities	144
5.3.1.1 Simulation Environment	146
5.3.1.2 Road	148
5.3.1.3 Vehicle	149
5.3.1.4 Sensor	152
5.3.1.5 Pedestrian	155
5.3.1.6 Fog.....	157
5.3.1.7 Road Disturbance	157
5.3.1.8 Generic Simulation Object	159
5.3.2 Configuring the Simulation Execution	160
5.3.2.1 Additional Controller Parameters	160
5.3.2.2 Heartbeat Configuration	162
5.3.2.3 Data Log Item	163

CHAPTER	Page
5.3.2.4 Initial State Configuration	165
5.3.2.5 Viewpoint Configuration	166
5.3.2.6 Simulation Configuration	167
5.3.3 Executing the Simulation	169
5.4 Combinatorial Testing	170
5.5 Falsification / Search-based Testing	175
5.5.1 Connecting to MATLAB	176
5.5.2 Connecting to S-TALiRO	178
5.6 Other Remarks	181
BIBLIOGRAPHY	182

LIST OF TABLES

Table	Page
1. SAE International’s Definitions for Levels of Automation in Automated Driving Systems.	3
2. Data Stored on the RRT Nodes.	50
3. Results from Autonomous Driving Tests Using Virtual Framework.	125
4. Simulation Environment Class.	147
5. Simulation Entity: Road.	148
6. Simulation Entity: Vehicle.	149
7. Simulation Entity: Sensor.	152
8. Simulation Entity: Pedestrian.	155
9. Simulation Entity: Fog.	157
10. Simulation Entity: Road Disturbance Object.	158
11. Simulation Entity: Generic Simulation Object.	159
12. Controller Parameters.	161
13. Heartbeat Configuration.	162
14. Data Item Description.	163
15. Initial State Configuration.	165
16. Viewpoint Configuration.	166
17. Simulation Configuration.	168
18. A List of Sim-ATAV Methods for Loading Test Cases from Csv Files.	171
19. Describing Test Parameters for Creating a Covering Array.	172

LIST OF FIGURES

Figure	Page
1. A High-Level Architecture of an Automated Driving System.	2
2. An Unavoidable Collision Example.	13
3. An Overview of the Framework Architecture.	35
4. Sensor Placement.	42
5. History of the Vehicles Before a Collision Instance.	45
6. The Input for Target Lateral Position of the Agent Vehicle.	45
7. Flowchart Illustrating the RRT-Based Approach.	49
8. Sampling a Waypoint.	51
9. Sampling a Target Path Segment.	52
10. Sampling a Target Path Segment with Space Constraints.	52
11. Cost Function to Guide the Search toward the Boundary between Collisions and near Collisions.	58
12. Initial States of the Vehicles in the Simulation Setup for Case Study 1.	62
13. Ego Vehicle Sensor Setup for Case Study 1.	63
14. The Minimum Cost Result Returned by the RRT-Based Approach in Case Study 1.	65
15. The Minimum Cost Result Returned by the Falsification-Based Approach in Case Study 1.	65
16. Comparison of the Minimum Cost Achieved by the Falsification Approach and the RRT-Based Approach in Case Study 1.	66
17. Initial States of the Vehicles in the Simulation Setup for Case Study 2.	66
18. Ego Vehicle Sensor Setup for Case Study 2.	67
19. One of the Collision Cases Discovered by the RRT-Based Approach.	69

Figure	Page
20.The Small-Speed Collision Discovered by the Falsification-Based Approach...	69
21.One of the Cases Where Falsification-Based Approach Was Stuck in a Local Minimum.	69
22.Comparison of the Minimum Cost Achieved by the Falsification Approach and the RRT-Based Approach in Case Study 2.	70
23.Solution Overview.	74
24.Accelerations for Original <i>Stop and Go</i> Scenario.....	81
25.Comparison of the Gradient Descent-Based Approach with Simulated Annealing	83
26.Agent Vehicle Acceleration and Resulting Worst-Case Performance.	84
27.Accelerations of Vehicles under Updated Input.....	84
28.An Illustration of Slow Convergence of the Steepest Descent Algorithm.	91
29.Specific Configurations Impacting DNN Performance.	104
30.Robotic Pedestrian Surrogate Target with a Toyota Autonomous Vehicle. ...	110
31.Overview of the Simulation Environment.	113
32.Outputs from the SqueezeDet DNN, Based on a Synthesized Camera Image..	114
33.Sensor Fusion Outputs.	115
34.Flowchart Illustrating the Combinatorial Testing.....	117
35.Flowchart Illustrating the Falsification.	118
36.Overview of the Scenario 1.	120
37.Overview of the Scenario 2.	122
38.Robustness Guided Falsification Utilizes Global Optimization Techniques to Guide the Test Cases toward Falsification.	127
39.Time-Ordered Images from the Falsifying Example on Model M_2	128
40.Analysis of Falsification for Model M_3	131

Figure	Page
41.LIDAR Reflection Points versus Distance to Various Pedestrian and Pedestrian-Like Targets.	134
42.An Overview of Sim-ATAV.	138
43.A View from the Generated Scenario for the Running Example.	145

Chapter 1

INTRODUCTION

1.1 Automated Driving Systems

Driver assistance systems such as lane keeping, adaptive cruise control, pedestrian/obstacle collision avoidance, automatic lane change, emergency braking systems and many more are being used in high-end modern automotive systems.

Fully automated driving systems are in a stage of rapid research and development spanning a broad range of maturity from simulations to on-road testing and deployment. Prototype autonomous vehicles (self-driving cars) have already driven millions of miles on public roads [179, 70]. Automated Driving Systems (ADS) are expected to have a significant impact on the vehicle market and the broader economy and society in the future. The expected primary impacts of ADS include a significant reduction in the number of fatalities and injuries caused by the traffic accidents, increased mobility for disabled and/or elderly people, reduction in transportation costs, and increased road capacity [59, 119].

1.1.1 Levels of Automation

The level of automation in ADS can vary from only creating warnings to fully automated driving. Definitions for the levels of automation by SAE International are given in Table 1 which is taken from SAE International’s report J3016 [153]. I use the term *autonomous vehicles* for the fully-automated driving systems, *i.e.*, the systems

that are SAE level 4 and 5, for which the system is responsible for all aspects of the driving task in its operating domain.

1.1.2 Automated Driving Systems Architecture

Figure 1 illustrates a (possible) high-level view of a typical ADS architecture. Currently, there is no unique correct approach to the development of ADS. Hence, the architecture we discuss here would probably differ between implementations. Although every organization has its own approach to the development of ADS, *e.g.*, using artificial intelligence vs. logic-based reasoning or differences in the selection of sensor setup, a generally accepted approach partitions the problem of automated driving into the subproblems of perception, planning, and control [179, 70, 133].

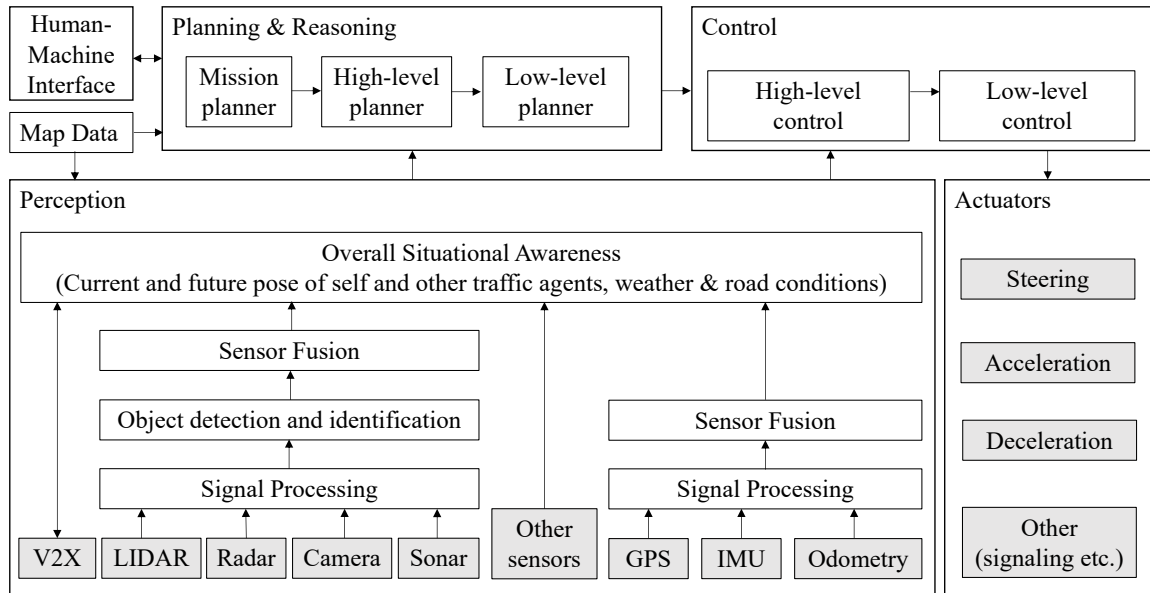


Figure 1. A high-level architecture of an automated driving system.

For the perception of the environment and other road occupants, *i.e.*, *agents*,

Level	Name & Definition	Control	Monitoring of driving environment	Fallback Performance of Dynamic Driving Task	System Capability (Driving Modes)
<i>Human driver monitors the driving environment (Levels 0-2)</i>					
0	No automation The full-time performance by the human driver of all aspects of the dynamic driving task, even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	N/A
1	Driver Assistance The driving mode-specific execution by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task	Human driver & System	Human driver	Human driver	Some driving modes
2	Partial Automation The driving mode-specific execution by one or more driver assistance systems of both steering and acceleration/deceleration using information about the driving environment and with the expectation that the human driver perform all remaining aspects of the dynamic driving task	System	Human driver	Human driver	Some driving modes
<i>Automated driving system ("system") monitors the driving environment (Levels 3-5)</i>					
3	Conditional Automation The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task with the expectation that the human driver will respond appropriately to a request to intervene	System	System	Human driver	Some driving modes
4	High Automation The driving mode-specific performance by an automated driving system of all aspects of the dynamic driving task, even if a human driver does not respond appropriately to a request to intervene	System	System	System	Some driving modes
5	Full Automation The full-time performance by an automated driving system of all aspects of the dynamic driving task under all roadway and environmental conditions that can be managed by a human driver	System	System	System	All driving modes

Table 1. SAE International’s definitions [153] for levels of automation in automated driving systems.

LIDAR, radar, camera, and sonar sensors are commonly used. Each of these sensors has its own strengths and weaknesses. For instance, while an object detection and classification algorithm on camera images can give good results under good lighting conditions, a LIDAR may be more reliable for dark environments. On the other hand, while a LIDAR normally gives more accurate readings than a radar, under rainy conditions a radar can be more reliable than a LIDAR. The typical refresh rates for the sensors also vary. In order to make the best use of several sensor data and the estimations based on the previous detections, sensor fusion algorithms are commonly used. A sensor fusion algorithm may involve logic-based rules as well as estimation algorithms like Kalman filters and particle filters together with expected motion models of the agents [165, 77]. Development of a good sensor fusion algorithm is a challenging job. The perception of the environment may also rely on vehicle-to-vehicle and vehicle-to-infrastructure communication systems where available.

For the estimation of the vehicle’s self-pose, commonly used sensors are GPS, Inertial Measurement Unit (IMU), and wheel odometry sensors. Sensor data is generally noisy, and typically, the measurements from different sensors are combined with previous estimations at a sensor fusion level to generate better estimations [165, 175].

Once a general picture of the self-pose and the environment is generated, this information is generally used at the planning level to generate a target path and velocity. In the planning level, if available, map information and user inputs, *e.g.*, target location, are used to generate a mission which can be at a very high-level such as “go to the location x by using the minimum amount of fuel”. Combining the mission with the current situation, a high-level plan, *e.g.*, a target path, is generated. Because driving environments are generally highly dynamic, the plan should be frequently

updated with the constraints of the current environment, *i.e.*, the other vehicles or unexpected road conditions. A low-level planner is responsible for creating a shorter-term local plan with regard to these constraints [27].

In the control level, the plan is executed and signals to actuators are generated. In the high-level control, optimal control algorithms like Model Predictive Control (MPC) may be used. However, because optimal control is generally slower to compute and update, a low-level controller, *e.g.*, a Proportional-Integral-Derivative (PID) controller, is typically used to update the actuator signals with high frequency [114, 65].

Considering all the hardware, software and the vehicle dynamics, an ADS is a complex Cyber-Physical System (CPS), *i.e.*, a system in which the dynamics of physical components and software components continuously interact with each other.

1.2 Approaches and Challenges in Testing and Verification of Automated Driving Systems

An ADS that is operating in a traffic environment and/or carrying passengers is a safety-critical system. This means that some types of failures in such a system may cause harm to humans. Hence, guaranteeing safe operation of these systems is crucial. Both governmental and non-governmental organizations are grappling with the unique requirements of these new, highly complex systems, as they have to operate safely and reliably in diverse driving environments. Government and industry-sponsored partnerships have produced a number of guiding documents and clarifications, such as NHTSA [129], SAE [152], CAMP [33], NCAP [164], PEGASUS [137].

Testing of ADS is an area of active research and the research community has been contributing to the development of methodologies for testing ADS. As addressed

by Bengler *et al.* [24], testing autonomous vehicles is a challenging problem which cannot be efficiently handled with conventional approaches. According to Maurer and Winner [120], considering the pace of functional growth in DAS, lack of efficient testing can affect time-to-market for more advanced systems like fully-automated vehicles. Stellet *et al.* [160] surveyed existing approaches to testing such as simulation-only, X-in-the-loop and augmented reality approaches, as well as test criteria and metrics (see also [190]). The publications [106], [28], and [176] provide in-depth discussions on the challenges of safety validation for autonomous vehicles, arguing that virtual testing should be the main target for both methodological and economic reasons.

The expected properties and functionality for a system are specified by a set of system *requirements*. However, specifying complete and unambiguous requirements for a system is a difficult task, especially if the system consists of multiple complex subsystems. *Formal requirements*, such as temporal logic specifications, can be used to remove ambiguity from the requirements, and to enable the mathematical analysis the system behaviors with respect to its requirements. A set of formal requirements can also be mathematically analyzed for vacuity or conflicts between requirements [42]. The use of formal requirements for a system allows the applicability of *formal methods*, which are mathematical approaches, to prove or disprove that the system satisfies its requirements [15]. The task of using formal methods to provide mathematical guarantees of the correctness of a system is called *formal verification* [163]. I use the terms *verification* and *formal verification* interchangeably when referring to formal verification.

Besides itself being a complex CPS, an ADS is also in continuous interaction with the road surface, other road users, *e.g.*, pedestrians, and other vehicles, and is also affected by the weather and lighting conditions. At any moment in a typical

traffic environment, the future behaviors of other vehicles and pedestrians contain uncertainties. Hence, if we want to model an ADS in its environment, the overall environment would be a large highly nonlinear system with uncertainties. Furthermore, the evolution of this large system is not bounded with time, *i.e.*, the operation is not limited within a certain time range. Formal verification of such a system, *i.e.*, a system which is a combination of multiple hybrid subsystems with nonlinear dynamics and uncertainties, for unbounded time using reachability analysis is an undecidable problem [82, 83, 15]. This means, there cannot be a computer program that can always give the correct answer to the question of whether such a system is correct or not. Despite the undecidability results, there is some work in the literature which aim verification of some components of ADS with some assumptions and simplifications. In the following, I present applications of several different approaches to the verification problem of ADS and the challenges in verifying ADS using these approaches.

Reachability analysis computes the set of all reachable states of a system and checks whether the set of reachable states intersects with the set of incorrect states [20, 21]. Althoff *et al.* use reachability analysis techniques for safety verification of autonomous vehicles [11, 12, 13, 148]. In [14], reachability analysis is used to generate critical simulations with the aim of minimizing the drivable area for the vehicle under test. Reachability analysis for nonlinear systems involve linearization, over-approximations and it can only be applied for a bounded time. Possibilities in real-world driving environments should be captured in the verification of ADS. Hundreds of pedestrians may be crossing at intersections or walking on the sidewalks. Tens of vehicles with different shapes and drivers with different driving characteristics may be in the surroundings of the ADS under test. The road and weather conditions can vary dramatically. All this openness of the operating environment of an ADS

makes it very difficult to account for all possible behaviors with reachability analysis tools without being too conservative or too optimistic in the assumptions. In a real-world driving environment, drivers make some optimistic assumptions on the behavior of other traffic occupants, and they take action when they realize that their assumptions do not hold. Mathematically capturing these assumptions and real-world driving behaviors correctly is challenging. Overly optimistic assumptions may result in verifying a system that may not actually be safe when these assumptions do not hold. On the other hand, pessimistic assumptions would result in very conservative results. Hence, the biggest challenge for bounded time verification of ADS using reachability analysis techniques is scalability and conservativeness of the results.

Automated theorem proving, which is another approach that is used for formal verification, utilizes mathematical descriptions of system logic and dynamics together with formal requirements to compose mathematical proofs of correctness of the system [141]. Loos *et al.* propose and verify safety properties of an adaptive cruise control system by using a *semi-automated theorem prover* [142]. They consider only the longitudinal motion of the Ego vehicle under the assumptions that all the vehicles on the road are automated, the system dynamics and limits on acceleration and deceleration values are well known, sensor and communication network is accurate and updates within a time limit and any instance a vehicle first appears in front of the Ego vehicle is safe [116]. A problem with theorem provers is the need for a high level of user interaction even for a simple example [19]. Considering all the subsystems of an ADS the openness of the driving environment, the overall system becomes very complex which makes the applicability of automated theorem proving questionable. For verification of ADS as an overall system, sensors on the vehicle and their performance under various conditions must be considered. For instance, there is

an enormous number of ways that a LIDAR point cloud can be affected by different materials and shapes of the objects in the environment as well as the pose of the ego vehicle. Another challenge in applying verification techniques such as automated theorem proving or reachability analysis is in providing mathematical descriptions of such details to discover potential problems originating from the sensor systems.

Temporal logic model checking is a verification approach that relies on the exhaustive search of the state space of a finite state system with the target of answering the question whether the system is correct with respect to its temporal logic requirements [35, 146, 37]. However, the state space of a system grows exponentially with the number of states in the system, which is a phenomenon called the *state explosion problem*. Because of the state explosion problem, model checking is unusable for most industrial-size systems [36]. There are several heuristics aiming to overcome the state explosion problem. A Monte Carlo algorithm for model checking that performs a random walk on the system state transition graph is proposed in [75]. An approach that combines motion planning and model checking for the temporal logic falsification of hybrid systems is proposed in [139].

Automated driving systems generally use machine learning components such as deep neural networks (DNNs) for evaluating sensor outputs such as camera images or LIDAR point clouds for object detection and classification as well as for estimating future behaviors of other road occupants. Verification of such machine learning components is still an open problem [157]. Although some assumptions can be made on the performance of these components with the target of verification of the overall system without verifying machine learning components in isolation, the validity of such assumptions would also be questionable. For instance, one can assume that a DNN may fail to detect an object for some time and aim to verify the system under this

assumption. But, what if the DNN detects the object at a different location, instead of failing to detect the object, and misguides the trajectory estimation algorithms? Capturing all such possibilities with the state of the art verification approaches is impractical, if not impossible.

Despite their limitations, the aforementioned verification approaches are very powerful techniques and such techniques are needed for guaranteeing safe operation of ADS. Ideally, both automated theorem proving and reachability analysis techniques should be used whenever possible to provide guarantees for some subsystems or the overall system under some assumptions and limitations in the driving environment. To overcome the limitation of verification techniques, they can be complemented with testing-based approaches.

Simulation-based falsification is an automated test generation approach that aims to find falsifications of formal system requirements using simulations of the system. A *falsification* of a formal requirement, *i.e.*, a *falsifying trace*, is a system simulation trace that does not satisfy the corresponding requirement. Hence, a falsifying trace is a counterexample to the correct system behavior. A system is said to be *falsified* if a falsifying trace is discovered for at least one of the formal system requirements. Falsification techniques are considered as *semi-formal* methods. They are formal in the sense that they rely on mathematical analysis of the systems with respect to formal requirements and they can provide counterexamples (falsifications) that disprove that a system satisfies its requirements. They are best effort approaches toward verification, and generally, they cannot prove, *i.e.*, they cannot *verify*, the correctness of a system. However, it may be possible to provide statistical guarantees of correctness for stochastic cyber-physical systems [5].

Although falsification approaches cannot verify system properties, they are more

scalable than formal verification techniques and generally are easier to utilize, and they give valuable information on the safety properties of the system under test. Another advantage of falsification approaches is that as long as the system can be simulated, we don't need to know all the details of the system under test. By utilizing optimization techniques for falsification, we can guide automatic test generation toward incorrect system behaviors and we can have increased confidence in the correctness of the system and identify risky, *i.e.*, safe but close to unsafe, operating conditions. More detail on the falsification techniques is provided in Section 2.

The problem of verification of ADS is an open problem. As discussed above, formal verification methods have shortcomings in verifying a complex ADS within an open driving environment that contains many uncertainties. On the other hand, falsification methods cannot provide mathematical guarantees for the correctness of the system even when they cannot falsify the system.

Both verification and falsification approaches have their own advantages and disadvantages as described above, and they are both valuable in the process of checking whether an ADS design and implementation is correct with respect to its requirements. Hence, they can be used complementary to each other. While some components or subsystems can formally be verified, the overall system that potentially consists of formally verified, non-verified and even black-box components can be tested systematically using falsification techniques. The approaches that are presented in this dissertation are on the falsification of ADS.

1.2.1 Summary of Contributions

The typical operating domain of automated driving systems is an open traffic environment with many other road occupants for which the behavioral models are not available. Besides other road occupants, road specific parameters such as friction, slope, surface type, other environmental conditions such as lighting amount and angle, the existence of fog and the corresponding visibility, types of buildings or trees can all vary significantly from time to time and from location to location. Automatically identifying unintuitive configurations and behaviors that lead to challenging situations for the vehicle under test (Ego vehicle) is a difficult task in such an open environment.

One of the main contributions presented in this dissertation is the proposed concepts for the formulation of adversarial configurations and behaviors in the simulation environment aiming to automatically identify critical cases for the Ego vehicle. For this purpose, several approaches to formulate the problem as an optimization problem are proposed. Furthermore, new heuristic methods are proposed that are empirically shown to improve the test generation performance. While identifying the collision cases is important, because of the nature of the driving environment, an automated vehicle cannot avoid all collisions. For instance, a collision with a vehicle that loses control and drives into the automated vehicle may not be avoidable. Figure 2 illustrates an unavoidable collision example. However, identifying the boundaries between the collisions that are barely avoided or the collisions that could have been avoided with minor changes in the control or environment would be valuable for the engineering teams to improve the safety-related capabilities of the system.

Another contribution of this dissertation is the proposed cost functions that aim to identify such boundaries (between safe and unsafe situations) by guiding the test

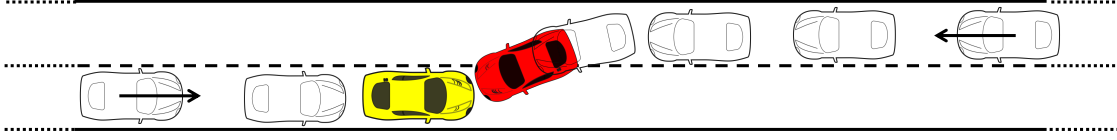


Figure 2. An unavoidable collision example. Red car suddenly moves into the Ego vehicle’s lane at a very short distance.

generation process toward those boundaries. Yet another contribution is the provided formal requirements for ADS that define the correct operation at the system level, subsystem (perception system) level and acceptable impacts of subsystem-level failures to the overall system operation.

Finally, a publicly available, open-source simulation-based framework is developed that can be used to experiment with different test generation approaches in a 3D modeled environment, in which physical models for several sensor types and vehicles are available. Webots [39, 123], which is an open-source robotics simulator, is utilized for simulating the sensor and vehicle models in a 3D modeled environment.

I summarize my contributions and publications on my published work in the following.

1.2.1.1 Optimization-guided Automatic Test Generation for Automated Driving Systems

- *Cumhur Erkan Tuncali, Theodore P. Pavlic, and Georgios Fainekos, Utilizing S-TALIRO as an Automatic Test Generation Framework for Autonomous Vehicles, in IEEE Intelligent Transportation Systems Conference (ITSC), 2016 [173].*

In this paper, I developed a simulation-based testing framework in MATLAB

which uses S-TALiRO to automatically generate test cases for testing motion controllers of ADS. The goal of the framework in this work is to search over agent vehicle trajectories to find boundary-case collisions or near-miss situations, *i.e.*, the situations that are on the boundary between safe and unsafe operations. I proposed a robustness metric that is used as the cost function in a global optimization setting to guide the generation of vehicle trajectories for test cases towards the boundary between safe and unsafe operations. I conducted a case study using Matlab simulations.

- ***Cumhur Erkan Tuncali, Shakiba Yaghoubi, Theodore P. Pavlic, and Georgios Fainekos, Functional Gradient Descent Optimization for Automatic Test Case Generation for Vehicle Controllers, IEEE International Conference on Automation Science and Engineering (CASE), 2017 [174].***

In this paper, I developed a hierarchical test generation framework that combines analytical functional gradient computations with simulations. A system performance function is used as the cost function that is to be minimized. Starting from a random location, a descent direction is computed on a simplified model of the system dynamics, on which the gradients can be analytically computed. Then, a bisection technique is used to search for a minimal performance on the simulations of a high-fidelity model of the system. The system performance evaluations from the simulation outputs are used to guide the search by changing the bisection step size. Once a minimal performance is found with the bisection technique, the whole process is restarted at another random location which is selected by low-discrepancy sampling. I implemented a full-range adaptive cruise

control from literature and set up a case study to evaluate the results of this work.

1.2.1.2 Requirements-driven Testing of Autonomous Vehicles with Machine Learning Components

- **Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski, *Sim-ATAV: Simulation-Based Adversarial Testing Framework for Autonomous Vehicles*, in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (HSCC)*, 2018 [168].**

Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito and James Kapinski, *Simulation-based Adversarial Test Generation for Autonomous Vehicles with Machine Learning Components*, in *IEEE Intelligent Vehicles Symposium (IV)*, 2018 [169].

In this line of work, I developed a framework, Sim-ATAV, which utilizes a robotics simulation platform, Webots [39], and S-TALiRO [60] for falsification of autonomous driving vehicles that have machine learning-based perception systems. Sim-ATAV is publicly available as an open-source project [167]. I proposed and studied a test generation approach that combines covering arrays with optimization-guided falsification. The proposed approach, which uses covering arrays with an aim to find better initial samples for optimization-guided falsification, enabled a search over discrete and discretized continuous parameters and increased the probability of finding interesting cases with a smaller number of simulations compared to uniform random or only optimization guided testing.

1.2.2 Other Contributions

My other contributions that are not directly related to the topics covered in this dissertation.

- *Cumhur Erkan Tuncali, Georgios Fainekos, and Yann-Hang Lee, **Automatic Parallelization of Simulink Models for Multi-core Architectures**, in IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 [170].*

This paper addresses the problem of parallelizing existing single-rate Simulink® models for embedded control applications on multi-core architectures considering communication cost between blocks on different CPU cores. I developed a Mixed Integer Linear Programming (MILP) formulation for computing an optimal mapping of Simulink model blocks on different CPU cores that aims to minimize the overall Worst-Case Execution Time (WCET) while maintaining functional equivalence to the original model. I proposed three heuristics to reduce MILP solver time for complex models. The input to the MILP formulation is a dependency graph of Simulink model blocks with WCET information for each block given on the corresponding node and the inter-core communication time between two dependent blocks given on the directed edge that connect those blocks. The output of the optimization is a mapping of each Simulink model to a specific CPU core. I developed a tool in Matlab for flattening a given Simulink model, creating the aforementioned dependency graph and using MILP outputs to create a new Simulink model for each target CPU core that contains all (and only) the blocks that are assigned to that specific core. The tool I developed also

introduces necessary inter-core communication and synchronization mechanisms into the generated Simulink models, where necessary to guarantee functional equivalence to the original model execution. I studied the scalability and efficiency of the MILP formulation on synthetic directed acyclic graphs. I studied the applicability of the tool on real-world problems on (1) “Fault-Tolerant Fuel Control System” demo from Simulink and (2) a Diesel engine controller model from Toyota [90].

- *Cumhur Erkan Tuncali, Georgios Fainekos, and Yann-Hang Lee, **Automatic Parallelization of Multirate Block Diagrams of Control Systems on Multicore Platforms**, in ACM Transactions on Embedded Computing Systems (TECS), 2016 [170].*

This journal article extends the previous work to multi-rate Simulink models. In this work, I extended the previous MILP formulation for single-rate Simulink models to multi-rate Simulink models. I modified the Matlab tool from the previous work to generate a separate dependency graph for each sampling rate of the model and generate Simulink models for each core with multiple sampling rates.

- *Cumhur Erkan Tuncali, Bardh Hoxha, Guohui Ding, Georgios Fainekos, and Sriram Sankaranarayanan, **Experience Report: Application of Falsification Methods on the UxAS System**, NASA Formal Methods Symposium, 2018 [171].*

In this paper, our experiences in applying falsification methods over the Unmanned Systems Autonomy Services (UxAS) system is presented. UxAS is

a collection of software modules that enables complex mission planning for multiple vehicles. In this work, I developed new modules for UxAS that enable monitoring of the unmanned air vehicle states. I developed a mechanism that enables the user to define ranges of parameters for simulation configurations of UxAS. I developed an interface between S-TALiRO and UxAS that can start simulations of UxAS with a sampled configuration and collect values of monitored parameters as a simulation trace. I also developed example MTL requirements for several UxAS scenarios. I conducted experiments with the developed framework.

- *Cumhur Erkan Tuncali, James Kapinski, Hisahiro Ito, and Jyotirmoy V Deshmukh, Reasoning about safety of learning-enabled components in autonomous cyber-physical systems, IEEE 56th Design Automation Conference (DAC), 2018 [172].*

In this paper, a simulation-based approach is presented for generating barrier certificate functions for safety verification of cyber-physical systems (CPS) that contain neural network-based controllers. Simulation outputs are used in a linear programming setting to generate candidate safety barrier functions, which are then evaluated with a Satisfiability Modulo Theories (SMT) solver with respect to the analytical representation of the system. A level set of the safety barrier function is computed that would separate safe and unsafe system states and returned as a safety certificate for the system. In this work, I created different sized Neural Networks (NNs) and used a Covariance Matrix Adaptation-Evolution Strategy (CMA-ES) based policy search technique to train these NNs as line following controllers for a Dubins car. I extended previous work

in the literature to the systems with feed-forward neural network controllers. I conducted experiments to evaluate the scalability of the proposed approach on different sized NNs.

- *Joseph Campbell, **Cumhur Erkan Tuncali**, Peng Liu, Theodore P Pavlic, Umit Ozguner, and Georgios Fainekos, **Modeling concurrency and reconfiguration in vehicular systems: A π -calculus approach**, in IEEE International Conference on Automation Science and Engineering (CASE), 2016 [29].*

In this paper, a modeling framework is proposed where communication and system reconfiguration is modeled through π -calculus expressions while the closed/loop control systems are modeled through hybrid automata. In this work, I contributed to the modeling of control and physical systems as a hybrid automaton, I implemented a Model Predictive Controller (MPC) from the literature and I conducted a case study with different types of vehicles that are controlled by the implemented MPC.

Chapter 2

SYSTEMS AND REQUIREMENTS

2.1 Systems and Simulation

In this section, we describe the notation used in this dissertation to describe models of systems and their simulations. We denote the model of a system as a tuple $\mathcal{M} = (X, \mathbf{U}, P, \text{sim})$ where X is the set of system states, \mathbf{U} is the set of system inputs, P is the set of system parameters, and sim is the system simulation function.

The set of system variables, $X = X^{(C)} \cup X^{(D)}$, is the union of the set of continuous-valued variables, $X^{(C)} = \{x_1^{(C)}, \dots, x_N^{(C)}\}$, and the set of discrete-valued variables, $X^{(D)} = \{x_1^{(D)}, \dots, x_K^{(D)}\}$. We assume that each $x_i^{(C)} \in \mathcal{X}_i^{(C)} \subseteq \mathbb{R}$, and each $x_i^{(D)} \in \mathcal{X}_i^{(D)}$, where $\mathcal{X}_i^{(D)}$ is some finite domain. The state space of the system is denoted by $\mathcal{X} = \mathcal{X}^{(C)} \times \mathcal{X}^{(D)}$, where $\mathcal{X}^{(C)} = \mathcal{X}_1^{(C)} \times \dots \times \mathcal{X}_N^{(C)}$ and $\mathcal{X}^{(D)} = \mathcal{X}_1^{(D)} \times \dots \times \mathcal{X}_K^{(D)}$. A state vector at time t_i is denoted by $\mathbf{x}_i \in \mathcal{X}$.

The set of system inputs is $\mathbf{U} = \{u_1, \dots, u_M\}$, and the corresponding input space is $\mathcal{U} = \mathcal{U}_1 \times \dots \times \mathcal{U}_M$ where M is the dimension of the input space. An input vector at time t_i is denoted by $\mathbf{u}_i \in \mathcal{U}$.

The set of system parameters, $P = P^{(C)} \cup P^{(D)}$, is the union of the set of real-valued parameters, $P^{(C)} = \{p_1^{(C)}, \dots, p_W^{(C)}\}$, and the set of discrete-valued parameters, $P^{(D)} = \{p_1^{(D)}, \dots, p_V^{(D)}\}$. Each $p_i^{(C)} \in \mathcal{P}_i^{(C)} \subseteq \mathbb{R}$ and each $p_i^{(D)} \in \mathcal{P}_i^{(D)}$, where $\mathcal{P}_i^{(D)}$ is some finite domain. The parameter space is denoted by $\mathcal{P} = \mathcal{P}^{(C)} \times \mathcal{P}^{(D)}$, where $\mathcal{P}^{(C)} = \mathcal{P}_1^{(C)} \times \dots \times \mathcal{P}_W^{(C)}$ and $\mathcal{P}^{(D)} = \mathcal{P}_1^{(D)} \times \dots \times \mathcal{P}_V^{(D)}$. A parameter vector is denoted by $\mathbf{p} \in \mathcal{P}$.

Given $\mathbf{x} \in \mathcal{X}$, $\hat{\mathbf{x}} = \text{sim}(\mathbf{x}, \mathbf{u}, \mathbf{p}, t)$ is the state reached starting from the system state \mathbf{x} after time $t \in T$ under the input vector $\mathbf{u} \in \mathcal{U}$ and the parameter vector $\mathbf{p} \in \mathcal{P}$. Hence, the system simulation function is a mapping $\text{sim} : \mathcal{X} \times \mathcal{U} \times \mathcal{P} \times T \rightarrow \mathcal{X}$ where T is a discrete set of time samples t_0, t_1, \dots, t_N , with $t_i < t_{i+1}$. We call a sequence $\omega = (\mathbf{u}_0, t_0)(\mathbf{u}_1, t_1) \cdots (\mathbf{u}_N, t_N)$, where $t_i > t_{i-1}$, an input trace of \mathcal{M} . Given a model \mathcal{M} , an input trace of \mathcal{M} , ω , and a $\mathbf{p} \in \mathcal{P}$, a simulation trace of \mathcal{M} under input ω and parameters \mathbf{p} is a sequence

$$\mu = (\mathbf{x}_0, \mathbf{u}_0, \mathbf{p}, t_0)(\mathbf{x}_1, \mathbf{u}_1, \mathbf{p}, t_1) \cdots (\mathbf{x}_N, \mathbf{u}_N, \mathbf{p}, t_N),$$

where $\text{sim}(\mathbf{x}_{i-1}, \mathbf{u}_{i-1}, \mathbf{p}, t_{i-1}) = \mathbf{x}_i$ for each $1 \leq i \leq N$, and $\mathbf{x}_0, \mathbf{u}_0$ are the initial states and initial inputs of the system. We denote the set of all simulation traces of \mathcal{M} by $\mathcal{L}(\mathcal{M})$. For a given simulation trace μ , we call $\zeta = (\mathbf{x}_0, t_0)(\mathbf{x}_1, t_1) \cdots (\mathbf{x}_N, t_N)$ the state trace. For a set of equally distributed time samples, *i.e.*, $t_i - t_{i-1} = \Delta t, \forall i$, the state trace can be simplified as $\zeta = \mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_N$. Finally, we define the system output equation as $y = h(\mathbf{x}, \mathbf{u}, t)$ and the output trace corresponding to the μ as $\mathbf{y} = y_0, y_1, \dots, y_N$, where the function h maps the system states, inputs and time to the system outputs.

2.2 System Requirements

In order to design and test a system, a set of requirements that describe the properties of the system is needed. Those requirements can be in the natural language or in some mathematical form. Natural language requirements are generally easier to write. However, they can easily contain ambiguities. This brings the risk that the requirement developer, the engineer implementing the system and the tester can all interpret the same requirement differently. It is also difficult to develop tools that

can automatically analyze a system behavior with respect to its natural language requirements. On the other hand, formal requirements have the advantage of being unambiguous, and one can mathematically check system behaviors with respect to its formal requirements or analyze requirements with respect to each other, *e.g.*, checking conflicts, vacuity etc. [42].

2.2.1 Temporal logic requirements and robustness

Temporal logic is an extension of Boolean logic with time-related operators which add time constraints on Boolean expressions. Temporal logic formulas are one form of describing formal requirements for systems. Three commonly used variations of temporal logic are Linear Temporal Logic (LTL), Metric Temporal Logic (MTL), and Signal Temporal Logic (STL). LTL works on sequences of atomic propositions. MTL can describe real-time constraints on atomic propositions. STL can describe real-time constraints on predicates over real-valued signals.

2.2.1.1 Linear Temporal Logic

Linear temporal logic is an extension of Boolean logic with time constraints on sequences of atomic propositions [143]. LTL formulae can contain Boolean operators *conjunction* (\wedge), *disjunction* (\vee) and *negation* (\neg) and temporal operators *eventually* (\Diamond), *always* (\Box), *next* (\bigcirc), and *until* (\mathcal{U}). The temporal operators are evaluated at each sample of the sequence with respect to the future values of the atomic propositions in the sequence.

Let $k \in \mathbb{N}$ denote the position of samples in a given sequence, where $(k + 1)^{th}$

sample is the sample which immediately proceeds the k^{th} sample. For a sequence of the values of predicate π_i , let $\pi_i[k]$ denote the value of the predicate at the k^{th} sample, which can be true (\top) or false (\perp). Then, the temporal operators on predicates are evaluated as follows at the k^{th} sample of a sequence:

$$\begin{aligned}
\Diamond \pi_i = \top &\iff \exists j \geq k \text{ s.t. } \pi_i[j] = \top, \\
\Box \pi_i = \top &\iff \pi_i[j] = \top \forall j \geq k \\
\bigcirc \pi_i = \top &\iff \pi_i[k+1] = \top \\
(\pi_1 \mathcal{U} \pi_2) = \top &\iff (\exists l > k \text{ s.t. } \pi_2[l] = \top) \text{ and } (\pi_1[j] = \top \forall j \text{ s.t. } k \leq j < l).
\end{aligned}$$

2.2.1.2 Metric Temporal Logic / Signal Temporal Logic

Metric temporal logic works on the atomic propositions, *i.e.*, Boolean predicates, in real-time. For MTL, predicates are in the Boolean form, *i.e.*, a predicate ν_i is given as a signal of Boolean variables.

Signal Temporal Logic (STL) was introduced as an extension to MTL to reason about real-time properties of signals (simulation traces) (for an overview see [23]). Signal temporal logic works on predicates over real-valued signals. For STL, the predicates π are expressions built using the following grammar:

$$\pi ::= f(\mathbf{x}, \mathbf{u}, \mathbf{p}) \geq c \mid \neg \pi_1 \mid (\pi) \mid \pi_1 \vee \pi_2 \mid \pi_1 \wedge \pi_2$$

where f is a function and c is a constant in \mathbb{R} [117]. The difference between MTL and STL can be considered as a difference in how predicates are represented. An STL formula can be expressed as an MTL formula by appropriately defining a corresponding atomic proposition ν_i for each π_i of an STL formula where for some time t we have

$\nu_i = \top \iff \pi_i = \top$. Hence, in the following we will utilize MTL and STL formulas interchangeably based on the context and readability.

The temporal operators of MTL/STL include *eventually* ($\Diamond_{\mathcal{I}}$), *always* ($\Box_{\mathcal{I}}$) and *until* ($\mathcal{U}_{\mathcal{I}}$), where \mathcal{I} is a time interval that encodes timing constraints.

Definition 1 (MTL/STL Syntax) *Assume π is the set of predicates and \mathcal{I} is any non-empty interval of $\overline{\mathbb{R}}_+$. The set of all well-formed MTL or STL formulas is inductively defined as*

$$\varphi ::= \top \mid \pi \mid \neg\varphi \mid \phi_1 \vee \phi_2 \mid \phi_1 \mathcal{U}_{\mathcal{I}} \phi_2,$$

where π is a predicate, \top is true and $\mathcal{U}_{\mathcal{I}}$ is Until operator that is constrained to time interval \mathcal{I} .

A formula is evaluated at time t as follows:

$$\pi = \top \iff \begin{cases} \pi(t) = \top, & \text{for MTL,} \\ f(x_1(t), \dots, x_n(t)) > 0, & \text{for STL.} \end{cases}$$

$$\neg\varphi = \top \iff \varphi = \perp \text{ at time } t$$

$$\phi_1 \vee \phi_2 = \top \iff \phi_1 = \top \text{ at time } t \text{ or } \phi_2 = \top \text{ at time } t$$

$$(\phi_1 \mathcal{U} \phi_2) = \top \iff \exists t_1 \geq t \text{ s.t. } \phi_2 = \top \text{ at time } t + t_1 \text{ and } \phi_1 = \top \forall t_2 \in [t, t_1].$$

For the formulas ψ, ϕ , we define the following using syntactic manipulation:

$$\begin{aligned}
\psi \wedge \phi &\equiv \neg(\neg\psi \vee \neg\phi) \quad (\psi \text{ AND } \phi), \\
\perp &\equiv \neg\top \quad (\text{False}), \\
\psi \rightarrow \phi &\equiv \neg\psi \vee \phi \quad (\psi \text{ Implies } \phi), \\
\Diamond_{\mathcal{I}}\psi &\equiv \top U_{\mathcal{I}}\psi \quad (\text{Eventually } \psi \text{ within time interval } \mathcal{I}), \\
\Box_{\mathcal{I}}\psi &\equiv \neg\Diamond_{\mathcal{I}}\neg\psi \quad (\text{Always } \psi \text{ within time interval } \mathcal{I}), \\
\psi \mathcal{R}_{\mathcal{I}}\phi &\equiv \neg(\neg\psi U_{\mathcal{I}}\neg\phi) \quad (\psi \text{ Releases } \phi \text{ within time interval } \mathcal{I})
\end{aligned}$$

When we work in discrete-time, we use the discrete-time semantics of MTL/STL by applying the above definitions in the discrete time domain. For the discrete-time, we can also define the “next” operator (\bigcirc) for MTL or STL formulas in a similar fashion as it is defined for LTL, *i.e.*, $\bigcirc\pi_1 = \top \text{ at time } t \iff \pi_1 = \top \text{ at the sample proceeding the time } t$.

2.2.1.3 Robustness for MTL / STL

Each predicate π of an STL formula represents a subset in the space $\mathcal{X} \times \mathcal{U} \times \mathcal{P}$. In the following, we represent that set that corresponds to the predicate π using the notation $\mathcal{O}(\pi)$.

In [63], Fainekos and Pappas proposed robust semantics for STL formulas. Robust semantics (or robustness metrics) provide a real-valued measure of satisfaction of a formula by a trace in contrast to the Boolean semantics that just provides a *true* or *false* valuation. In more detail, given a trace μ of the system, its robustness w.r.t. a temporal property φ , denoted $\llbracket \varphi \rrbracket_{\mathbf{d}}(\mu)$ yields a positive value if μ satisfies φ and a negative value otherwise. Moreover, if the trace μ satisfies the specification φ , then

the robust semantics evaluate to the radius of a neighborhood such that any other trace that remains within that neighborhood also satisfies the same specification. The same holds for traces that do not satisfy φ .

Definition 2 (MTL / STL Robust Semantics) *Given a metric \mathbf{d} , trace μ of length N , where μ_i corresponds the value of the trace at time t_i , i.e., $\mu_i = (\mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i), \forall i \in [0, N]$, and $\mathcal{O} : \pi \rightarrow 2^{\mathcal{X} \times \mathcal{U} \times \mathcal{P}}$, the robust semantics of any MTL formula φ w.r.t μ_i is defined as follows:*

$$\begin{aligned}
\llbracket \top \rrbracket_{\mathbf{d}}(\mu_i) &:= +\infty \\
\llbracket \pi \rrbracket_{\mathbf{d}}(\mu_i) &:= \begin{cases} -\inf\{\mathbf{d}((\mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i), \mathbf{y}) \mid \mathbf{y} \in \mathcal{O}(\pi)\}, & \text{if } (\mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i) \notin \mathcal{O}(\pi) \\ \inf\{\mathbf{d}((\mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i), \mathbf{y}) \mid \mathbf{y} \in \overline{\mathcal{O}(\pi)}\}, & \text{if } (\mathbf{x}_i, \mathbf{u}_i, \mathbf{p}_i) \in \mathcal{O}(\pi) \end{cases} \\
\llbracket \neg\varphi \rrbracket_{\mathbf{d}}(\mu_i) &:= -\llbracket \varphi \rrbracket_{\mathbf{d}}(\mu_i) \\
\llbracket \phi_1 \vee \phi_2 \rrbracket_{\mathbf{d}}(\mu_i) &:= \max(\llbracket \phi_1 \rrbracket_{\mathbf{d}}(\mu_i), \llbracket \phi_2 \rrbracket_{\mathbf{d}}(\mu_i)) \\
\llbracket \bigcirc\varphi \rrbracket_{\mathbf{d}}(\mu_i) &:= \begin{cases} \llbracket \varphi \rrbracket_{\mathbf{d}}(\mu_{i+1}), & \text{if } i+1 \leq N \\ -\infty, & \text{otherwise} \end{cases} \\
\llbracket \phi_1 \mathcal{U}_{\mathcal{I}} \phi_2 \rrbracket_{\mathbf{d}}(\mu_i) &:= \max_{j \text{ s.t. } (t_j - t_i) \in \mathcal{I}} \left(\min \left(\llbracket \phi_2 \rrbracket_{\mathbf{d}}(\mu_j), \min_{i \leq k < j} \llbracket \phi_1 \rrbracket_{\mathbf{d}}(\mu_k) \right) \right)
\end{aligned}$$

Later, in [46] a relaxed notion of robustness was introduced for STL. Namely, over 1D signals, the two notions of robustness coincide. However, in multi-dimensional spaces, the robustness definition is replaced by simply $f(x) - \alpha$ for a given predicate $f(x) > \alpha$. For instance, for a predicate $2x < 1$, when the value of x is 1, STL robustness will result in $2 - 1 = 1$ which is the perturbation needed on the signal value for violation of the requirement, while the robustness value for MTL will be 0.5, which is the perturbation needed on the variable x for the violation.

A trace μ satisfies ϕ (denoted by $\mu \models \phi$), if $\llbracket \phi \rrbracket_{\mathbf{d}}(\mu_0) > 0$. On the other hand, a trace μ' falsifies ϕ (denoted by $\mu' \not\models \phi$), if $\llbracket \phi \rrbracket_{\mathbf{d}}(\mu'_0) < 0$. Algorithms to compute $\llbracket \varphi \rrbracket_{\mathbf{d}}$ have been presented in [63, 60, 46].

2.3 Testing and Falsification methods

Testing is to execute a system (or a simulation of the system) with some (test) parameters and to compare the system outputs with the desired behavior (system behaviors) to decide whether the system conforms to its requirements or not for the executed test cases. Optimization-guided falsification methods use optimization methods to automatically generate (and execute) test cases and seek a set of test parameters that would cause the system to fail its requirements.

There are approaches that compute a measurement of how well a system satisfies or how much does it violate its temporal logic specifications [63, 64, 149], which is defined as *robustness* as described in subsection 2.2.1.3. The relaxed notion of robustness in [46] is faster to compute in high-dimensional spaces, but it loses the semantic interpretation of the robustness neighborhoods. For instance, using simulations toward temporal logic verification through robust semantics for MTL formulas is discussed in [62]. A Monte Carlo optimization approach that performs a random walk on the input space and utilizes the robustness metric for increasing the efficiency of the test generation process for falsification of hybrid systems is proposed in [132]. Besides Monte Carlo approaches, another approach is to utilize functional gradient descent-based optimization with the target of minimizing the robustness value by computing a descent direction that is guaranteed to reduce the distance between an unsafe set of states and a specific location on a simulation trace [8]. A survey on

the approaches and tools for using temporal logic specifications for checking system behaviors is presented in [23].

Among the falsification approaches, *single-shooting* approaches are the ones that work on complete simulation traces, each starting from an initial condition. *Multiple-shooting* approaches work on multiple partial trajectories that start from different initial conditions. A multiple-shooting, Counterexample-Guided Abstraction Refinement (CEGAR)-based falsification approach is proposed in [191].

Two commonly known publicly available tools for falsification of cyber-physical systems that utilize the robustness values of simulation traces with respect to STL/MTL specifications are S-TALiRO [18] and Breach [45]. S-TALiRO and Breach are MATLAB[®] toolboxes for systematic testing of hybrid systems, *i.e.*, the systems that exhibit continuous and discrete dynamics. Both of these tools are single shooting approaches and they support various global optimization methods for minimizing the robustness function and, thus, for seeking a falsifying system trajectory. While S-TALiRO utilizes robust semantics for MTL, Breach utilizes robust semantics for STL. The optimization approaches that are supported by S-TALiRO include simulated annealing, cross-entropy, ant-colony optimization while Breach supports Nelder-Mead technique. S-TALiRO also supports coverage-based testing for discrete spaces [44] and conformance testing [6].

One important notion in testing is coverage, which is generally used as a measure of how exhaustive was a system tested. There are multiple ways to measure coverage, including input-space coverage and system state-space coverage. Algorithms from the motion planning domain such as Rapidly-exploring Random Trees (RRTs) are utilized in the literature for test generation with the target of increased coverage [58, 103, 26, 140, 49]. An approach for increased state-space coverage that utilizes

graph search for systematic simulation-based testing is proposed in [98]. RRT coverage measures are introduced and RRT-based algorithms for increasing state-space coverage during test generation are proposed in [58]. A test generation framework which is targeting increased initial-state coverage is presented in [95]. An approach that combines RRT-based search and sensitivity analysis for better selection of inputs, *i.e.*, selecting inputs that are guaranteed to increase the state-space coverage, is proposed in [40]. Tree search and model checking is combined for checking LTL safety properties in [139].

Chapter 3

OPTIMIZATION-GUIDED AUTOMATIC TEST GENERATION FOR AUTOMATED DRIVING SYSTEMS

3.1 Introduction

A common approach to testing ADS is using simulations and real-world driving. Because tests involving ADS would comprise test cases which may lead to collisions or near collisions, performing many tests with actual vehicles ending with collisions would not be economically efficient and practical. Hence, testing ADS using computer simulations is crucial. However, manually creating test scenarios with a large number of different environmental settings and road conditions would be difficult and highly time-consuming. Furthermore, in order to extract the limits of the systems under design, engineering teams would like to discover the behaviors on the boundary between safe and unsafe operations. Creating test cases manually for detecting the boundary conditions which barely cause collisions like fender-benders may be a challenging job. We believe that automatic test generation frameworks utilizing simulations are crucial for the future of autonomous vehicle testing. Such frameworks would produce a large number of tests generated in an intelligent way and help engineering teams to discover unforeseen scenarios that could lead to failure.

Discovering potential problems by testing in the simulation environments would be beneficial, but due to the inevitable differences in the simulation environments and the real-world, some test cases that do not fail in simulations may fail in the real-world or vice versa. So, instead of getting pass/fail results from the tests, using a

metric that indicates how close each test result is to a failure case, similar to phase and gain margins in control theory [78], would be more useful. With the availability of such a metric, engineering teams can run a large number of tests in faster simulation environments and after (automatic) analysis of the test results, they can repeat some scenarios in the real world using the methods described in [181] or in much more accurate simulators that require more computation power/execution time.

In this chapter, we address the problem of creating an automatic test generation framework for automated driving systems. The framework automatically searches for maneuvers for other vehicles in the driving environment that are used as the test cases. We describe two approaches with a focus on collisions. Because of the dynamics and possible physical limitations on the motion, an autonomous vehicle cannot be expected to be collision-free in every possible situation. For instance, it may not be possible to avoid a collision with a vehicle cutting in front at a very short distance or with a vehicle approaching very fast from a side. Our main focus is to find the conditions on the boundary between safe scenarios and collision scenarios. Our approach is based on running simulations, computing a cost using the simulation results that shows how close a system trajectory gets to an unsafe set of states and utilizing optimization methods to seek smaller costs by changing initial states and inputs for the system for the next test case. We describe another approach with a focus on driving comfort that relies on applying gradient-based optimization on a simplified system dynamics for seeking poor system performance on the more complex model.

3.2 Problem Definition

We denote the set of autonomous Vehicles Under Test (VUT), *also referred to as the set of Ego vehicles*, by $\mathcal{E} = \{\mathbf{e}_1, \dots, \mathbf{e}_p\}$, a set of dummy actors, *also referred to as Agents*, by $\mathcal{A} = \{\mathbf{a}_1, \dots, \mathbf{a}_q\}$, and the surroundings by $\mathcal{S} = \{\mathbf{s}_1, \dots, \mathbf{s}_r\}$. In more detail, the vehicles in the set \mathcal{E} are the vehicles which are to be tested either partially, *e.g.*, controller only, or as a system. The set \mathcal{A} of agents, *i.e.*, the mobile or immobile objects in the driving environment, may contain agent vehicles, pedestrians, obstacles, etc., optionally with controllers that can give them mobility. The agents physically interact with the VUT and they are typically used to trick the VUT in various test scenarios. The surroundings \mathcal{S} is a set of environmental settings like road network, the weather and road conditions, traffic rules and special zones. A driving scenario $\mathcal{T} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$ contains a set of Ego vehicles, agents and the surroundings. For simulation and testing purposes, we consider the overall driving scenario as a single system, and following the notation given in 2.1, we denote the model of the driving scenario as $\mathcal{M}_{\mathcal{T}} = (X_{\mathcal{T}}, \mathbf{U}_{\mathcal{T}}, P_{\mathcal{T}}, sim_{\mathcal{T}})$.

The initial state vector of the vehicle \mathbf{e}_i is denoted by $\mathbf{x}_{0,\mathbf{e}_i}$ for all $\mathbf{e}_i \in \mathcal{E}$. Similarly, the initial state vectors for the agent $\mathbf{a}_j \in \mathcal{A}$ and the surrounding $\mathbf{s}_k \in \mathcal{S}$ are denoted by $\mathbf{x}_{0,\mathbf{a}_j}$ and $\mathbf{x}_{0,\mathbf{s}_k}$, respectively. The domains of the corresponding initial states are denoted as $\mathcal{X}_{0,\mathbf{e}_i}$, $\mathcal{X}_{0,\mathbf{a}_j}$ and $\mathcal{X}_{0,\mathbf{s}_k}$, respectively. The initial state vector for the overall scenario is denoted by $\mathbf{x}_{0,\mathcal{T}} = (\mathbf{x}_{0,\mathbf{e}_1}, \dots, \mathbf{x}_{0,\mathbf{e}_p}, \mathbf{x}_{0,\mathbf{a}_1}, \dots, \mathbf{x}_{0,\mathbf{a}_q}, \mathbf{x}_{0,\mathbf{s}_1}, \dots, \mathbf{x}_{0,\mathbf{s}_r}) \in \mathcal{X}_{0,\mathcal{T}}$ where $\mathcal{X}_{0,\mathcal{T}} = \prod_{i=1}^{|\mathcal{E}|} \mathcal{X}_{0,\mathbf{e}_i} \times \prod_{j=1}^{|\mathcal{A}|} \mathcal{X}_{0,\mathbf{a}_j} \times \prod_{k=1}^{|\mathcal{S}|} \mathcal{X}_{0,\mathbf{s}_k}$, and the operators \prod and \times denote Cartesian product.

Abbas *et al.* [4] parameterize input signals $\mathbf{u}(t)$ over a bounded time domain R , by the parameter vectors $\lambda = [\lambda_1 \dots \lambda_m]^T \in \Lambda$, $\tau = [\tau_1 \dots \tau_m]^T \in R^m$, where Λ is a

compact set, $\tau_i < \tau_j$ for $i < j$, such that for all $t \in R$, $\mathbf{u}(t) = \mathfrak{U}(\lambda, \tau)(t) \in \mathbb{R}$. The function $\mathfrak{U}(\lambda, \tau)$ returns a function which is parameterized by λ and τ . For instance, \mathfrak{U} could represent the space of functions parameterized using splines [55]. We slightly modify the notation used in [4] to allow inputs to be functions of the states of the system, not only signals over time. Hence, we allow the set R in the above notation to be the bounded domain of any variable which can be a state of the system or time. We use the notation $\mathbf{u}_{\mathbf{e}_i, j} = (\lambda_{\mathbf{e}_i, j}, \tau_{\mathbf{e}_i, j}, \mathfrak{U}_{\mathbf{e}_i, j})$ for referring to the j^{th} input, and $\mathbf{u}_{\mathbf{e}_i}$ for referring to the vector of all inputs of the Ego vehicle $\mathbf{e}_i \in \mathcal{E}$. Similar notation applies to the inputs of agents and surroundings (such as the weather or lighting conditions changing over time) in the sets \mathcal{A} and \mathcal{S} , respectively. Concatenation of input vectors for all simulation entities is denoted by $\mathbf{u}_{\mathcal{T}}$, which is the overall input vector of the driving scenario. Parameter vector and bounded-time domains of $\mathbf{u}_{\mathcal{T}}$ are denoted by $\Lambda_{\mathcal{T}}$ and $R_{\mathcal{T}}$.

Ignoring further parameterization of the model $\mathcal{M}_{\mathcal{T}}$, *i.e.*, $P_{\mathcal{T}}$ and following from $\mathcal{U}_{\mathcal{T}} = \Lambda_{\mathcal{T}} \times R_{\mathcal{T}}$, the simulation function for the driving scenario is $sim_{\mathcal{T}} : \mathcal{X}_{\mathcal{T}} \times \Lambda_{\mathcal{T}} \times R_{\mathcal{T}} \rightarrow \mathcal{X}_{\mathcal{T}}$.

The problem we target is to compute:

$$(\mathbf{x}_{0, \mathcal{T}}^*, \mathbf{u}_{\mathcal{T}}^*) = \operatorname{argmin}_{\mathbf{x}_{0, \mathcal{T}} \in \mathcal{X}_{0, \mathcal{T}}, \lambda_{\mathcal{T}} \in \Lambda_{\mathcal{T}}, \tau_{\mathcal{T}} \in R_{\mathcal{T}}} \mathcal{R}(sim_{\mathcal{T}}(\mathbf{x}_{0, \mathcal{T}}, \mathbf{u}_{\mathcal{T}}(\lambda_{\mathcal{T}}, \tau_{\mathcal{T}}))) \quad (3.1)$$

where $\mathcal{R} : \mathbb{R}^{n \times k} \mapsto \mathbb{R}$ is defined as a cost function for a simulation of n time-steps length and an $\mathcal{M}_{\mathcal{T}}$ with a k -dimensional state space. The vector $\mathbf{u}_{\mathcal{T}}(\lambda_{\mathcal{T}}, \tau_{\mathcal{T}})$ contains all input functions, and it is obtained by applying the interpolation function $\mathfrak{U}(\lambda, \tau)$ for each input to the corresponding parameter vectors λ, τ from the selected vectors $\lambda_{\mathcal{T}}, \tau_{\mathcal{T}}$.

In other words, for a given simulation function, a set of vehicles under test (Ego vehicles), a set of dummy actors (Agents), surroundings information and constraints

on state space and input space, we are seeking the particular inputs and initial states for the simulation that would minimize a cost function. To discover system operating points that are of interest for testing purposes, one must carefully design a cost function that will (quickly) guide the search toward the critical points.

In the following, we propose two alternative cost functions and two falsification approaches for discovering the boundaries between safe and unsafe behaviors. We also propose a functional gradient descent-based approach to find specific conditions, such as minimum performance, of systems utilizing a simplification of system dynamics.

3.3 Falsification-based Approach to Test Generation

A simplified overview of our falsification-based approach is illustrated in Fig. 3. The main components of the vehicular systems testing framework that we propose are the optimization engine of S-TALIRO, a simulation engine, a cost evaluation function, and a simulation configuration.

First, a sample space is created from the user-defined input and/or initial state configuration. Then, an initial states vector and an input functions vector are sampled from the generated sample space. The simulation of the vehicular systems is executed for a predefined amount of time with the selected initial states and inputs. As illustrated in Fig. 3, the simulation engine returns the simulation output trajectory which consists of states and/or outputs of the simulated system(s) for each time step of the simulation. The output trajectory obtained from the simulation is supplied to the cost evaluation function that returns a real-valued cost as an evaluation of how close the simulation results are to an unsafe set of states. The obtained cost is used by the optimization engine and the stochastic sampler in S-TALIRO for generation of

the inputs and initial states for the next simulation with an attempt to obtain smaller cost values. This cycle of input generation, simulation, and cost evaluation continues until either a negative cost value is achieved or the maximum number of simulations is reached which we use as the termination conditions for the optimization problem given in Eq. (3.1).

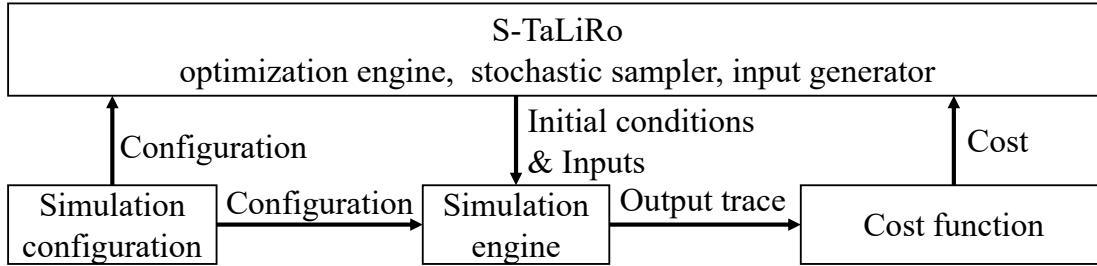


Figure 3. An Overview of the Framework Architecture.

3.3.1 S-TALiRO

S-TALiRO [18] is a MATLAB[®] [118] toolbox for systematic testing of hybrid systems, *i.e.*, the systems that exhibit continuous and discrete dynamics. It uses a robustness metric that represents how far a system trajectory is from falsifying formal system requirements. In particular, negative robustness values mean a requirement is falsified, *i.e.*, conditions have been found under which the system does not satisfy a requirement. S-TALiRO uses one of the various global optimization methods for minimizing the robustness function [18] and, thus, for seeking a falsifying system trajectory. We utilize S-TALiRO for solving the problem defined in Section 3.2, basically for intelligently sampling initial states and input functions that will be applied to the simulations. The robustness evaluation function, which is used in the

optimization engine of S-TALiRO, can either be supplied by the user or used from robustness computation implementations with respect to temporal logic specifications available in S-TALiRO [64]. In our approach, instead of using the temporal logic robustness, we provide a custom cost function to S-TALiRO.

3.3.2 Simulation Configuration

Simulation configuration is used to parameterize a wide range of classes of systems and scenarios. The automatic test generation proceeds by sampling points from this parameterized space as explained above. A test scenario can be described in a simulation configuration with a focus on a function with different types of systems and environments.

Referring to the definitions given in Section 3.2, a simulation configuration is basically a structure describing the driving scenario $\mathcal{T} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$ with the domains for initial states and inputs, *i.e.*, $\mathcal{X}_{0,\mathcal{T}}$ and $\mathcal{U}_{\mathcal{T}} = \Lambda_{\mathcal{T}} \times R_{\mathcal{T}}$.

A typical configuration contains environmental parameters, the number of vehicles in the simulation and parameters for each vehicle. Some examples of the environmental parameters could be wind, road incline, lane width, number of lanes, inputs, and states of the environment. Vehicle-related parameters can be mass, tire-friction, ranges of initial states and inputs, function handles that describe dynamics of these vehicles or controllers for the vehicles. These are only some examples and the actual parameters must be completely defined by the user in accordance with the user-supplied simulation engine and the cost evaluation function.

3.3.3 Simulation Engine

The simulation engine can be a MATLAB function, a SIMULINK [118] model or any external simulator like WEBOTS [124] or CARSIM [121] that can be wrapped by a MATLAB function. The simulation engine must be able to accept inputs described in the configuration, and it must return the computed states and/or outputs for each time step of the simulation. Because the simulation configuration is available to the simulation engine, the user can freely parameterize the simulation engine in the desired level of detail in accordance with the testing purposes.

Here, we recall the definition of a simulation function given in Section 3.2 as $sim_{\mathcal{T}} : \mathcal{X}_{\mathcal{T}} \times \Lambda_{\mathcal{T}} \times R_{\mathcal{T}} \rightarrow \mathcal{X}_{\mathcal{T}}$. In summary, the simulation engine first initializes the models and/or controller functions of the VUT in the set \mathcal{E} , surroundings \mathcal{S} and the agents \mathcal{A} with given initial states. Then, it executes these models/controllers with respect to the given inputs while considering the interactions of the entities in the above sets with each other and generates an output trace \mathbf{y} .

3.3.4 Cost Function to Detect Boundary-case Collisions

In a simulation setup, as the number of variable parameters increases, the space created by these parameters can be very large. Testing every combination over such a large space and finding falsifying behaviors is infeasible in most cases. Because S-TALIRO is based on optimization, the obtained cost values from different simulations are expected to guide the search toward failure as opposed to a completely random selection of test cases. It should be noted that the choice of the cost function plays an important role for better guidance.

A cost function must return smaller values as we get closer to the most interesting failing system behavior that we seek. If a negative cost value is obtained, S-TALiRO immediately stops and returns the related trajectory as a falsifying trajectory. Otherwise, the search for a smaller cost value over trajectories continues until a given maximum test count is reached.

In this chapter, we mainly focus on testing autonomous vehicles against collisions in an environment where some vehicles may follow trajectories that can lead to dangerous situations. Furthermore, we seek the conditions, *i.e.*, initial states and input signals, that lead to near collisions. Hence, we design a cost function so that the boundaries between safe and unsafe behavior can be reached by minimizing the cost. Here, we will propose a cost function that can be applicable to a wide range of setups for testing autonomous vehicles with a purpose of detecting collisions and/or the situations where vehicles exit a predefined drivable area.

For a collision instance between two vehicles with velocities \vec{v}_1 and \vec{v}_2 at the time of the collision, we compute the severity of the collision as $\|\vec{v}_1 - \vec{v}_2\|$, where $\|\cdot\|$ is the Euclidean norm. When a collision involving a VUT is detected in a simulation output trace \mathbf{y} , we compute $v_{coll,\mathbf{y}}$ as the collision severity at the moment of the first collision experienced in \mathbf{y} .

If there is no collision involving a VUT in a simulation output trajectory, we use a safety measure called Time-To-Collision (TTC) [81]. The TTC is the time required for two vehicles to collide when they are on a collision path. Being on a collision path for two vehicles means that they will collide if they continue their current motion. In particular, the TTC for two vehicles that are not on a collision path is infinite. We use the looming points approach described by Ward *et al.* [178] for collision path

and TTC computations. The minimum TTC experienced between any two vehicles during a simulation trace \mathbf{y} is denoted by $ttc_{min,\mathbf{y}}$.

Note that we can use collision severity and TTC metrics for testing against a vehicle exiting the drivable area. Considering the boundaries of drivable areas as stationary objects, *e.g.*, a wall, TTC or collision severity with these objects can be computed in a similar manner by taking the velocity of the objects as a zero vector.

Because we search for the boundary between safe and unsafe operations, we can consider a collision where vehicles barely touch each other with zero difference in velocities as the boundary case that we seek. Hence, a collision with a high relative velocity between the vehicles must have a larger cost compared to a collision with low relative velocity. In addition, a simulation trace with no collision must have a larger cost compared to a simulation result involving a collision. Our proposed cost function $\mathcal{R}(\mathbf{y})$ for a simulation output trace \mathbf{y} is given below:

$$\mathcal{R}(\mathbf{y}) = \begin{cases} v_{coll,\mathbf{y}} - v_{\epsilon} & , \text{collision detected in } \mathbf{y} \\ ttc_{min,\mathbf{y}} + v_{coll,max} & , \text{otherwise.} \end{cases} \quad (3.2)$$

where $v_{coll,max}$ is the maximum possible relative collision velocity and v_{ϵ} is a user-defined nonnegative real-valued number representing the minimum collision severity of concern. Whenever the framework achieves a collision with a severity smaller than v_{ϵ} , the cost value will be negative and the search will be terminated. In particular, setting v_{ϵ} to zero means that we are seeking the collisions with the vehicles barely touching each other. However, in this case, the search will continue until the maximum number of simulations is reached and the test case leading to the minimum cost value will be returned.

We assume that we know maximum possible velocity for all the objects in the

simulation which is denoted by v_{max} . The maximum collision velocity in a simulation can be experienced between two vehicles traveling at the maximum speed in opposite directions. Hence, $v_{coll,max} = 2v_{max}$.

3.3.5 Case Study

As a case study, we use the simulation engine with the simulation configuration described below and the cost function in (3.2). S-TALIRO is configured to use the simulated annealing method [3].

3.3.5.1 Simulation Configuration for the Case Study

Our case study consists of two VUT in the set \mathcal{E} and an agent vehicle in the set \mathcal{A} on a straight two-lane road that is described in \mathcal{S} . The inputs to the simulation are the target speed functions for the VUT and target speed and lateral position functions for the agent vehicles. Target speed functions are defined over time, and the target lateral position function is defined over the longitudinal position state of the agent vehicle.

One of the VUT is following the other on a straight target trajectory on the right lane of a two-lane road. The agent vehicle has a trajectory which starts on the left lane of the road and changes to the right lane after a distance chosen by the testing algorithm. The target position of the agent vehicle inside a lane is varying over the course of the simulation and the lane change position is also sampled from a predefined longitudinal position range.

The shape and dimensions of the vehicles are described in the configuration as the

critical points, *e.g.*, corners, of the vehicles. These points are used to detect collisions and also used in the looming points method [178] to check collision paths and to compute the TTC values.

3.3.5.2 Simulation Engine for the Case Study

For the simulation of the VUT, we use a vehicle dynamics model from the literature [187], [128]. To accurately represent the dynamics of the VUT, we use relatively complex dynamical models that are costly to compute during simulation. However, it is not computationally practical to use dynamical models of similar complexity for the agent vehicles that are merely meant to generate reasonable test trajectories to challenge the VUT. Furthermore, the actual controllers on the agent vehicles will be out of the control of the tester, and so all that is necessary for the agent vehicles is to capture the salient features of realistic vehicles in the simulation. Consequently, for the agent vehicles, we use simpler kinematic models and controllers. The kinematic model we use for the agent vehicle in our case study is described by Walsh [177]. We have implemented our simulation engine for the case study as a MATLAB function.

3.3.5.3 Sensor Setup

We describe the sensors on the vehicles by their orientation, range, maximum sensing angle and position with respect to the center of mass of the vehicle. In our case study, we use a sensor setup for side collision avoidance. The vehicles under test have one distance sensor in front with a range of 40 m and 10° sensing angle and one distance sensor on the left side with a range of 3 m and 45° sensing angle. The

sensor placement and orientation are illustrated in Fig. 4. The rectangle in the figure represents the top view of the vehicle where the tip of the arrow on the rectangle is towards the front of the vehicle. This sensor configuration is used to test the framework’s ability to detect possible collisions resulting from the corresponding blind spots.

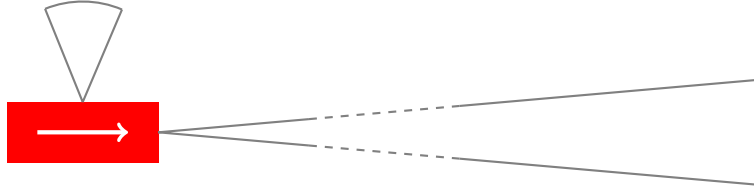


Figure 4. Sensor Placement.

3.3.5.4 Vehicle Controller with Collision Avoidance

We have implemented a controller with basic forward and side collision avoidance capabilities by merging two controllers from the literature. For steering control, we used the Stanford’s Racing Team’s approach [86] for the DARPA Grand Challenge 2005. For the longitudinal control, we used the model predictive convoy controller from Liu and Ozguner [115]. A reactive planner generates a target path based on the input target speed and the forward and side distance sensor data. The generated target path and the input target speed are supplied to the controller, which generates force and steering inputs. We use this controller and the simulation setup only for demonstrating our framework, and we do not claim any performance or accuracy guarantees for the controller or the simulator.

We describe the target speed for the VUT as an input signal chosen by the testing algorithm in a predetermined range $[5, 15]$ m/s over the simulation time. The target

lateral position for the VUT is the midpoint of the right lane. However, because the VUT controller has collision avoidance capabilities, the target lateral positions for the vehicles are updated during run-time based on the sensor data.

3.3.5.5 Motion Controller for the Agent Vehicle

The target trajectory for the agent vehicle is described by two input functions for S-TALIRO. One input function is the target speed for the vehicle. We define the target speed as a signal with a predefined number of control points, *i.e.*, the parameter τ described in Section 3.2, equally distributed over the simulation time. We set the lower and upper limits, *i.e.*, the domain for the parameter λ described in Section 3.2, for the target speed at each control point. Piecewise cubic Hermite interpolating polynomial (pchip) interpolation [68] function that is available in MATLAB® [118] is used for interpolation between the control points, *i.e.*, the \mathfrak{U} described in Section 3.2. Thus, the target speed for the agent vehicle is a signal interpolated between the values chosen by the test algorithm from a given range.

The other input function for the agent vehicle describes its target lateral position. We describe this input with respect to the vehicle's longitudinal position state instead of the simulation time. We use 4 control points for this function where the first and the last control points are located at positions 0 m and 300 m in the longitudinal axis. The locations for second and third control points are chosen by the test algorithm between these positions with a constraint for the distance between two consecutive control points to be at least 5 m. The value ranges for the control points are the limits of the left lane for the first two control points and the limits of the right lane for the remaining control points. This describes a trajectory that starts at the left lane and

then changes to the right lane. The lateral position inside the lanes varies between the selected values by the test algorithm.

For the agent vehicle, as opposed to the VUT, we have implemented a PID controller for tracking the target speed instead of the costly model predictive controller. As discussed in subsection 3.3.5.2, the controller for the agent vehicle is only used for roughly following the trajectory proposed by the tester that will be used to challenge the VUT.

3.3.5.6 Experimental Results

As stated in subsection 3.3.5.3, we intentionally created a blind spot in the sensor setup for the VUT. During our experiments, the framework successfully detected failure cases caused by this weakness. Figure 5 illustrates a near collision where the VUT (at the bottom-right of the figure) avoids a side collision in the first place and then collides with the agent vehicle when returning back to its lane. In this case, the VUT first avoids the side collision by changing its lateral position and slowing down. However, after avoiding the side collision it loses track of the agent vehicle because of the blind spot. Hence, the VUT does not continue slowing down although it should have. Furthermore, it starts making the maneuver to return to its lane. As a consequence, it barely touches the agent vehicle at its rear-right corner. The final parts of the vehicle trajectories are displayed as traces behind the vehicles in Fig. 5. The second VUT, *i.e.*, the one on bottom-left of the figure, is following the VUT that had a collision. This VUT is far from the collision scene, and it is not affected by the collision.

There are additional collisions detected by the framework, and all of them are

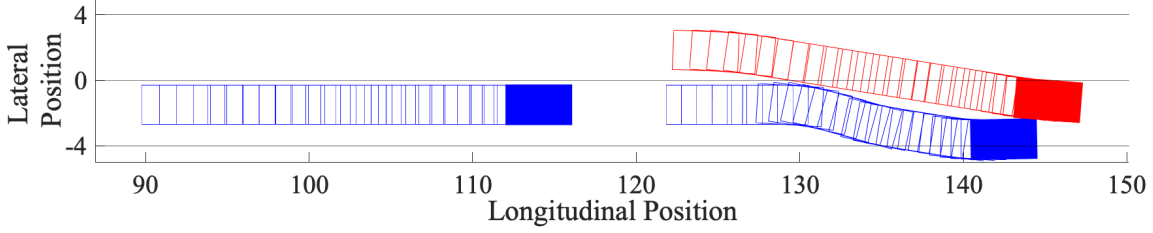


Figure 5. History of the Vehicles Before a Collision Instance. The bottom-right vehicle makes a failed attempt at evading a collision with the top-right vehicle due to a sensor blindspot.

returned to the user for further analysis; however, this was the collision with the minimum cost value returned by S-TALIRO, which makes it an interesting case at a boundary between safe and unsafe operation. The sampled input parameters and the generated input function as the target lateral position of the agent vehicle leading to the collision is given in Fig. 6. The τ parameters are defined over the longitudinal position state of the agent vehicle, and the λ parameters are the target lateral positions for the corresponding τ parameters. For this case study, the (τ, λ) samples that led to the collision of concern are $(0.0, 1.95)$, $(121.2, 1.83)$, $(148.2, -1.15)$, $(300.0, -1.11)$.

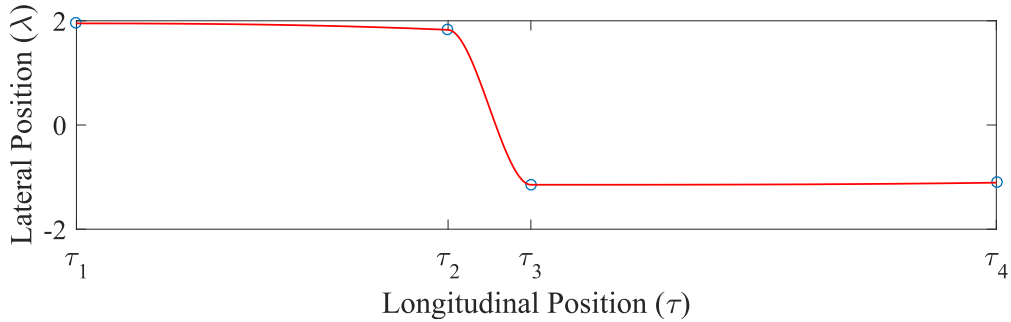


Figure 6. The Input for Target Lateral Position of the Agent Vehicle.

We have run our experiments on a Windows[®] PC with an Intel[®] Core[™] i7-4790 CPU and 16GB RAM. A simulation of 32s of the described test setup takes 18s physical time on our setup. The above failure condition was detected in 100

simulations. Even though the convergence to the global minimum cost is guaranteed with simulated annealing [3], our stochastic approach provides no guarantee on the number of simulations required to achieve the global minimum. In general, the execution time of one simulation in the proposed framework depends on the complexity of the vehicle ODEs and controllers. The overall worst-case execution time for the framework grows linearly with the maximum number of simulations chosen by the user.

3.4 Rapidly-exploring Random Trees for Testing Automated Driving Systems

As an alternative to the falsification-based approach that is described in Section 3.3, we propose an approach that utilizes Rapidly-exploring Random Trees (RRTs). In this approach, we consider the test generation problem as a path planning problem for agent actors with the aim of creating interesting collisions with Vehicles Under Test (VUT), which are also called as *Ego* vehicles. As it is discussed in Section 3.1, we focus on finding test cases that lead to behaviors at the proximity of the boundary between collisions and near-collisions.

Rapidly-exploring Random Trees, first introduced by LaValle in [110], provide an efficient method for exploring and covering high-dimensional spaces. Although RRTs are mostly used for path planning, they find applications in various domains including test case generation [58, 103, 26, 40, 140, 49] as we have discussed in Section 2.3.

Since their first introduction, many variants of RRTs have been proposed. In [93], a method called Transition-based RRT (T-RRT) was introduced. Transition-based RRT method extends the classical RRT by incorporating additional cost criteria to the explored paths rather than only aiming to reach a target configuration. T-RRT

borrows the notion of transitions tests from stochastic optimization approaches. Hence, it can be considered as a method that is merging RRTs with stochastic optimization. Furthermore, T-RRT controls exploration versus refinement using a method called *minimal expansion control* which helps to promote the expansion of a tree to the unexplored areas of the search space. Efficiency of T-RRT on continuous cost spaces is studied in [93].

3.4.1 Overview of the Approach

In our approach, we utilize T-RRT with a custom cost function that we propose to find boundary-case collisions. We implement our version of *minimal expansion control* using the notion of *sparseness* from evolutionary algorithms that perform novelty search [159, 112].

With this approach, we address the following limitations of our falsification-based approach [173] that is described in Section 3.3:

1. In [173], we use a limited number of control points over the longitudinal position axis as the specific points where the lateral axis of the vehicle trajectories are sampled. As the number of control points decreases, possible variations in shapes of the trajectories are limited. On the other hand, increasing the number of control points also increases the dimension of the search space, which makes the problem more challenging.
2. In [173], the duration of the simulations is fixed. With the RRT-based approach, although there is an inherent limit on the maximum simulation duration that is dictated by the maximum number of RRT nodes, there is flexibility in the

duration of the simulations. So, non-promising simulations are stopped earlier while more promising ones can be executed for longer times.

3. With the RRT-based approach, we can minimize the need for hand-designing a test scenario in detail and allow more freedom in the exploration of the space compared to the falsification-based approach.
4. RRT-based approach is promising to avoid local minima that can be challenging to the falsification-based approach proposed in [173].

The flowchart of our RRT-based approach is shown in Fig. 7. In the rest of this section, we will describe the key components of our approach.

3.4.1.1 Initializing the Search

We reuse the definitions and notation given in Section 3.2. A scenario, \mathcal{T} , is described by the sets of Ego vehicles, agent actors and environment as $\mathcal{T} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$. The model of the driving scenario is given as $\mathcal{M}_{\mathcal{T}} = (X_{\mathcal{T}}, \mathbf{U}_{\mathcal{T}}, P_{\mathcal{T}}, sim_{\mathcal{T}})$. In this approach, the set of inputs, $\mathbf{U}_{\mathcal{T}}$, can simply be a set of target paths for the agent actors, as well as inputs to the other entities in the simulation environment such as models of road and weather conditions. After general outlines of the driving scenario are described, the sampling space for the initial states, *i.e.*, $\mathcal{X}_{0,\mathcal{T}}$, is used to sample initial configurations of the simulation entities, including Ego vehicles and agent actors.

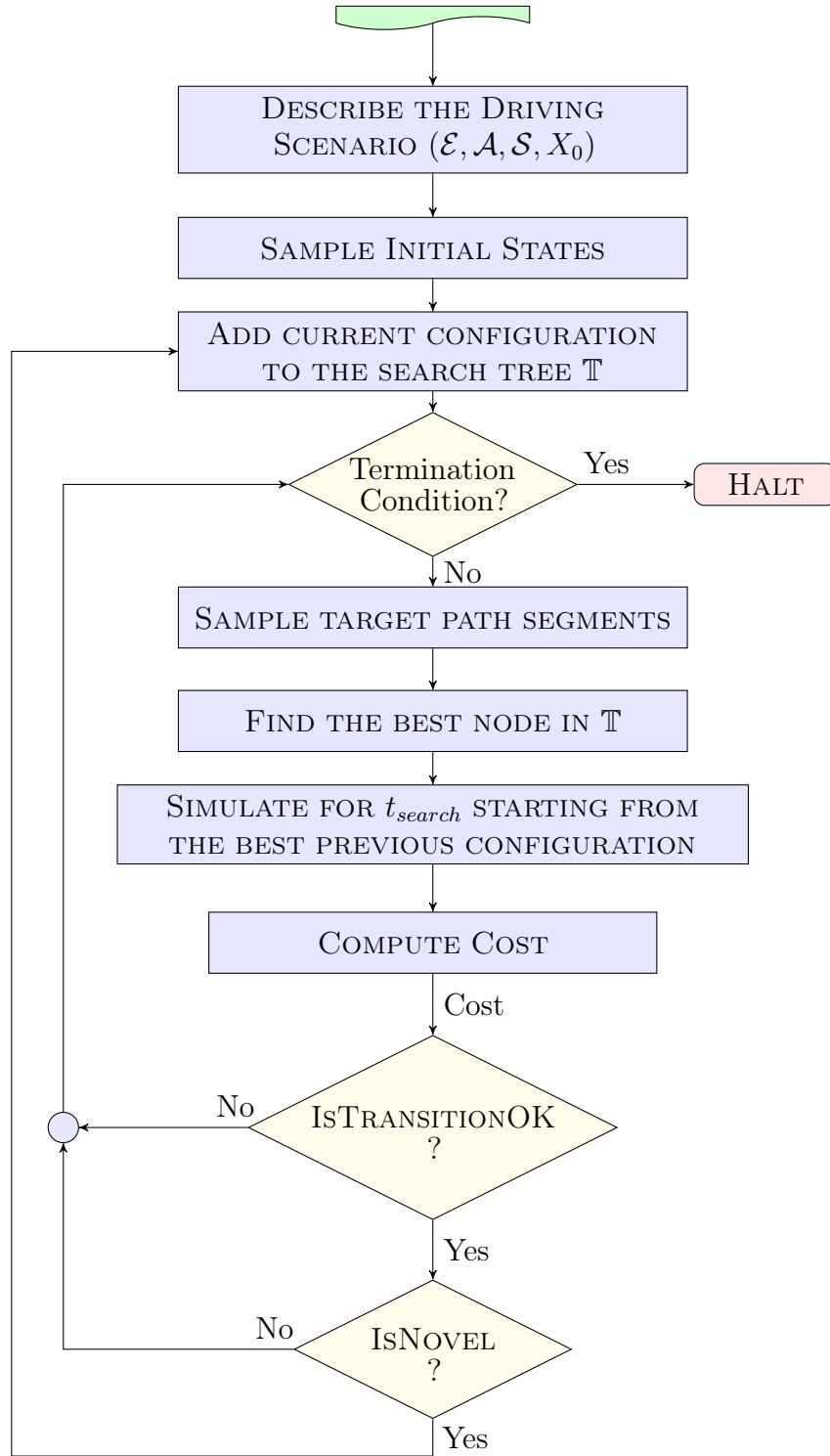


Figure 7. Flowchart illustrating the RRT-based approach.

3.4.1.2 Information Stored on RRT Tree Nodes

A tree grows while seeking to discover interesting behaviors. While growing the tree, instead of executing simulation traces starting from the initial configuration, only a partial simulation is executed starting from an existing node in the tree. For that purpose, the state of the system, the state of the controllers, and the simulation time are stored on the tree nodes. Table 2 provides more details on the data stored on the tree nodes after each partial simulation. The history-related fields will be empty for the root nodes of trees.

Data	Description
State	The state of the system at the end of the corresponding part of the simulation is stored. It serves as the initial state for a new partial simulation that is starting from the configuration represented in the current node.
State History	(optional) The state history over the corresponding part of the simulation is used to (i) reproduce agent behaviors after the search is over (ii) to simulate any sensor delays for the next simulation step.
Input History	(optional) The history of inputs to the system is used as the past input data for the next simulation step, which may be useful for the systems that need to remember past inputs. This data is also valuable for analysis purposes when the search is over.
Controller State	The final state of controllers, if available, are stored so that the next step of the simulation can be started from the current node without needing to run the whole simulation from the starting node to the current node.
Simulation Time	The time at the end of the corresponding part of the simulation is stored.

Table 2. Data stored on the RRT nodes.

3.4.1.3 Sampling a Target Path Segment

A sample target path segment is simply a set of waypoints which is used as an immediate target for the corresponding vehicle. A waypoint is denoted as $\mathbf{w} = (x_{\mathbf{w}}, y_{\mathbf{w}}, \theta_{\mathbf{w}}, v_{\mathbf{w}}) \in \mathcal{P}_{\mathbf{w}}$ where $x_{\mathbf{w}}, y_{\mathbf{w}}, \theta_{\mathbf{w}}, v_{\mathbf{w}}$ are the x -coordinate, y -coordinate, target driving direction and the target speed at the waypoint. The sampling space for the waypoints is defined by a corresponding parameter space $\mathcal{P}_{\mathbf{w}} = \mathcal{P}_{\mathbf{w},x} \times \mathcal{P}_{\mathbf{w},y} \times \mathcal{P}_{\mathbf{w},\theta} \times \mathcal{P}_{\mathbf{w},v}$ that describe the limits on the $x - y$ coordinates, driving direction, and target speed where $\mathcal{P}_{\mathbf{w}} \subseteq \mathcal{P}_{\mathcal{T}}$. An example waypoint sampled on a straight road is shown in Fig. 8. A coordinate transformation can be applied for sampling from curved roads. Although the example waypoint in Fig. 8 is sampled from a road, the sample space of the waypoint doesn't have to be the same as the area of a road in the simulation. It may be defined to go beyond the road limits, it may be limited to only a part of a road, or it may be completely irrelevant to a road in the simulation environment.

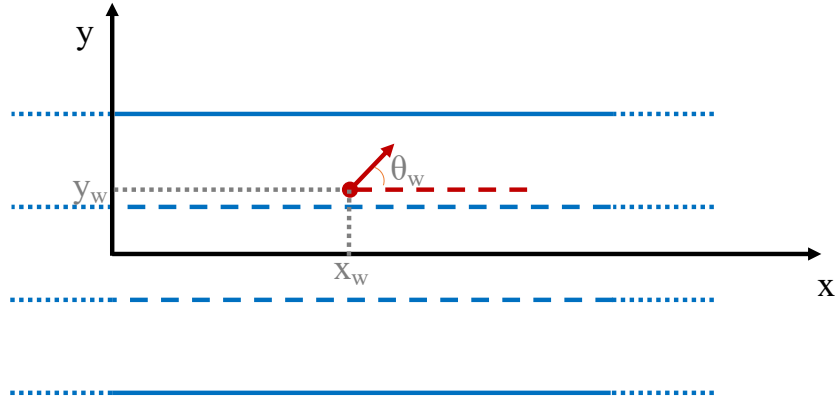


Figure 8. Sampling a waypoint.

Once a waypoint is sampled, the next step in sampling a target path segment is to add an endpoint at a predefined distance d_{leg} from the waypoint, along the direction of the waypoint. Figure 9 shows a target path segment formed using this

approach. The sampled target path segment for this example can be denoted by $\mathbf{p} = ((x_w, y_w, \theta_w, v_w), (x_{w2}, y_{w2}, \theta_w, v_w))$.

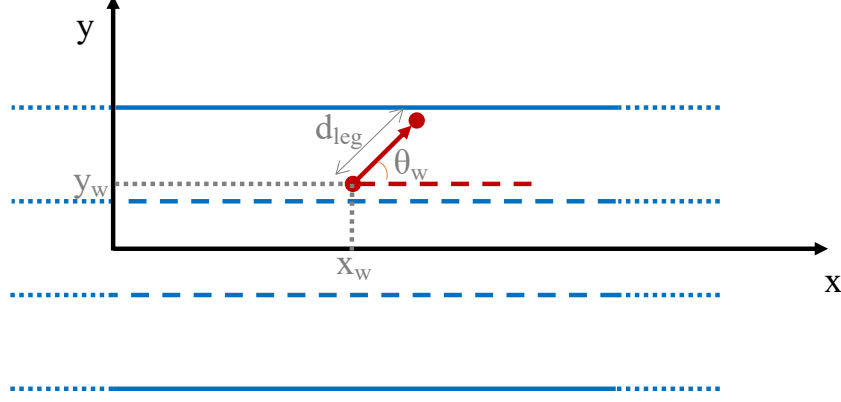


Figure 9. Sampling a target path segment.

If the endpoint of a target path segment is outside the sampling space of the waypoint, we simply break the segment at the boundary of the sampling space and add a second leg along the boundary in the direction closest to the waypoint direction. Figure 10 shows an example target path section for a longer $d_{leg} = d_{leg1} + d_{leg2}$ compared to the one in Fig. 9. The sampled target path segment for this example can be denoted by $\mathbf{p} = ((x_w, y_w, \theta_w, v_w), (x_{w2}, y_{w2}, \theta_w, v_w), (x_{w3}, y_{w3}, \theta_w, v_w))$.

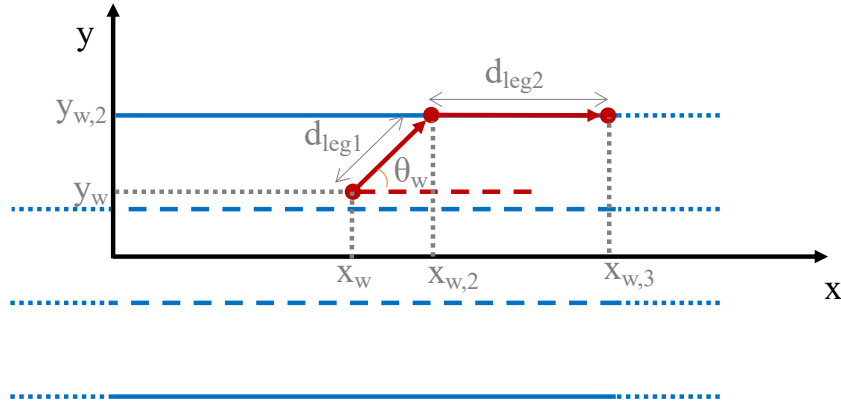


Figure 10. Sampling a target path segment with space constraints.

After the shape of the target segment is decided, we also sample a target speed, v_w , for the target path segment.

This approach is applied to each agent vehicle for sampling their target path segments. Note that the framework we propose allows using a different algorithm for selecting a target path and/or any other type of input to the agent vehicles.

3.4.1.4 Selecting the Best Node from the Search Tree

Once target path segments for the agent vehicles are sampled, we pick one node from the existing tree, as the initial configuration for the simulation that will be executed with the sampled target path segments. There is no single correct approach to decide which node of the tree would be the best choice.

In our study, the notion of selecting the optimal node is adopted from the RRT* method [99]. In [99], when adding a new node to the tree, existing nodes in a neighborhood of the new node are all checked and the one which minimizes the cost is selected as the previous node. Our approach has similarities to the RRT* one. We compute the sum of distances from each vehicle to the starting point of their target path segment with the constraint that the configurations of all vehicles are behind the start position of the initial target waypoint with respect to the driving direction of that waypoint. Then, we execute partial simulations from the best n candidate previous nodes ($n = 5$ for our case studies) and pick the one which gives the minimum cost. We believe that this approach is promising to create relatively natural-looking vehicle trajectories while still allowing enough randomness in the maneuvers.

In [99], after adding the new node to the tree, there is a rewiring step which modifies the connections to the other existing nodes in the neighborhood of the

new node. The rewiring step checks whether reaching to an existing node in that neighborhood from the newly added node would result in a reduction in the cost without violating any constraints such as obstacles. If so, it replaces the existing edge incoming to that node with an edge originating from the newly added node. Application of the rewiring step is straightforward for path planning problems on Euclidean spaces where the tree nodes represent planned waypoints on a path instead of the simulated vehicle configurations. In our approach, the tree nodes are the resulting configurations reached by the simulated vehicles with a controller and an input target path. Hence, the rewiring step requires execution of partial simulations starting from the newly added node to the other nodes in the neighborhood of the newly added node. Since the resulting configuration will most likely be different at the end of such a partial simulation, the configurations on the target nodes will change. This creates the necessity to execute simulations from the updated nodes to all of the remaining tree nodes that can be reached from the modified node. Hence, the rewiring step can be computationally costly in our approach, and so, we do not apply the rewiring step and leave it as a future work for which the applicability should be analyzed.

We would like to emphasize that the function used for selecting the best previous node is user-configurable in our framework and depending on how much randomness is plausible in the generated driving paths, a different algorithm, *e.g.*, simply selecting the closest node, can be utilized.

3.4.1.5 Simulating the System

After obtaining a set of target path segments for agent vehicles and deciding the initial configuration for the simulation, we create the simulation scene in the simulation environment using the data stored in the selected node of the search tree. That is, we set the initial states of the simulation entities and initialize the Ego vehicle controllers with the previous inputs and the saved controller states. We also pass the sampled target path segments to the agent vehicles as inputs. Finally, we simulate the system for t_{search} time and collect state and input histories at each time step of the simulation. For our setup, we use MATLAB simulations, however, this is not mandatory and another simulator can be used. Note that if the simulator and the Ego vehicle controllers allow saving the state and continuing simulation from a saved state, *which is the case in our setup*, the time spent in the simulations can be radically reduced because it would be enough to simulate only the new part of the simulation. Otherwise, the simulation should always start from the root node of the tree and run until the current target time.

3.4.1.6 Cost Function

After a simulation is executed, a cost function is used to compute how close the simulation trace is to an interesting behavior. The approach we describe here can be utilized to discover other types of interesting/failing behavior; however, our target in this work is to explore the behaviors that are on the boundary between safe and unsafe operation. Hence, an interesting behavior for our purposes would be (i) a collision between an Ego vehicle and an agent that could have been avoided with

a minor change in the control applied or agent trajectories, (ii) an almost-collision (near-collision) which could have ended with a collision with a minor change in the control applied or agent trajectories.

The properties of a good cost function that would guide the search toward an interesting behavior for our purposes can be listed as follows:

- Among two similar collisions between an Ego vehicle and an agent vehicle, the one which has the smaller magnitude in the relative speed between the vehicles should have a smaller cost, as a smaller change in the speed of the Ego vehicle would be enough to avoid the collision.
- Among two similar collisions, the one which has the smaller impact area, *i.e.*, the area of the collision surface, should have a smaller cost, as a smaller change in the steering maneuver of the Ego vehicle would be enough to avoid the collision.
- For vehicle paths without a collision, a smaller time-to-collision at any point of the path, and a smaller corresponding collision speed and a smaller area for that collision-to-be should lead to a smaller cost.

The cost function we have described in Eq. (3.2) satisfies some of the required properties listed above. However, it always prioritizes collisions over near-collisions. This may result in returning a high-speed collision case instead of a case where a collision was avoided by centimeters. Although such a near-collision case can be extracted from the framework outputs, (i) this is not ideal as it requires further analysis of outputs, (ii) the global optimizer is less likely to explore the vicinity of a case with a near-collision to find a boundary-case collision as it is guided to a different point with a high-speed collision. Another problem in Eq. (3.2) is its discontinuous nature.

We propose the following cost function to address the weaknesses of Eq. (3.2):

$$\mathcal{R}(\mathbf{y}) = (1 + s_{coll,\mathbf{y}})(v_{coll,\mathbf{y}}^2 + ttc_{min,\mathbf{y}}^2) \quad (3.3)$$

where $s_{coll,\mathbf{y}} \in [0, 1]$ is the ratio of the collision surface to the overall surface on the collision side of the vehicle, $v_{coll,\mathbf{y}}$ is the relative speed of the vehicles at the moment of collision, and $ttc_{min,\mathbf{y}}$ is the minimum time-to-collision encountered during the simulation output trace \mathbf{y} . For the simulations with a collision, $ttc_{min,\mathbf{y}}$ is 0. For the simulations without a collision, $s_{coll,\mathbf{y}}$ and $v_{coll,\mathbf{y}}$ are computed at the instance of smallest time-to-collision with the assumption that the vehicles continue their motion without changing their speeds and orientations. When the simulation output trace \mathbf{y} contains collision(s) with Ego vehicle, we only consider the first collision of an Ego vehicle with any object for computing Eq. (3.3). Figure 11 shows the function with respect to the minimum time-to-collision and collision speed variables for a fixed collision surface. The effect of the collision surface to the cost is linear.

3.4.1.7 Transition Check Function

The function we use for accepting a new configuration based on the cost is similar to the one proposed in [93]. Algorithm 1 repeats it for convenience. We denote the newly simulated configuration as the *candidate* and the initial configuration selected from the search tree as *previous* configuration. In the algorithm, K is a constant parameter normalizing the change in the cost, T is the temperature parameter that is governing the likelihood of acceptance. The temperature parameter T is adaptively tuned by multiplying with or dividing by α with respect to the ratio of rejections to acceptances.

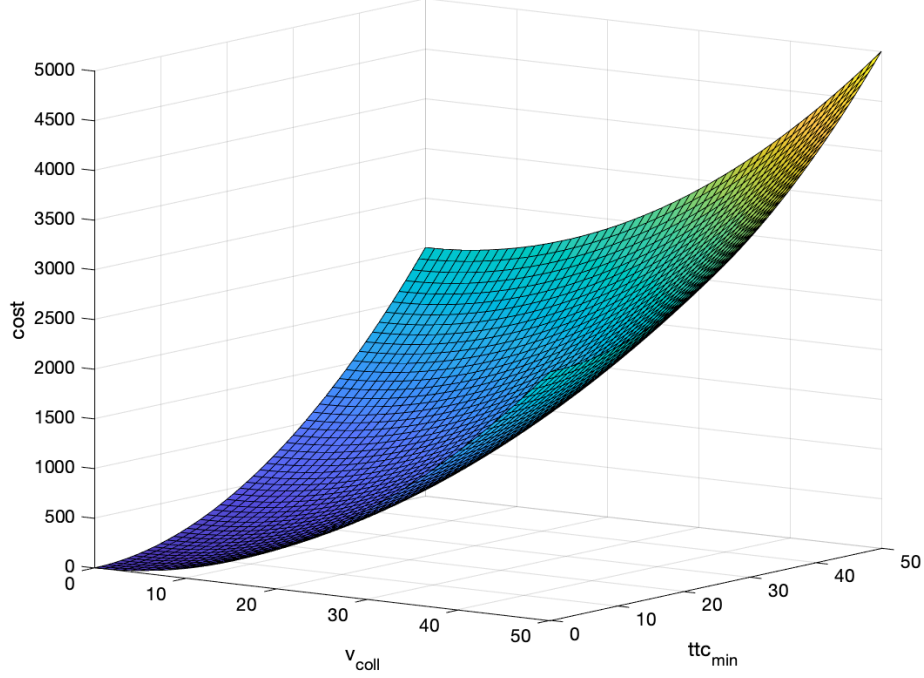


Figure 11. Cost function to guide the search toward the boundary between collisions and near collisions.

3.4.1.8 Novelty Function

To get better coverage of the state space and to avoid local minima, we reward novelty in our search. For an Ego vehicle $\mathbf{e}_i \in \mathcal{E}$ and an agent $\mathbf{a}_j \in \mathcal{A}$, we define $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$ as the vector of relative states and change in relative states at discrete simulation time $k \in [0, n]$, *i.e.*, $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k] = (\mathbf{x}_{\mathbf{e}_i}[k] - \mathbf{x}_{\mathbf{a}_j}[k], (\mathbf{x}_{\mathbf{e}_i}[k] - \mathbf{x}_{\mathbf{a}_j}[k]) - (\mathbf{x}_{\mathbf{e}_i}[k-1] - \mathbf{x}_{\mathbf{a}_j}[k-1]))$.

We compute the novelty of $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$ as follows:

$$\mathcal{N} = \sum_{l=0}^m \text{dist}(\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k], \mu_l) \quad (3.4)$$

where $\mu_l \in X_{rel, k-1}$ is the l^{th} nearest neighbor of $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$ in the set $X_{rel, k-1}$ which contains $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}$ vectors for all ego-agent pairs for all times before k . The function dist computes the Mahalanobis distance between $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$ and the elements of its m -nearest neighbors set. We choose to use the Mahalanobis distance because of its ability to

Algorithm 1 Algorithm used to check acceptance of a new configuration based on the change in the cost.

```

1: function ISTRANSITIONOK( $c_{prev}, c_{cand}$ )  $\triangleright c_{prev}$  and  $c_{cand}$  are the costs
   associated with the previous and candidate configurations, respectively.  $\alpha$  and  $K$ 
   are constant parameters, and  $T$  is a persistent parameter.
2:   if  $c_{cand} < c_{prev}$  then
3:     return True
4:   end if
5:   if  $rand(0, 1) < e^{(c_{prev} - c_{cand}) / (K * T)}$  then
6:      $T = T / \alpha$ 
7:      $numberOfFails = 0$ 
8:     return True
9:   else
10:    if  $numberOfFails > maxNumberOfFails$  then
11:       $T = T * \alpha$ 
12:       $numberOfFails = 0$ 
13:    else
14:       $numberOfFails = numberOfFails + 1$ 
15:    end if
16:    return False
17:  end if
18: end function

```

provide a dissimilarity measure between two observations by utilizing the sample covariance matrix [41].

As each new configuration has a corresponding partial simulation of length t_{search} starting from a previous configuration, we compute the novelty for the trace of that partial simulation using Algorithm 2.

3.4.1.9 Termination Condition

Our algorithm checks a set of termination conditions to stop the search and returns the configuration which has the minimum cost associated with it. One of the termination conditions we use is a threshold for the minimum interesting cost.

Algorithm 2 Algorithm used to check the novelty of a new configuration.

```

1: function ISNOVEL( $\zeta, \mathcal{E}, \mathcal{A}, k_{start}, k_{end}, c_{prev}, c_{cand}$ )
2:    $\mathbf{N}_{last}$  is persistent and keeps the last computed 10 novelty values.
3:   numR is persistent and keeps the number of rejections.
4:   maxReject is the maximum number of consecutive rejections.
5:    $X_{rel,k}$  is persistent and keeps the set of all past relative state computations.
6:    $c_{prev}$  and  $c_{cand}$  are the costs associated with the previous and candidate configurations, respectively.
7:
8:   Initialize  $\mathbf{N}$  as an empty set
9:   for each  $\mathbf{e}_i \in \mathcal{E}$  do
10:    for each  $\mathbf{a}_j \in \mathcal{A}$  do
11:      for  $k = k_{start}$  to  $k_{end}$  do
12:        Compute  $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$  from  $\zeta$ 
13:        Compute  $m$ -nearest neighbors of  $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$  in  $X_{rel,k-1}$ 
14:        Compute  $\mathcal{N}$  (novelty) with Eq. (3.4)
15:        Add  $\mathcal{N}$  to  $\mathbf{N}$ 
16:        Add  $\mathbf{x}_{\mathbf{e}_i, \mathbf{a}_j}[k]$  to  $X_{rel,k}$ 
17:      end for
18:    end for
19:  end for
20:   $\mathcal{N} = \max(\mathbf{N})$ 
21:  Update  $\mathbf{N}_{last}$  to keep the last computed 10 novelty values
22:  if  $c_{cand} < 0.9c_{prev}$  or numR > maxReject or  $|\mathbf{N}_{last}| < 10$  or  $\mathcal{N} > \text{mean}(\mathbf{N}_{last})$ 
    then
23:    numR = 0
24:    return True
25:  else
26:    numR = numR + 1
27:    return False
28:  end if
29: end function

```

Another termination condition is a preset maximum overall time spent. Alternative termination conditions can be used, *e.g.*, a maximum number of nodes in the search tree.

3.4.2 Case Studies

Here, we present 2 case studies and compare our RRT-based approach with our falsification-based approach described in Section 3.3.

3.4.2.1 Case Study 1

Scenario Setup

In this case study, we have 2 agent vehicles and 1 Ego vehicle on a 3-lane straight road, *i.e.*, $\mathcal{A} = \{\mathbf{a}_1, \mathbf{a}_2\}$ and $\mathcal{E} = \{\mathbf{e}\}$. Figure 12 gives an high-level overview of our simulation setup. The initial position of agent vehicle 1 on the x axis is randomly sampled between 0 m and 25 m, the initial x position of agent vehicle 2 is randomly sampled between 10 m and 20 m, and the initial x position of Ego vehicle is randomly sampled between 30 m and 50 m. The initial positions of agent vehicles on the y axis are sampled between the centers of lane 1 and lane 3, *i.e.*, between -3.5 m and 3.5 m, and the initial y position of Ego vehicle is randomly sampled between -1.75 m and 1.75 m, that is the lane markings separating Lane 3 and Lane 1 from Lane 2, respectively. The initial orientation of the Ego vehicle with respect to the x axis is randomly sampled between $-\pi/8$ rad and $\pi/8$ rad. The initial speed of Ego vehicle is sampled between 10 m/s and 15 m/s while the target speed is fixed to 15 m/s. The

initial speeds of the agent vehicles are sampled between 0 m/s and 15 m/s, and their target speeds at each waypoint are sampled between 0 m/s and 30 m/s.

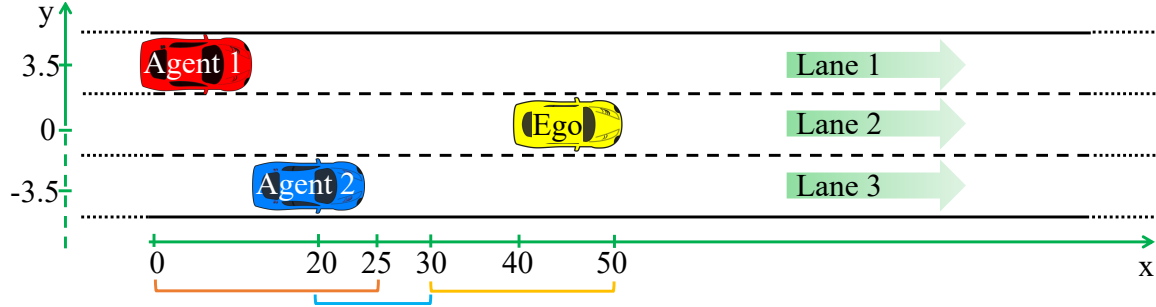


Figure 12. Initial states of the vehicles in the simulation setup for case study 1.

Ego vehicle has 5 sensors. Figure 13 visualizes the sensor placement and ranges of the sensors. A long-range sensor with a 45° field of view and 60 m range is placed at the front of the vehicle. Two 10 m-range sensors with 90° field-of-view are placed on the sides, facing left and right. Two 10 m-range sensors with 90° field-of-view are placed at the rear-left and rear-right corners with an angle to scan the area behind the rear corners of the vehicle.

Agent vehicles are controlled by the *move-to-pose* controller described in [38]. Ego vehicle controller has multiple modes based on the collision risk. When there is no estimated collision risk, a proportional control is applied to track target driving speed. If a collision is estimated in front, emergency brakes are applied and at the same time, depending on the occupancy of rear-left and rear-right areas, left or right steering is applied, respectively. If a collision is estimated in front-left (right), emergency brakes are applied and if rear-right (left) area is empty, also steering is applied to the right (left). If a collision is estimated in rear-left (right), if front and front-right (left) areas are empty, vehicle is accelerated with a right (left) steering, if only front

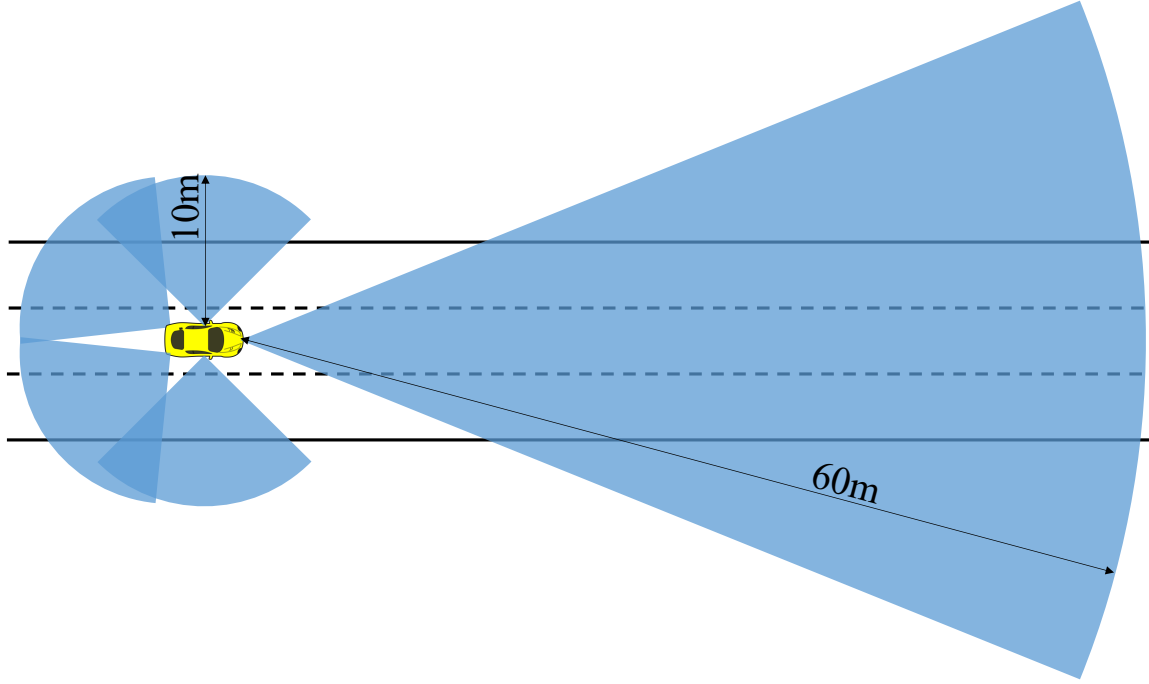


Figure 13. Ego vehicle sensor setup for case study 1.

area is empty and the vehicle on the front-right (left) is not imposing a risk, vehicle is accelerated without steering, otherwise emergency brakes are applied and right (left) steering is applied if the rear-right (left) area is empty. If the area to which a maneuver is being done gets occupied during the maneuver, emergency brakes are applied. Control switches back to normal mode if there is no more collision risk and a predefined time has passed since the last collision estimation. The collision avoidance algorithm presented here is very simple and it is not comparable to a controller that could be found in a real ADS. However, since our target in this work is to study test generation approaches, rather than proposing a controller, we argue that this naive control approach is satisfactory for the purpose of this work. For the lateral control, we use the Stanford Racing Team's controller that was used in the DARPA Grand challenge [86].

Experiment Results

We have run 200 experiments with the falsification approach and with the RRT-based approach. The minimum, mean and maximum costs achieved by the falsification approach were 0.0001, 12.4794, and 100.6082, respectively. The minimum, mean and maximum costs achieved by the RRT-based approach were 3.9124, 17.7190, and 88.9793, respectively. Figure 14 and Fig. 15 visualize the minimum-cost trajectories returned by the RRT-based and falsification-based approach, respectively. Histories of the vehicles are numbered to show their evolution over time. Figure 16 provides box and whisker diagrams of the minimum costs achieved by the two approaches among the 100 experiments we have carried. The black diamonds plotted on top of the box plots show the mean values for the returned minimum costs. In this case study, the falsification-based approach achieved smaller mean cost values, as well as the smaller minimum cost compared to the RRT-based approach. One reason for this is that, since the space between agent vehicles and Ego vehicle is open, there are not many local minimums that would make the exploration capabilities to achieve better than falsification-based approach. As it is easy to find a trajectory that is in the neighborhood of an interesting case, falsification approach can focus on that neighborhood and minimize the cost as much as possible while RRT-based approach keeps looking for novel trajectories.

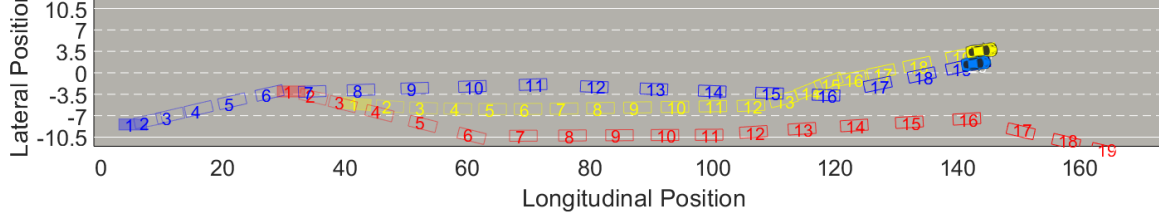


Figure 14. The minimum cost result returned by the RRT-based approach in Case Study 1.

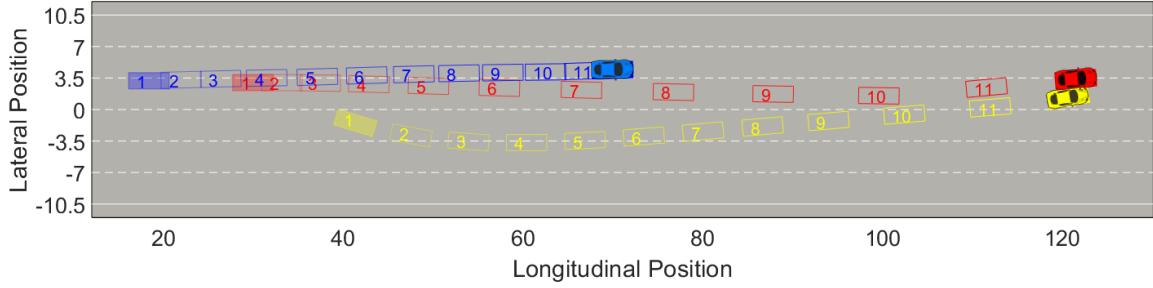


Figure 15. The minimum cost result returned by the falsification-based approach in Case Study 1.

3.4.2.2 Case Study 2

Scenario Setup

In this case study, we have 4 agent vehicles and 1 Ego vehicle on a multiple-lane straight road, *i.e.*, $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$ and $\mathcal{E} = \{e\}$. Figure 17 gives an high-level overview of our simulation setup. The initial position of agent a_1 on the y axis is randomly sampled between 1.25 m and 6 m, the initial y positions of agent vehicles a_2, a_3, a_4 are randomly sampled between -2.25 m and -1.75 m. The initial speed of a_1 is randomly sampled between 5 m/s and 15 m/s, and its target speed at each waypoint is sampled between 0 m/s and 30 m/s. The initial and target speeds of all other vehicles are set to 15 m/s. All other initial states of the vehicles are fixed.

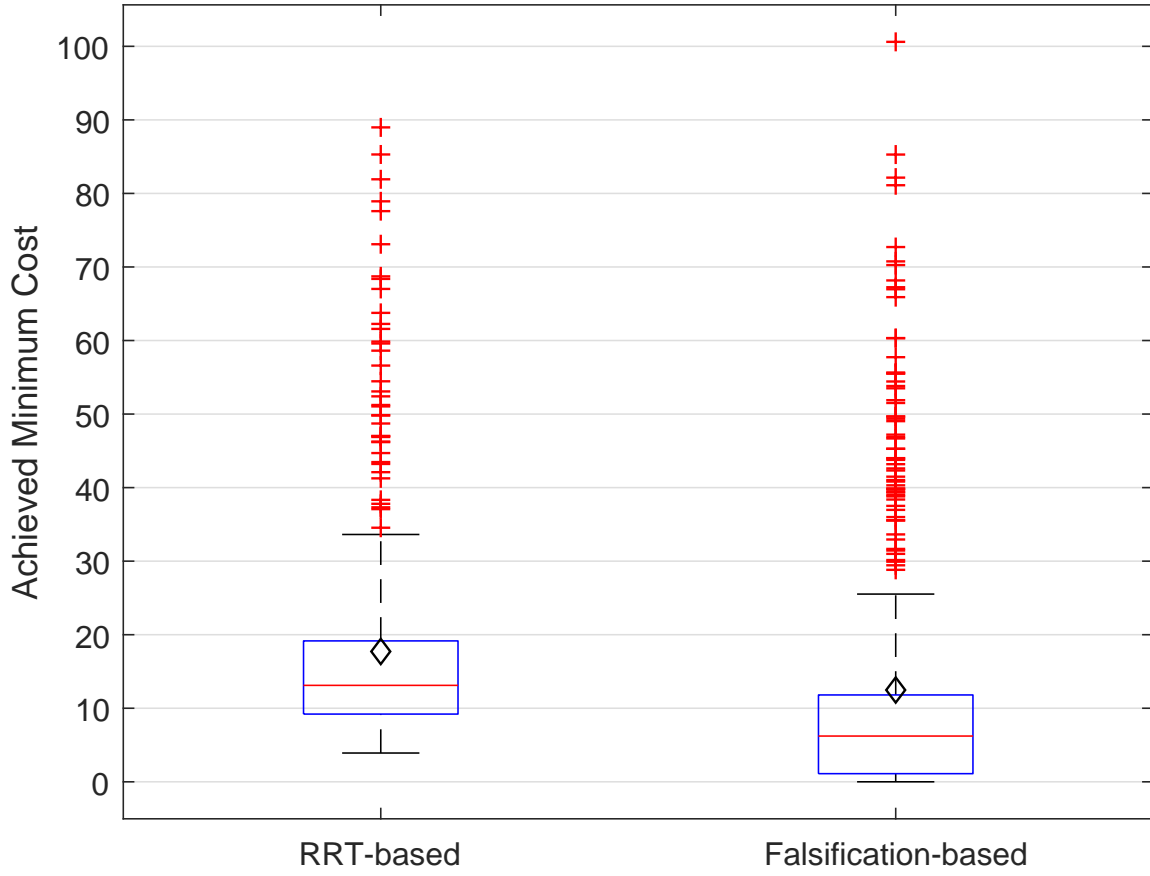


Figure 16. Comparison of the minimum cost achieved by the falsification approach and the RRT-based approach in Case Study 1.

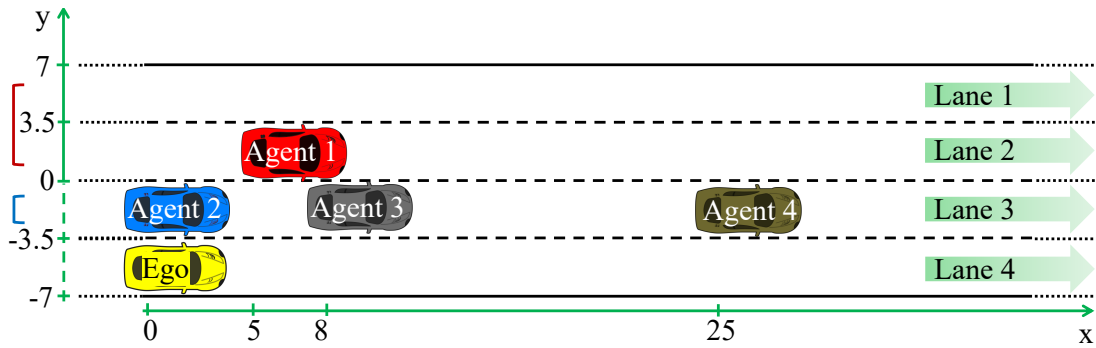


Figure 17. Initial states of the vehicles in the simulation setup for case study 2.

Ego vehicle has 5 sensors. Figure 18 visualizes the sensor placement and ranges of the sensors. A long-range sensor with a 22.5° field of view and 50 m range is placed at the front of the vehicle. Two 5 m-range sensors with 90° field-of-view are placed on the sides, facing left and right. Two 7 m-range sensors with 90° field-of-view are placed at the rear-left and rear-right corners with an angle to scan the area behind the rear corners of the vehicle.

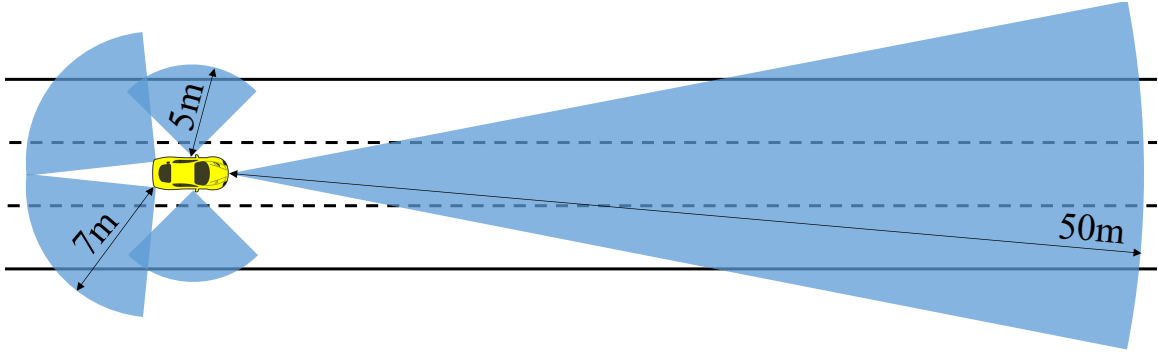


Figure 18. Ego vehicle sensor setup for case study 2.

Agent vehicle \mathbf{a}_1 is controlled by the *move-to-pose* controller described in [38]. Agent vehicles \mathbf{a}_2 , \mathbf{a}_3 , and \mathbf{a}_4 are driven with a constant speed on a straight line. Ego vehicle controller is the same as the one described in Case Study 1.

Experiment Results

For this case study, we only search for an optimal trajectory for \mathbf{a}_1 with the target of minimizing the cost function described in Eq. (3.3). The existence of agent vehicles \mathbf{a}_2 , \mathbf{a}_3 , and \mathbf{a}_4 between Ego vehicle \mathbf{e} and agent vehicle \mathbf{a}_1 creates many local minima for the selection of \mathbf{a}_1 trajectories.

In this case study, we have executed 100 experiments with both RRT-based

approach and falsification-based approach in MATLAB. Each run of both approaches had a time-out duration of 30 min. Out of 100 runs, only 5 of the minimum-cost trajectories returned by the falsification-based approach was able to make \mathbf{a}_1 to move into the lane of Ego vehicle, and only 2 trajectories were able to cause a collision (1 high-speed collision and 1 boundary-case collision) and, other than the collision cases, only 1 trajectory was able to challenge Ego vehicle by activating its collision avoidance system. All other trajectories returned by the falsification approach were stuck in local-minima where \mathbf{a}_1 tries to get closer to Ego vehicle and ends up colliding with one of the other agent vehicles. On the other hand in 28 of the minimum-cost trajectories returned by the RRT-based approach, agent \mathbf{a}_1 was able to get into the lane of Ego vehicle and it was able to cause Ego vehicle to collide in 11 of those cases.

Figure 19 shows one of the interesting collision cases discovered by the RRT-based approach. Agent \mathbf{a}_1 first forces Ego to move to the right to avoid a collision and then to the left where it ends up colliding with Agent \mathbf{a}_3 . Histories of \mathbf{a}_1 (red) and Ego (yellow) vehicles are numbered to show their evolution over time. Figure 20 shows the only small-speed collision case discovered by the falsification-based approach. Agent \mathbf{a}_1 moves into the Ego vehicle’s lane, accelerates and rear-ends with Ego vehicle even though Ego vehicle tries to avoid the collision by accelerating and steering away. Figure 21 shows a typical trajectory that is stuck in a local minimum. Agent \mathbf{a}_1 tries to move closer to Ego vehicle and reduces the time-to-collision but collides with one of the other agents, which is \mathbf{a}_2 in this figure. Although both approaches can get stuck in a local minimum, this case is significantly more common for the falsification-based approach as discussed above.

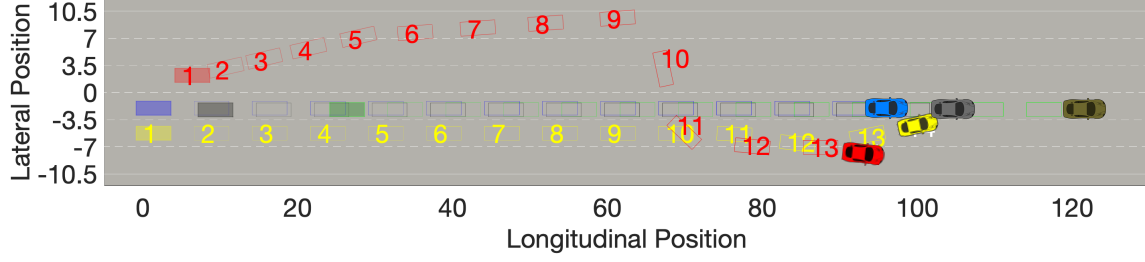


Figure 19. One of the collision cases discovered by the RRT-based approach.

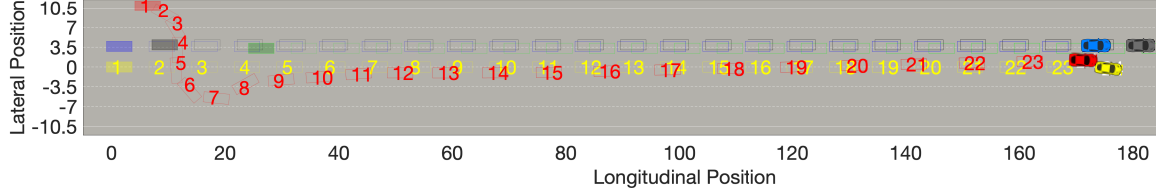


Figure 20. The small-speed collision discovered by the falsification-based approach.

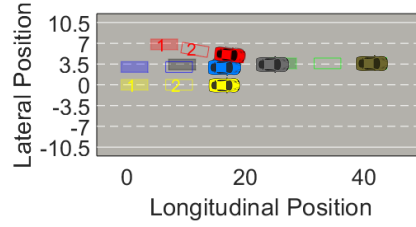


Figure 21. One of the cases where falsification-based approach was stuck in a local minimum.

As a numerical comparison of the minimum costs discovered by the two approaches, the minimum, mean and maximum costs achieved by the falsification approach were 0.0043, 13.7134, and 50.6017, respectively. The minimum, mean and maximum costs achieved by the RRT-based approach were 4.8955, 10.2571, 15.0856. Figure 22 provides box and whisker diagrams of the minimum costs achieved by the two approaches among the 100 experiments we have carried. The black diamonds plotted on top of the box plots show the mean values for the returned minimum costs. While 44 out of 100

RRT-based approach experiments were able to avoid local minima occurring around the cost 10, only 2 of the falsification approach experiments were able to achieve this.

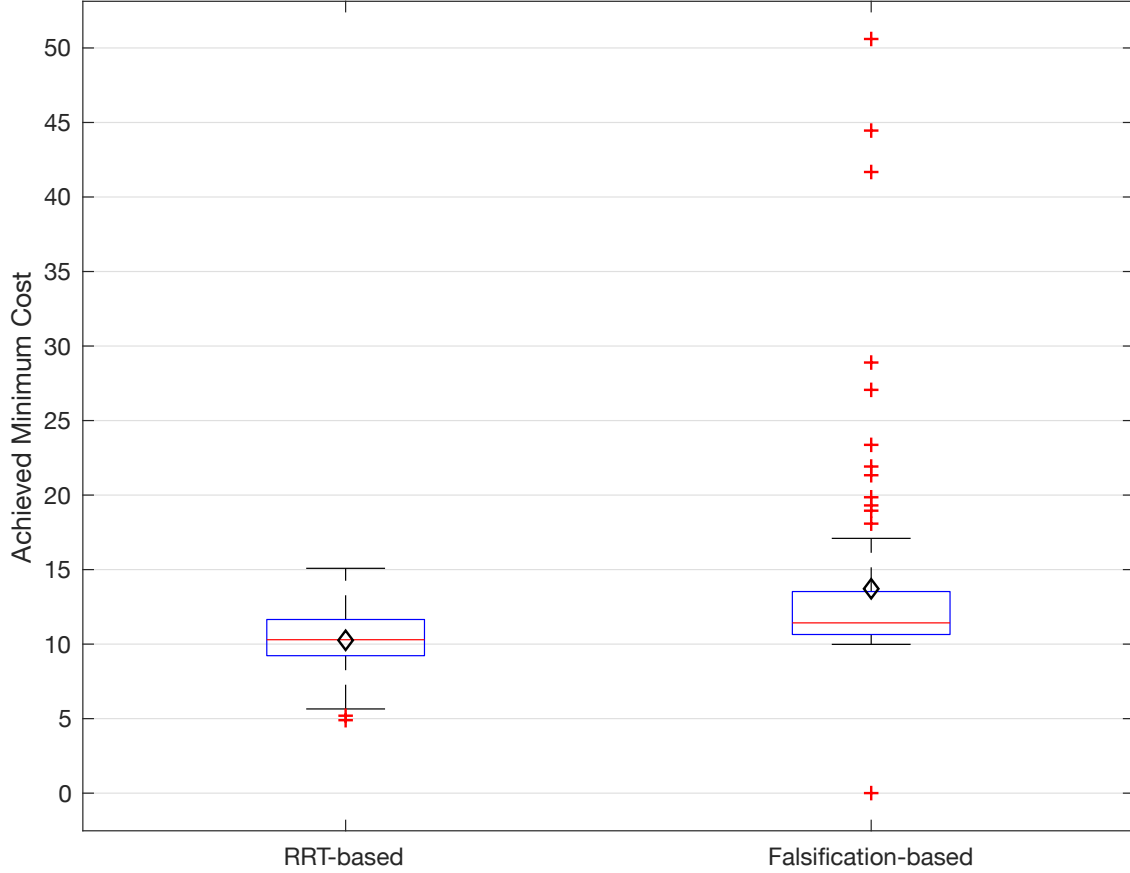


Figure 22. Comparison of the minimum cost achieved by the falsification approach and the RRT-based approach in Case Study 2.

An observation we would like to share is that in the single case where the falsification approach was able to cause a small speed collision, the achieved minimum cost was significantly smaller than any of the 11 collision cases discovered by the RRT-based approach. This observation supports our conclusion in Case Study 1, *i.e.*, when the falsification-based approach can get into the neighborhood of a local or global minimum, it is more able to get closer to the minimum point than the RRT-based approach. The

advantage of the RRT-based approach in avoiding local minima and the advantage of the falsification-based in reaching minima suggests that an approach combining these two has the potential to improve the overall test generation performance. In such an approach, firstly, RRT-based approach would discover neighborhoods of minima and then the falsification-based approach would guide the test toward the minima starting from the results of the RRT-based approach.

3.5 Functional Gradient Descent-based Approach

In principle, the vehicle models and the model of the environment should be of sufficiently high fidelity to exercise and detect model behaviors consistent with the real system. However, as the input and state space become larger, such a computational optimization approach would require an increased number of simulations and would become impractical for computationally expensive high-fidelity models.

Recently, functional gradient descent methods have been proposed as an approach to reduce the total number of simulations needed to converge to a local minimum of the cost function that captures the system safety. Abbas *et al.* [8] utilize the functional gradient descent method for minimizing cost metrics for Metric Temporal Logic (MTL) specifications, and that work was extended by Yaghoubi and Fainekos [185] to approximate the descent direction using a set of carefully chosen system linearizations. Here, we consider an alternative approach for the problems where the computation of system linearizations on the high-fidelity model is not practical or desirable.

In this section, we explore whether applying functional gradient descent over a simpler system model can assist in the test generation process for falsifying the original more complex model. In particular, we propose using a simplified, alternative version

of the system dynamics as much as possible in the process of generating test cases that aim to find the worst-case performance of the system. We show that (1) the proposed approach can improve upon human-generated test cases, which may rely on intuition or prior knowledge, and that (2) it can generate interesting test cases from scratch. Our approach relies on applying the gradient-based optimization technique of Abbas *et al.* [8] on simplified system dynamics for quickly updating the inputs to lead to worse system performance on the more complex model.

The primary contribution of our work is the enhancement of functional gradient descent for falsification [8] with methods from multi-fidelity optimization so that these automatic falsification techniques can be applied to a wider range of complex, realistic models.

3.5.1 Definitions

Following the notation and definitions given in Section 3.2, we describe a driving scenario $\mathcal{T} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$ that contains a set of Ego vehicles, agents and the surroundings. We consider the overall driving scenario as a single system $\mathcal{M}_{\mathcal{T}} = (X_{\mathcal{T}}, \mathbf{U}_{\mathcal{T}}, P_{\mathcal{T}}, sim_{\mathcal{T}})$. The state vector for the overall system is \mathbf{x} , and we denote the flow through the overall system by $\dot{\mathbf{x}} = \mathbf{F}(t, \mathbf{x}, \mathbf{u})$ where \mathbf{u} is the vector of inputs to the system. We denote a finite-time state trace of the overall system with input signal \mathbf{u} by $\zeta_{\mathbf{u}}$ where $\zeta_{\mathbf{u}}(t)$ is the value of the system states at the time $t \in [t_0 \leq t \leq T]$.

We denote the performance metric for the system by G , where $G(\zeta_{\mathbf{u}}(t))$ gives the instantaneous performance of the system at the time t . The worst-case performance of the system over the trajectory $\zeta_{\mathbf{u}}$ starting from time t_0 up to the time T is defined as:

$$\mathcal{R}(\zeta_{\mathbf{u}}, [t_0, T]) = \min_{t_0 \leq t \leq T} G(\zeta_{\mathbf{u}}(t)) \quad (3.5)$$

This minimum type (non-additive) objective function is similar to the one considered by Melikyan *et al.* [122]. For brevity, we will omit the $[t_0, T]$ in the rest of this section and simplify the notation to $\mathcal{R}(\zeta_{\mathbf{u}})$.

The focal problem in this section is to find an input signal that will globally or locally minimize the worst-case performance of the system over a trajectory of length T over the feasible input space. For this problem, we define an input signal as a sequence of n input points for a simulation of n time steps, *i.e.*, $T = n\Delta t$ where Δt is the simulation time step. Formally, this problem is to find the input signal \mathbf{u}^* such that

$$\mathbf{u}^* = \operatorname{argmin}_{\mathbf{u} \in \mathcal{U}^n} \mathcal{R}(\zeta_{\mathbf{u}}) \quad (3.6)$$

where $\mathcal{U} \triangleq [u_{\min}, u_{\max}]$, and u_{\min}, u_{\max} are the minimum and the maximum values for the input signals, respectively.

3.5.2 Solution

Our approach for this problem is based on the functional gradient descent method described by Abbas *et al.* [8]. This method relies on using sensitivity analysis and computing gradients of the system dynamics to search for a local minimizer of Eq. (3.5). However, in many cases either the symbolic representation of the actual system dynamics is not available or it is not possible to compute descent directions for the actual system dynamics because $\mathbf{F}(t, \mathbf{x}, \mathbf{u})$ may, in general, be nonlinear, high dimensional, time-varying, and it may incorporate time delays. So we propose to use an approximate low-fidelity model to guide the search for the minimizer on the original high-fidelity model. The low-fidelity model should be selected such that the

gradients of the system dynamics can be computed analytically. That is, it should not contain time delays and discontinuities.

We denote the flow for the low-fidelity model by $\dot{\tilde{\mathbf{x}}} = \tilde{\mathbf{F}}(t, \tilde{\mathbf{x}}, \mathbf{u})$, and the system trajectory under the input \mathbf{u} by $\tilde{\zeta}_{\mathbf{u}}$. We illustrate an overview of our approach in Fig. 23.

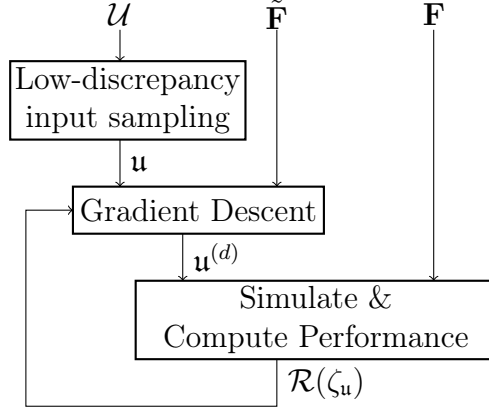


Figure 23. Solution Overview.

From a random input signal, we use the low-fidelity model of the system with functional gradient descent computations for updating the input such that the new input is guaranteed to lead to smaller $\mathcal{R}(\tilde{\zeta}_{\mathbf{u}})$ on the low-fidelity model. Then, we apply the updated input signal on the high-fidelity model to get the resulting $\mathcal{R}(\zeta_{\mathbf{u}})$. This information is used to update the step size in functional gradient descent using the bisection method until the stopping criteria are met. Finally, we do low-discrepancy sampling [57] to get a new random input and repeat this process.

Our approach is given as a pseudo-code in Algorithm 3. We first initialize the descent step size to a predefined value and the minimum performance to infinity. Starting with an empty set of previous samples, we sample an input signal from the set of possible inputs and add the new sample into the set of previous samples. We use simplified system dynamics ($\tilde{\mathbf{F}}$) to compute a descent direction, $d\mathbf{u}(t)$, which

is a vector such that $G(\tilde{\zeta}_{\mathbf{u}^{(d)}}(t^*)) < G(\tilde{\zeta}_{\mathbf{u}}(t^*))$ when $\mathbf{u}^{(d)}(t^*) = \mathbf{u}(t) + h \cdot d\mathbf{u}(t)$ for all sufficiently small h . Because G is a function of $\tilde{\mathbf{x}}$, we choose the descent direction

$$d\mathbf{u}(t) \triangleq - \left(\frac{\partial \tilde{\zeta}_{\mathbf{u}}(t)}{\partial \mathbf{u}} \right)^{\top} \frac{\partial G(\tilde{\zeta}_{\mathbf{u}}(t))}{\partial \tilde{\mathbf{x}}} \quad (3.7)$$

where $\frac{\partial \tilde{\zeta}_{\mathbf{u}}(t)}{\partial \mathbf{u}}$ is the sensitivity of the system trajectory to the input, which is computed according to

$$\frac{d}{dt} \frac{\partial \tilde{\zeta}_{\mathbf{u}}(t)}{\partial \mathbf{u}} = \frac{\partial}{\partial \mathbf{u}} \tilde{\mathbf{F}}(t, \tilde{\mathbf{x}}, \mathbf{u}) = \frac{d\tilde{\mathbf{F}}}{d\tilde{\mathbf{x}}} \frac{\partial \tilde{\zeta}_{\mathbf{u}}(t)}{\partial \mathbf{u}} + \frac{d\tilde{\mathbf{F}}}{du} \quad (3.8)$$

from the initial condition $\frac{\partial \tilde{\zeta}_{\mathbf{u}}(0)}{\partial \mathbf{u}} = 0$ where $\frac{d\tilde{\mathbf{F}}}{d\tilde{\mathbf{x}}}$ and $\frac{d\tilde{\mathbf{F}}}{du}$ are the Jacobian matrices of the system flow along the trajectory. Because

$$dG(\tilde{\zeta}_{\mathbf{u}}(t)) = \frac{\partial G(\tilde{\zeta}_{\mathbf{u}}(t))}{\partial \tilde{\mathbf{x}}} \frac{\partial \tilde{\zeta}_{\mathbf{u}}(t)}{\partial \mathbf{u}} d\mathbf{u}(t) \quad (3.9)$$

from the Eq. (3.7), we have that $dG(\tilde{\zeta}_{\mathbf{u}}(t)) \leq 0$. So our choice of $d\mathbf{u}(t)$ will not increase G as desired.

We then update the input signal in the computed descent direction. We used $\tilde{\mathbf{F}}$ to compute the descent direction, but we use \mathbf{F} to evaluate the performance of the system under the updated input signal. If we achieve a decrease in the worst-case performance, then we accept the descent direction, and we increase the descent vector length by 50% for the next iteration to speed up the search for a local minimum. On the other hand, if the updated input leads to a larger worst-case performance on \mathbf{F} , then we use the bisection method on the descent vector and search for a smaller worst-case performance value with a maximum number of bisections.

For the functional gradient descent computation, we use the time at which the value of G is smallest for the system under the input \mathbf{u} as the critical time t^* . Hence, $\mathcal{R}(\tilde{\zeta}_{\mathbf{u}}) = G(\tilde{\zeta}_{\mathbf{u}}(t^*))$. In order to represent the input signals in a digital computer and to reduce the dimensionality of our search space, we parameterize the input signals

using the approach described by Abbas *et al.* [4]. Fundamentally, we represent an input signal with a finite, ordered set of parameters where each parameter corresponds to a time and contains the value of the input at that time. The functional gradient descent method converges to a local minimum. To search for smaller local minima, we repeat the search from different initial input signals sampled randomly from a set of input signals and selected to be farthest from previous samples [57].

3.5.3 Case Study

Mullakkal-Babu *et al.* [125] propose a Full Range Adaptive Cruise Control (FRACC) design with rear-end collision avoidance capability which is referred to as *P-FRACC*. They evaluate the performance of the proposed controller using Maximum Absolute Jerk (MAJ), which is used as an indication of driving comfort. They have used a *stop and go* scenario for evaluating the MAJ performance of the controller on an individual vehicle. This scenario consists of an external lead vehicle (*leader*) and a vehicle controlled by P-FRACC, and it is aimed to capture crowded highway driving conditions at small speeds. In the *stop and go* scenario, the leader starts with an initial speed of 5.5 m/s and constantly decelerates to 0 m/s starting at time 5 s with a deceleration of 0.39 m/s². Then, starting at time 40 s, the leader constantly accelerates to 15.6 m/s in 40 s. Finally, starting from the time 130 s, it decelerates to 0 m/s with a constant deceleration of 0.39 m/s².

Such a scenario designed by intuition and prior knowledge may capture the target operating conditions that are challenging for the vehicle under test. However, it would require very detailed system models and intensive analysis to understand whether a human-designed test scenario is the most challenging one based on the utilized

Algorithm 3 Functional Gradient Descent on Simplified Dynamics Method.

```
1:  $\mathbf{F}, \tilde{\mathbf{F}} \leftarrow$  Actual and simplified dynamics of the system
2:  $P_{min,all} \leftarrow \infty$ 
3: for Maximum number of starts do
4:    $h \leftarrow h_{init}; P_{min}, P_{prev} \leftarrow \infty; \text{iter} \leftarrow 0; \mathbf{U} \leftarrow \emptyset$   $\triangleright \mathbf{U}$  is the set of previous samples
5:    $\mathbf{u} \leftarrow \text{GETNEWSAMPLE}(\mathbf{U}, [u_{min}, u_{max}]^n)$   $\triangleright$  Input, as an  $n$ -dimensional vector
6:    $\mathbf{U} \leftarrow \mathbf{U} \cup \{\mathbf{u}\}$ 
7:   while iter < Maximum number of iterations do
8:      $d\mathbf{u} \leftarrow \text{COMPUTEDU}(\tilde{\mathbf{F}}, \mathbf{u}, G)$ 
9:      $\mathbf{u} \leftarrow \mathbf{u} + d\mathbf{u}/\|d\mathbf{u}\|$ 
10:    bisectCallCount  $\leftarrow 0$ 
11:    while bisectCallCount < Max. # of bisections do
12:       $P \leftarrow \mathcal{R}(\zeta_{\mathbf{u}})$   $\triangleright \mathbf{F}$  performance under  $\mathbf{u}$ 
13:      if  $P < P_{prev}$  then
14:         $h \leftarrow 1.5h; P_{min} \leftarrow P; \mathbf{u}^{(d)} \leftarrow \mathbf{u}$ 
15:        break
16:      else
17:         $h \leftarrow h/2; \text{bisectCallCount} \leftarrow \text{bisectCallCount} + 1$ 
18:      end if
19:    end while
20:    if bisectCallCount == Maximum number of bisections then
21:      break  $\triangleright$  Local minimum found
22:    end if
23:  end while
24:  if  $P_{min} < P_{min,all}$  then
25:     $P_{min,all} \leftarrow P_{min}; \mathbf{u}_{best}^{(d)} \leftarrow \mathbf{u}^{(d)}$ 
26:  end if
27: end for
28: return  $P_{min,all}, \mathbf{u}_{best}^{(d)}$ 

29: function COMPUTEDU( $F, \mathbf{u}, G$ )
30:    $\zeta \leftarrow$  State trace of the system  $F$  under the input  $\mathbf{u}$ 
31:    $t^* \leftarrow$  Critical time over  $\zeta$  w.r.t performance metric  $G$ 
32:   Compute  $d\mathbf{u}$  using Eq. (3.7)
33: end function

34: function GETNEWSAMPLE( $\mathbf{U}, \mathcal{U}$ )
35:    $S \leftarrow$  Randomly sampled set from  $\mathcal{U}$ 
36:    $\mathbf{u} \leftarrow$  The element of  $S$  with the maximum distance to the points in  $\mathbf{U}$ 
37:   return  $\mathbf{u}$ 
38: end function
```

performance metric. The safety or functional requirements of an adaptive cruise control system would contain boundaries on the acceptable values of a performance metric. Hence, it is important to find the test cases that push the system under test towards and, possibly, beyond these boundaries.

3.5.3.1 Gradient Descent Computation

In our experimental setup, we use an agent vehicle as the lead vehicle, and we refer to the vehicle controlled by P-FRACC as the vehicle under test. Hence, we have $\mathcal{E} = \{\mathbf{e}\}$ and $\mathcal{A} = \{\mathbf{a}\}$. Considering the acceleration of the agent vehicle as the input, the state vector of the agent vehicle in the low-fidelity model is $\tilde{\mathbf{x}}_{\mathbf{a}} = [v_{\mathbf{a}}, x_{\mathbf{a}}]^T$ where $v_{\mathbf{a}}$ and $x_{\mathbf{a}}$ are respectively the longitudinal velocity and position. The state vector for the VUT in the low-fidelity model is $\tilde{\mathbf{x}}_{\mathbf{e}} = [a_{\mathbf{e}}, v_{\mathbf{e}}, x_{\mathbf{e}}]^T$ where $a_{\mathbf{e}}$, $v_{\mathbf{e}}$, and $x_{\mathbf{e}}$ are respectively the longitudinal acceleration, velocity, and position. Hence, the state vector for the overall system in the low-fidelity model is $\tilde{\mathbf{x}} = [v_{\mathbf{a}}, x_{\mathbf{a}}, a_{\mathbf{e}}, v_{\mathbf{e}}, x_{\mathbf{e}}]^T$, and we apply functional gradient descent using piecewise-constant input signals with the low-fidelity dynamics:

$$\dot{\tilde{\mathbf{x}}} = \tilde{\mathbf{F}}(t, \tilde{\mathbf{x}}, u) = \left[u, v_{\mathbf{a}}, \frac{a_{des} - a_{\mathbf{e}}}{\tau_a}, a_{\mathbf{e}}, v_{\mathbf{e}} \right]^T \quad (3.10)$$

where τ_a is the actuation delay and desired acceleration a_{des} is the output from the P-FRACC controller from [125]. In particular,

$$a_{des} \triangleq \begin{cases} K_1 s_{\Delta} + K_2 (v_{\mathbf{a}} - v_{\mathbf{e}}) R(s) & \text{if } s \leq r^F, \\ K_1 (v_{des} - v_{\mathbf{e}}) t_d & \text{otherwise} \end{cases} \quad (3.11)$$

where K_1 and K_2 are the control gains, t_d is the desired time gap, v_{des} is the desired velocity, $s \triangleq x_{\mathbf{a}} - x_{\mathbf{e}}$ is the distance between the leader and the vehicle under test, r^F

is the front sensing range of the vehicle under test, s_Δ is spacing error defined by

$$s_\Delta \triangleq \min(s - s_0 - v_e t_d, (v_{des} - v_e) t_d) \quad (3.12)$$

with desired minimum vehicle spacing s_0 , and $R(s)$ is a sigmoidal error-reponse function defined by

$$R(s) \triangleq \frac{Q}{Q + e^{(s/P)}} \quad (3.13)$$

where Q and P are aggressiveness, and the perception range coefficient parameters, respectively. For brevity, we omit “(t)” from state variables in the equations.

For the functional gradient descent, we only consider the case when the agent vehicle is within the sensing range of the VUT (*i.e.*, $s \leq r^F$) which, by Eq. (3.11), is the case when the behavior of the VUT depends on the agent vehicle. Furthermore, because a more aggressive control is expected when the vehicles are closer to each other, we take $s_\Delta = s - s_0 - v_e t_d$ which is active in Eq. (3.12) when the spacing error is more of a concern than the speed error. Under these assumptions, the simplified control law we use for our descent computations is

$$a_{des} = K_1(s - s_0 - v_e t_d) + K_2(v_a - v_e)R(s). \quad (3.14)$$

To find the input which will cause the worst performance, *i.e.*, the largest MAJ, we use the performance metric

$$G(\tilde{\zeta}_u(t)) = -(\dot{a}_e(t)(t))^2 = -\left(\frac{a_{des} - a_e(t)}{\tau_a}\right)^2 \quad (3.15)$$

which gives us

$$\frac{\partial G}{\partial \tilde{\mathbf{x}}} = -2 \frac{a_{des} - a_e}{\tau_a} \begin{bmatrix} K_2 R(s) \\ K_1 + K_2(v_a - v_e)R'(s) \\ -1 \\ -K_1 t_d - K_2 R(s) \\ -K_1 - K_2(v_a - v_e)R'(s) \end{bmatrix} \quad (3.16)$$

where

$$R'(s) = \frac{-\left(\frac{Q}{P}\right)e^{(s/P)}}{(Q + e^{(s/P)})^2}$$

. Then, by Eq. (3.8), the sensitivity for our case study is:

$$\frac{d}{dt} \frac{\partial \tilde{\zeta}_u(t)}{\partial u} = \frac{d\tilde{\mathbf{F}}}{d\tilde{\mathbf{x}}} \frac{\partial \tilde{\zeta}_u(t)}{\partial u} + \frac{d\tilde{\mathbf{F}}}{du} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ & & \frac{\partial G}{\partial \tilde{\mathbf{x}}}^\top & & \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \frac{\partial \tilde{\zeta}_u(t)}{\partial u} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (3.17)$$

The descent direction is computed and applied to the input iteratively until the stopping criteria are met as discussed in Section 3.5.2.

The complex, high-fidelity system dynamics that we use for evaluating the descent vector incorporates sensor delay τ_s in the control law. In particular,

$$a_{des}(t) \triangleq \begin{cases} K_1 s_\Delta(t - \tau_s) + K_2 (v_a(t - \tau_s) - v_e(t - \tau_s)) R(s(t - \tau_s)), & s(t - \tau_s) \leq r^F \\ K_1 (v_{des} - v_e(t - \tau_s)) t_d, & \text{otherwise} \end{cases} \quad (3.18)$$

where

$$s_\Delta(t - \tau_s) = \min(s(t - \tau_s) - s_0 - v_e(t - \tau_s) t_d, (v_{des} - v_e(t - \tau_s)) t_d).$$

Furthermore, in the high-fidelity model, the velocity for the vehicles saturate at minimum and maximum velocities. Thus, applying functional gradient descent on the complex system dynamics is difficult due to the delay differential equations, the hybrid nature of the controller, and nonlinearities introduced for realism.

3.5.3.2 Stop and Go Scenario

We aim to automatically create a scenario to find the worst-case performance of P-FRACC and compare it to the original *stop and go* scenario from [125]. Our experimental setup is similar to the original setup described above. For the sake of staying inside the targeted operation region, we limit the minimum and maximum acceleration of the agent vehicle with -0.39 m/s^2 and 0.39 m/s^2 because these were the limit values in the original *stop and go* scenario. The accelerations of the vehicles simulated with the high-fidelity model in this scenario are shown in Fig. 24.

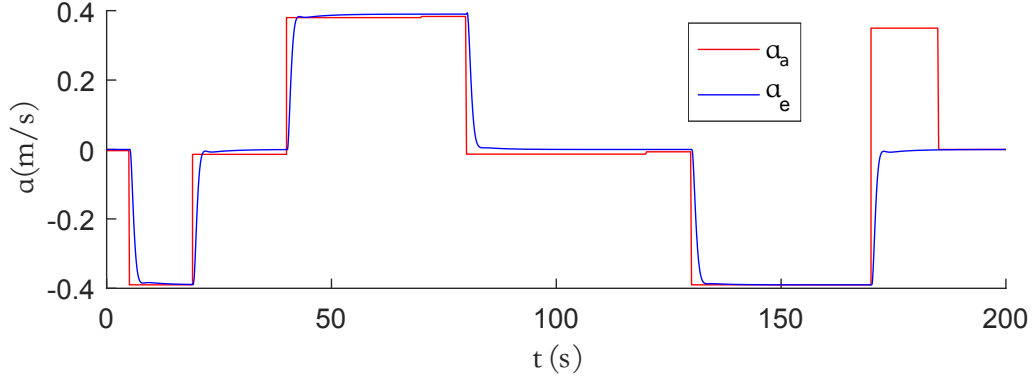


Figure 24. Accelerations for original *stop and go* scenario.

3.5.3.3 Experiment Results

We have implemented our experiments in MATLAB[®] [118]. We used the CVODES ordinary differential equation solver from SUNDIALS [84] to compute the performance and the sensitivity for the simple system dynamics, and we used the `dde23` solver [158] in MATLAB for simulating the complex system dynamics that contain delay differential equations.

For the experiments, we apply our approach with maximum of 1000 simulations. We compare the results of our approach with the Simulated Annealing (SA) optimization implementation in S-TALiRO [3]. For SA, we use the same random initial input signal that was used to start the functional gradient descent method, and we limit the maximum number of simulations with 1000. We utilize only the complex dynamics of the system in the SA optimization.

For comparison, we executed 190 experiments in total with our approach and with SA. In 167 of 190, *i.e.*, in 87.9% of the executed experiments, the proposed functional gradient descent method achieved a smaller worst-case performance on the complex dynamics of the system compared to the SA optimization. The average of the explored minimum worst-case performance is -0.6532 with our approach and -0.6363 with SA. When all experiment results are considered, although the difference is negligibly small, the minimum worst-case performance found with our approach (-0.6551) was also smaller than the one found by SA (-0.6550).

In Fig. 25, there is an entry (a blue star) for each experiment. Each entry represents how much the achieved minimum worst-case performance value by the SA is larger than the achieved minimum worst-case performance value by our approach for an experiment. The positive values (*i.e.*, the entries above the horizontal red line located at 0) are from the experiments where our approach outperformed SA, and the distance to 0 is a measure of how much it performed better than SA. Our approach outperforms SA in most of the experiments. Furthermore, the average performance improvement from our method is larger than the average performance improvement in the cases when SA outperforms our method.

The initial random acceleration input, the resulting input after functional gradient descent, and the corresponding performance through the simulation trajectory are

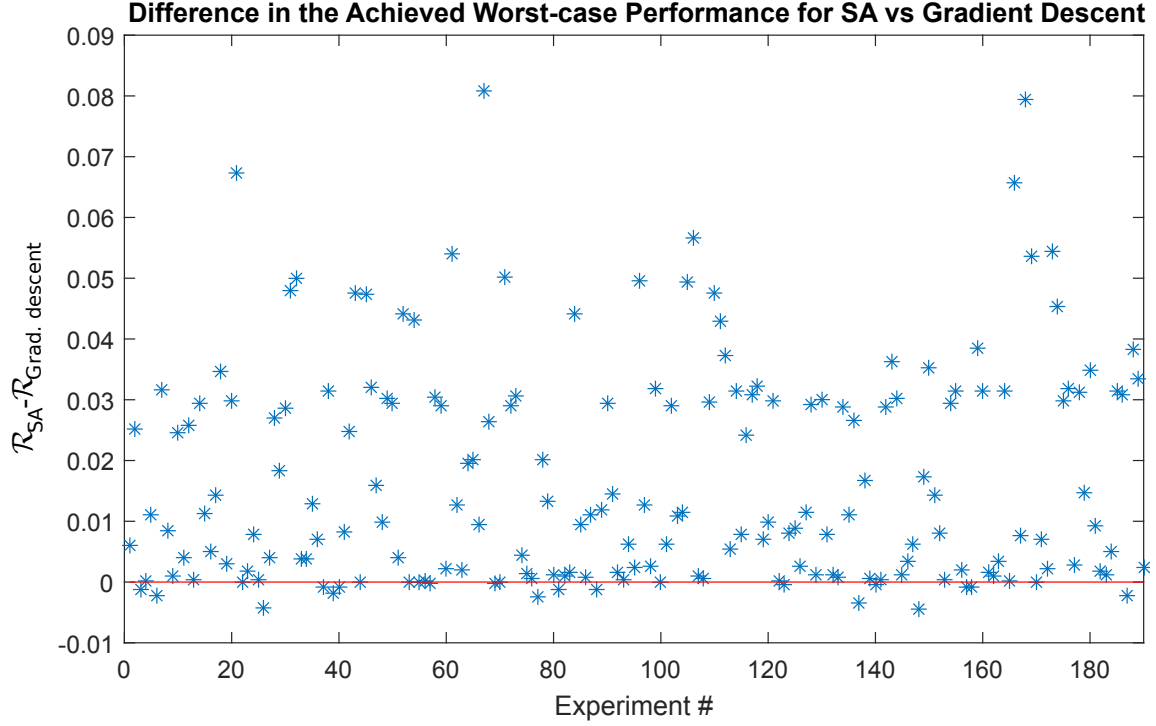


Figure 25. Comparison of the gradient descent-based approach with Simulated Annealing with differences in the achieved minimum worst-case performance values.

shown in Fig. 26. The figure is from the descent experiment which achieved the overall minimum worst-case performance. Figure 27 shows the accelerations of the vehicles for the input signal given in Fig. 26.

Comparing this test case with the original *stop and go* scenario, the maximum absolute jerk 0.6551 m/s^3 measured with this test case is larger than the maximum absolute jerk 0.3232 m/s^3 we have measured with the original *stop and go* scenario. Because of possible differences in implementation details and in the differential equation solvers, some numerical differences may exist between our results and those of [125]. The MAJ value 0.3232 m/s^3 we have measured is approximately 10 times larger than

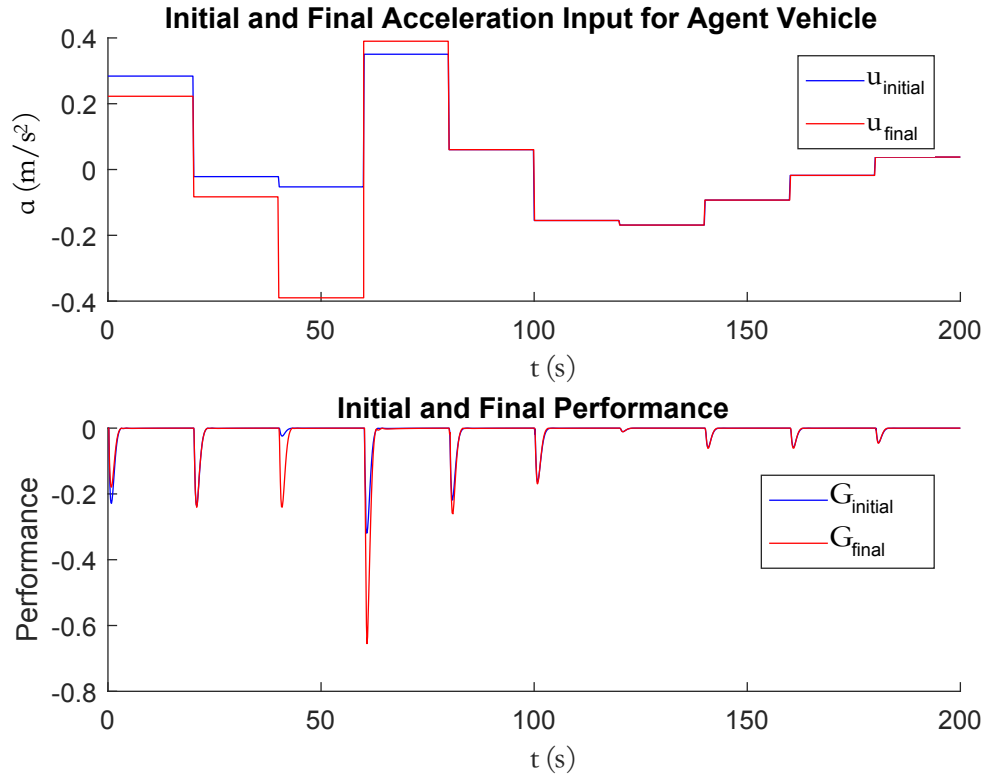


Figure 26. Agent vehicle acceleration and resulting worst-case performance.

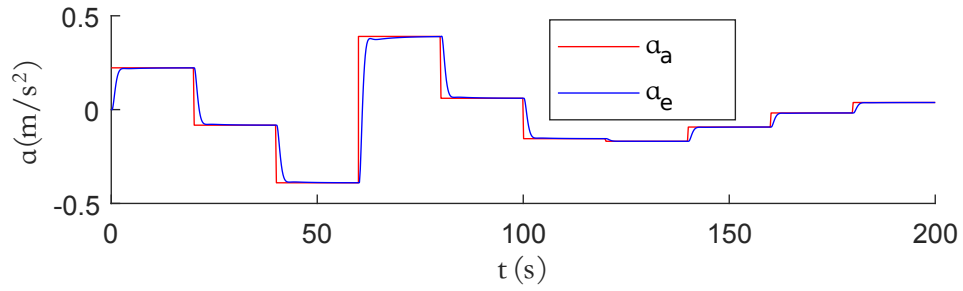


Figure 27. Accelerations of vehicles under updated input.

the results reported by Mullakkal-Babu *et al.* because they do not normalize their consecutive acceleration values by the time step 0.1 s in the jerk computations.

3.6 Related Work

State of the art in testing advanced DAS is discussed by Stellet *et al.* [160]. They categorize the main principles for a testing framework as (i) the derivation of test criteria and metrics, (ii) establishing a reference system as the ground truth information and (iii) generation of test scenarios. The use of a cost function in our approach falls into the category (i) per their taxonomy. Simulation environment itself can be considered as the category (ii), and the optimization engine for S-TALiRO that is used to create test trajectories can be considered as the category (iii) based on the discussions in that work. The cost functions we propose can be candidates to quantitatively measure the safety of autonomous vehicles against collisions. S-TALiRO provides methods for automatic, high throughput testing of fully autonomous vehicles inside simulations that can cover complex real-world traffic situations.

A vehicle in the loop (VIL) test setup is presented by Bock *et al.* [25]. They discuss the advantages of using simulators for testing DAS. Their approach is based on having a human driver in a simulator and using conventional methods for generation of test cases.

Althoff *et al.* [13] propose an online formal verification approach for autonomous vehicles. The approach proposed in that work is based on reachability analysis for the ego vehicle, *i.e.*, the vehicle under control, and other participants on the road. They compute the reachable sets and occupancy of the ego vehicle for reference trajectories and claim that the reference trajectory is safe if the occupancy of the ego vehicle does

not exit drivable area nor intersect with the other participants on the road. They assume that other participants obey the traffic rules. However, in [13], they are only considering verification of planned trajectories and executing the trajectories that are deduced to be safe. In contrast, the test results with our approach give an idea on how the mistakes of other vehicles in the traffic are handled by the ego vehicle which is a valid concern for traffic environments consisting of both autonomous and non-autonomous vehicles.

Our approach is complementary to the VIL testing, online verification and testing with real vehicles [181], as it suggests important and challenging test-cases for existing methods. The work by Winn and Julius [180] is one of the first approaches to apply functional gradient descent in order to improve the optimality of human-generated trajectories. That work was later extended in by Abbas *et al.* [8] to the problem of falsifying temporal logic requirements. In their extension, the most critical set constraints in the requirement are identified, and an optimal control is computed in order to get closer to these sets and falsify the requirement. In this problem, the resulting objective functions are minimum (non-additive) functions which are similar to the one considered by Melikyan *et al.* [122] for minimum-distance optimal control problems.

Our approach to mixing low- and high-fidelity models to reduce the computational burden of simulation optimization is, in spirit, similar to other multi-fidelity optimization techniques used in the engineering design optimization literature. For example, Xu *et al.* [183] introduce an Ordinal Transformation (OT) method that uses a low-fidelity model to rank regions of parameter space in order to prioritize the search through a high-fidelity model on a computational budget. This OT method has been combined with optimal sampling methods to further improve stochastic

search in a computationally costly, high-fidelity model [184], and improvements to these approaches continue to be developed [113]. The OT step of these approaches is similar to our method of using a low-fidelity model to generate a gradient direction; however, our approach is focused on the different problem of generation of optimal trajectories over dynamical systems.

3.7 Review and Discussion on Optimization Methods

In this chapter, we presented three different approaches to the automatic test case generation process for automated driving systems.

In Section 3.3, we presented our approach to the automatic generation of test cases in an open traffic environment for which we cannot compute gradients for the overall system that consists of multiple vehicles with complex controllers. In many cases, a complete algebraic representation of the system dynamics is not available. This creates the need for treating the system as blackbox and applying optimization methods that do not require gradient information. In this approach, we utilize a stochastic optimization method, Simulated Annealing (SA) [105] which has been successfully applied to the falsification problem of cyber-physical systems many times [132, 60, 4, 5, 23]. Simulated annealing comes with guarantees on the convergence to the global optimum, given enough time [1, 3]. However, in practice, the time budget on the testing is limited and we cannot provide guarantees on achieving the global minimum. We have picked SA as the optimization method because of its track record and availability in the falsification toolbox S-TALIRO [60]. However, our approach allows utilization of other optimization methods that do not require system gradients. Different optimization methods perform better on different problems depending on

the algorithm parameters and the (unknown) shape of the cost landscape. A survey of optimization methods that do not require gradient information can be found in [147]. Below is a non-exhaustive list of commonly used optimization methods that can be utilized in our approach.

Cross-Entropy optimization is a method that targets problems with both combinatorial and continuous properties [150]. It has been successfully used for the falsification of hybrid systems [154]. Another optimization method that has a proven track of success in the falsification of CPS is the Nelder-Mead method which is a direct search method that uses the concept of simplex for nonlinear optimization problems [130, 45, 97]. Ant Colony Optimization (ACO) [47, 61] is a population-based probabilistic approach which also has found applications in the falsification domain [17]. Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is an evolutionary algorithm which samples populations with multivariate Gaussian distribution [79, 91]. Its applicability on the falsification domain is studied in [10]. Particle Swarm Optimization (PSO) [100] is a population-based metaheuristic approach which has been applied to the temporal logic specification mining problems [76].

Besides the optimization methods mentioned above, there are new optimization methods that are promising but have not yet been applied to the falsification problem, to the best of our knowledge. An example of these algorithms is Mesh Adaptive Direct Search algorithm (MADS) which extends generalized pattern search by local exploration [22]. It searches over a conceptual *mesh* that is defined using the set of points for which the objective function had been evaluated. It can handle nonlinear constraints with convergence guarantees to local minima [9]. These properties make MADS a promising candidate for falsification problems. However, to the best of our knowledge, there is no literature on the application of MADS to the falsification

problem for CPS. Another optimization method that is promising, yet not studied in the falsification context is NM-PSO, which merges the Nelder-Mead and Particle Swarm Optimization methods [67, 100]. The power of NM-PSO comes from the fact that it combines the global search capabilities of the particle swarm optimization with the fast local convergence properties of the Nelder-Mead algorithm [147].

In Section 3.4, we utilized a random exploration method as an alternative to the optimization-based method that we have presented in Section 3.3. RRTs have been first developed for robot path/motion planning problems [110, 93, 99]. However, thanks to their ability to efficiently search over high-dimensional spaces, RRT-based approaches also deliver promising results in the test generation domain [58, 103, 26, 40, 140, 49]. In our approach, we have adopted notions from transition-based RRT [93] and RRT* [99] methods. Our RRT-based approach which is described in Section 3.4 delivered more promising results compared to our stochastic optimization-based approach which is described in Section 3.3 for the problems which contain many local minima that should be avoided for reaching the global minimum. There are many studies in the literature that aim to improve RRT and RRT*. Studying their applicability to our problem is worthwhile as they have the potential to improve our approach. For instance, the Hierarchical Rejection Sampling (HRS) approach that was proposed in [109] aims to improve the computation time which may be suffering from the bottleneck created by the sample rejection, especially in high-dimensional spaces. The HRS method performs informed sampling, *i.e.*, sampling from subsets of the search space that can potentially improve the solution. Furthermore, it performs partial sampling and hierarchically combines partial samples into larger samples so that only the partial information need to be resampled when a partial sample is rejected. Experimental

results show that applying HRS on RRT* can improve the computation time required to reach optimal solutions [109].

Finally, in Section 3.5, we presented our approach for utilizing a gradient-descent based approach for systems that we can either compute the gradients or that we can create a simpler but representative-enough version of the system which enables us to estimate the gradients. We have utilized the steepest descent method, which has been shown to be helpful in reducing the necessary number of simulations needed for falsification of systems for which we can compute or approximate the gradient information [180, 8, 185]. Although the steepest descent approach guarantees minimization of the cost with the selection of correct parameters such as step size, depending on the shape of the cost surface, correct selection of the parameters may become very challenging. Furthermore, the steepest descent approach can waste a significant amount of time by taking steps that are not in the direction toward a local minimum [162]. This issue is briefly illustrated in Fig. 28 in which the black ellipsoids represent the level set of the cost function $y = 10x_1^2 + x_2^2$, the blue star is the initial point, and the red dots are the steps taken toward the optimal solution. A survey of gradient-based optimization methods that are developed to address the issues in the steepest descent method can be found in [151]. Here, we list some of those approaches that can potentially decrease the computation time of our approach.

Momentum method aims to improve the convergence of gradient descent by damping the oscillations through the utilization of the previous descent vector multiplied by an adaptive momentum term [145, 151]. Nesterov Accelerated Gradient (NAG) method further improves the convergence by approximating the future positions and by shortening the step size before reaching a hill with a slope up [131, 151]. Adagrad [53] is another algorithm proposed for improving the convergence of gradient descent.

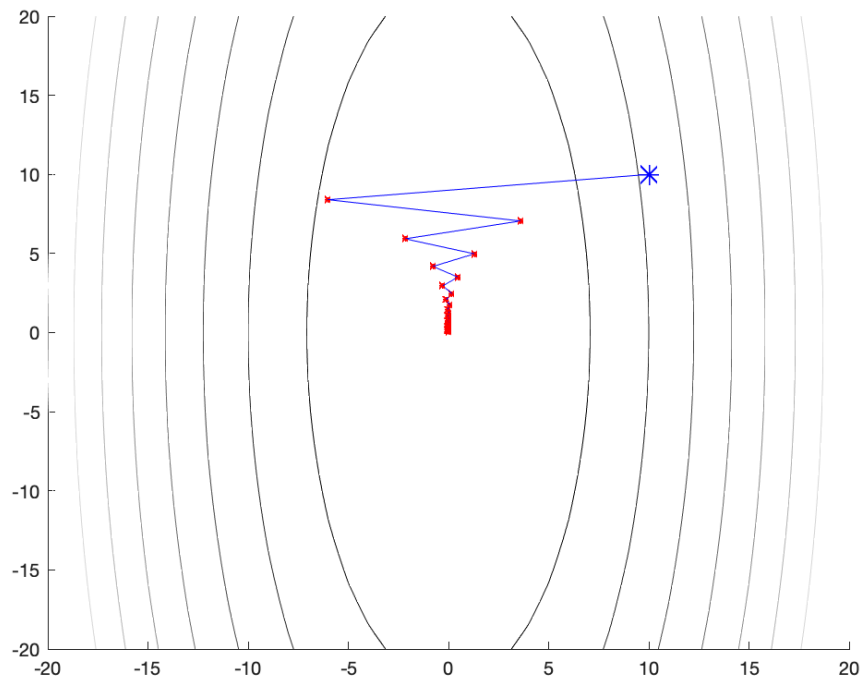


Figure 28. An illustration of slow convergence of the steepest descent algorithm.

It adapts the learning rate to the parameters and, in practice, also eliminates the need for manually setting a learning rate [151]. Adadelata [186] and an independently developed identical approach RMSprop [85] extend Adagrad to address an issue, which is the diminishing learning rate, by limiting the size of accumulated past gradients which are used in the learning rate adaptation [151]. Adaptive Moment Estimation (Adam), which adds bias correction and momentum to RMSprop, performs marginally better than RMSprop [104, 151]. Nesterov-accelerated Adaptive Moment Estimation (Nadam) modifies the momentum term of Adam and utilizes the approach used in Nesterov accelerated gradient [48, 151].

The approaches discussed above, especially Adagrad, Adadelata, RMSprop, and Adam are very similar approaches that may slightly outperform each other depending on the specifics of the problem in hand [151]. Most of those approaches have the

potential to significantly improve the execution time of our gradient-descent based approach, and we plan to experiment with them in our future work.

3.8 Conclusions and Future Work

In this chapter, we presented three different approaches, a falsification-based approach that utilizes global optimizers, an RRT-based approach that utilizes rapidly-exploring random trees, and a functional gradient-descent approach that utilizes simplified system dynamics, for generating maneuvers of other road occupants with the target of minimizing cost functions that are defined on the safety or performance of automated driving systems. Our experimental results suggest that each of these approaches have different advantages over each other under different conditions and that they can be used complementary to each other.

Falsification-based Approach

We proposed an approach for automatic simulation-based testing of autonomous vehicle controllers that is guided by a cost function. We believe that using optimization techniques for test guidance is a promising approach for testing cyber-physical systems in general [96].

Future work is to extend the capabilities of our framework by extracting the conditions leading to unsafe behavior from the simulations and use them for training a model for estimating the probability of future collisions.

The trajectory generation for agents in our framework is based on the boundaries described by the user. The generated trajectories are then tracked by the user-supplied

controllers. Nagy *et al.* [126] propose a method for generating trajectories for mobile robots that can be easily tracked by a real vehicle. Although the final trajectories followed by the controllers are realistic in our framework, utilizing the approach described in that work can allow using the trajectories directly without the need for a controller to track them. Another future work is to incorporate such a method for trajectory generation.

RRT-based Approach

We proposed an approach that explores maneuvers for road occupants using rapidly-exploring random trees with the target of minimizing safety/performance cost functions defined for the automated driving system under test. Our experimental analysis suggests that the RRT-based approach can avoid local minima that can be challenging for the falsification-based approach.

When formulating the agent trajectory generation problem as an optimization problem, the trajectories are presented with a finite number of parameters, *i.e.*, waypoint parameters for a fixed number of waypoints, which are provided by the test designer. One of the advantages of utilizing the RRT-based exploration is the ability to abandon the finite parameterization of the trajectories. This creates the opportunity to more freely explore the space, and to minimize the need for manually designing the general structure of the trajectory shapes. In an optimization-based approach, choosing a small number of parameters would limit the flexibility in generating critical trajectories, while choosing a large number would increase the dimensionality of the search space. For instance, in our Case Study 2, the number of parameters that the optimization method would need to create the minimum-cost trajectories that are

discovered by the RRT-based approach varies from 8 to 68 based on the number of parameters at each waypoint and the number of waypoints that are used to create these trajectories.

Future work on our RRT-based approach will include:

- Exploring new methods for computing the novelty of a trajectory instead of a single state
- Computing a cost on the shape of the trajectory instead of computing the minimum of the instantaneous costs of the points on the trajectory. This may especially be useful for rewarding some types of vehicle paths such as the ones which are closer to real-world driving behaviors.
- Exploring new methods for using multi-objective optimization can be studied for different objectives like time-to-collision, collision speed, collision impact area etc.
- Combining RRT-based approach with the falsification-based approach such that the exploration starts with RRT-based approach and falsification-based approach further minimizes the cost starting from the best cases discovered by the RRT-based approach.

Functional Gradient Descent Approach

For generating optimal test cases, we presented a functional gradient descent approach that uses simplistic yet analytically tractable dynamics to efficiently search more realistic dynamics of a complex system. As a case study, we implemented a full-range adaptive cruise control from the literature and utilized our approach for automatically generating test cases for exploring the worst-case performance of

the controller within the boundaries of a target scenario. The results of our case study show that simple, yet sufficiently realistic, dynamical equations for the system under test can be used with a functional gradient descent method to search over more realistic complex dynamics of the actual system. The results of our method were better than those from Simulated Annealing, a proven metaheuristic for global optimization. However, we are not claiming that our approach is superior to SA optimization for every application. Rather, our case study shows that when simplified dynamics of a system are available, satisfactory optimization results can be obtained without the process of tuning SA parameters. Furthermore, our study shows that automatic (search-based) test-case generation can create more challenging test cases than the human-generated ones that are traditionally used in the literature, and these test cases will provide better insights into the performance of controllers under test.

The work by Abbas *et al.* [6] defines a conformance metric for measuring the closeness of systems and methods to approximate this conformance. A formal analysis of closeness of the simple dynamics to the real system dynamics can be utilized to develop an approach for a better-guided descent computation. However, in many cases, we cannot have formal notions of distance between models as for example in a model-order reduction [155] or approximate bisimulations [71].

Another promising research direction is to combine our optimal trajectory generation approach with multi-fidelity optimization methods like those from Xu *et al.* [183, 184] and others [113, 31, 89]. In our method, the low-fidelity model is used to prioritize descent directions in a functional gradient descent. An alternative approach, for instance, is to mix high and low fidelity models to choose the best functional gradient descent directions, and existing multi-fidelity optimization methods may be well suited for this task.

REQUIREMENTS-DRIVEN TESTING OF AUTONOMOUS VEHICLES WITH MACHINE LEARNING COMPONENTS

4.1 Introduction

Automated driving system designs often use Machine Learning (ML) components such as Deep Neural Networks (DNNs) to classify objects within CCD images and to determine their positions relative to the vehicle, a process known as *object detection and classification* [69]. Other designs use Neural Networks (NNs) to perform *end-to-end* control of the vehicle, meaning that the NN takes in the image data and outputs actuator commands, without explicitly performing an intermediate object detection step [144, 30, 32]. Some other approaches use end-to-end learning to do intermediate decisions like risk assessment [161].

No universally agreed upon testing or verification methods have yet arisen for automated driving systems. One reason for this is that the ML components and DNNs are notoriously difficult to test and verify. We present a framework for Simulation-based Adversarial Testing of Autonomous Vehicles (Sim-ATAV), which can be used to check the closed-loop properties of automated driving systems that include ML components. We describe a testing methodology, based on a test case generation method, called covering arrays, and requirement falsification methods to automatically identify problematic test scenarios. The resulting framework can be used to increase the reliability of automated driving systems.

ML system components are problematic from an analysis perspective, as it is

difficult or impossible to characterize all of the behaviors of these components under all circumstances. One reason is that the complexity of these systems can be very high in terms of the number of parameters. For example, AlexNet [107], a pre-trained DNN that is used for classification of CCD images, has 60 million parameters. Another reason for the difficulty in characterizing behaviors of ML components is that the parameters are learned based on training data. In other words, characterizing ML behaviors is, in some ways, as difficult as the task of characterizing the training data. Again using the AlexNet example, the number of training images used was 1.2 million. While a strength of DNNs is their ability to generalize from training data; the challenge for analysis is that we do not understand well how they generalize for all possible cases.

There has been significant interest recently on verification and testing for ML components (see Section 4.6). For example, adversarial testing approaches seek to identify perturbations in image data that result in misclassifications. By contrast, our work focuses on methods to determine perturbations in the configuration of a testing scenario, meaning that we seek to find scenarios that lead to unexpected behaviors, such as misclassifications and ultimately vehicle collisions. The framework that we present allows this type of testing in a virtual environment. By utilizing advanced 3D models and image rendering tools, such as the ones used in game engines or film studios, the gap between testing in a virtual environment and the real world can be minimized.

Most of the previous work to test and verify systems with ML components focuses only on the ML components themselves, without consideration of the closed-loop behavior of the system. For autonomous driving applications, we remark that the

ultimate goal is to evaluate the closed-loop system performance, and hence, any testing methods used to evaluate such systems should support this goal.

The closed-loop nature of a typical automated driving system can be described as follows. A *perception* system processes data gathered from various sensing devices, such as cameras, LIDAR, and radar. The output of the perception system is an estimation of the principal (*ego*) vehicle’s position with respect to external obstacles (e.g., other vehicles, called *agent* vehicles, and pedestrians). A *path planning* algorithm uses the output of the perception system to produce a short-term plan for how the ego vehicle should behave. A *tracking controller* then takes the output of the path planner and produces actuation outputs, such as accelerator, braking, and steering commands. The actuation commands affect the vehicle’s interaction with the environment. The iterative process of sensing, processing, and actuating is what we refer to as closed-loop behavior.

The contributions of the work presented in this chapter can be summarized as follows.

- We provide an open-source autonomous vehicle testing framework [167].
- We provide a new algorithm to perform falsification of formal requirements for an autonomous vehicle in a closed-loop with the perception system, which includes an efficient means of searching over discrete and continuous parameter spaces.
- The method represents a new way to do adversarial testing in scenario *configuration space*, as opposed to the usual method, which considers adversaries in *image space*. Additionally, we demonstrated a new way to characterize problems with perception systems in *configuration space*.

- We extend the software testing theory of covering arrays to closed-loop Cyber-Physical System (CPS) applications that have embedded ML algorithms.
- We add models of LIDAR and radar sensors and include sensor fusion algorithms, and we demonstrate how the requirements-based testing framework we propose can be used to automate the search for specific types of fault cases involving sensor interactions.
- We provide requirements for both component-level and system-level behaviors, and we show how to automate the identification of behaviors where component-level failures lead to system-level failures. An example of the kind of analysis this allows is automatically finding cases where a sensor failure leads to a collision case.
- We include a model of agent *visibility* to various sensors and include this notion in the requirements that we consider. This provides a way to reason about how the system should behave, based on whether agents are or are not visible, including the ability to reason about the temporal aspects of agent visibility. For example, we can use this feature to test the requirement that within 1 second after an agent becomes visible to the LIDAR sensor, the perception system should correctly classify the agent. This allows us to automate the search for behaviors related to temporal aspects of sensor behaviors in the context of a realistic driving scenario.
- We demonstrate the ability to falsify properties by adversarially searching over agent trajectories. This permits the use of our requirements-driven search-based approach over a broad class of agent behaviors, which allows us to automatically identify corner cases that are difficult to find using traditional simulation-based techniques.

4.2 Preliminaries

This section presents the setting used to describe the testing procedures performed with our framework. The purpose of our framework is to provide a mechanism to test, evaluate, and improve on an automated driving system design. To do this, we use a simulation environment that incorporates models of a vehicle (called the *ego* vehicle), a perception system, which is used to estimate the state of the vehicle with respect to other objects in its environment, a controller, which makes decisions about how the vehicle will behave, and the environment in which the ego vehicle is deployed. The environment model contains representations of a wide variety of objects that can interact with the ego vehicle, including roads, buildings, pedestrians, and other vehicles (called *agent* vehicles). The behaviors of the system are determined by the evolution of the model states over time, which we compute using a *simulator*. Formally, the framework implements a system model \mathcal{M} which is defined in Section 2.1.

4.2.1 Robustness-Guided Model Checking (RGMC)

The goal of a *model checking* algorithm is to ensure that all traces satisfy the requirement. The robustness metric can be viewed as a fitness function that indicates the degree to which individual executions of the system satisfy the requirement φ , with positive values indicating that the execution satisfies φ . Therefore, for a given model \mathcal{M} of a system and a given requirement φ , the model checking problem is to ensure that for all $\mu \in \mathcal{L}(\mathcal{M})$, $\llbracket \varphi \rrbracket_{\mathbf{d}}(\mu) > 0$.

Let φ be a given STL property that the system is expected to satisfy. The robustness metric $\llbracket \varphi \rrbracket_{\mathbf{d}}$ maps each simulation trace μ to a real number r . Ideally, for

the STL verification problem, we would like to prove that $\inf_{\mu \in \mathcal{L}(\mathcal{M})} \llbracket \varphi \rrbracket_{\mathbf{d}}(\mu) > \varepsilon > 0$ where ε is a desired robustness threshold.

4.2.2 Falsification and Critical System Behaviors

In this chapter, we focus on the task of identifying critical system behaviors, including falsifying traces. To identify falsifying system behaviors, we leverage existing work on *falsification*, which is the process of identifying system traces μ that do not satisfy φ . For the STL falsification problem, falsification attempts to solve the problem: Find $\mu \in \mathcal{L}(\mathcal{M})$ s.t. $\llbracket \varphi \rrbracket_{\mathbf{d}}(\mu) < 0$. This is done using best effort solutions to the following optimization problem:

$$\mu^* = \arg \min_{\mu \in \mathcal{L}(\mathcal{M})} \llbracket \varphi \rrbracket_{\mathbf{d}}(\mu). \quad (4.1)$$

If $\llbracket \varphi \rrbracket_{\mathbf{d}}(\mu^*) < 0$, then a counterexample (adversarial sample) has been identified, which can be used for debugging or for training. In order to solve this non-linear non-convex optimization problem, a number of stochastic search optimization methods can be applied (e.g., [4] – for an overview see [87, 97]). We leverage existing falsification methods to identify falsifying traces of the automated driving system.

4.2.3 Covering Arrays

In software systems, there can often be a large number of discrete input parameters that affect the execution path of a program and its outputs. The possible combinations of input values can grow exponentially with the number of parameters. Hence, exhaustive testing on the input space becomes impractical for fairly large systems. A fault in such a system with k parameters may be caused by a specific combination of

t parameters, where $1 \leq t \leq k$. One best-effort approach to testing is to make sure that all combinations of any t -sized subset (i.e., all t -way combinations) of the inputs are tested.

A *covering array* is a minimal number of test cases such that any t -way combination of test parameters exist in the list [80]. Covering arrays are generated using optimization-based algorithms with the goal of minimizing the number of test cases. We denote a t -way covering array on k parameters by $CA(t, k, (v_1, \dots, v_k))$, where v_i is the number of possible values for the i^{th} parameter. The size of the covering array increases with increasing t , and it becomes an exhaustive list of all combinations when $t = k$. Here, t is considered as the *strength* of the covering array. In practice, t can be chosen such that the generated tests fit into the testing budget. Empirical studies on real-world examples show that more than 90 percent of the software failures can be found by testing 2 to 4-way combinations of inputs [108].

Despite the t -way combinatorial coverage guaranteed by covering arrays, a fault in the system possibly may arise as a result of a combination of a number of parameters larger than t . Hence, covering arrays are typically used to supplement additional testing techniques, like uniform random testing. We consider that because of the nature of the training data or the network structure, NN-based object detection algorithms may be sensitive to a certain combination of properties of the objects in the scene. Figure 29 shows outputs of a DNN-based object detection and classification algorithm for 4 different combinations of vehicle type, vehicle color, and pedestrian pants color while all other parameters like position and orientation of the objects are the same. In a comparison between configurations (a) and (b), the vehicle type does not change but the colors of the vehicle and pedestrian pants change from blue to white. While both the car and the pedestrian are detected in configuration (a), the

pedestrian is detected but the car is not detected in configuration (b); however, in a comparison between configurations (b) and (d), if we fix the colors of the vehicle and pedestrian pants to be white but change the vehicle type, then the car is detected but the pedestrian is not detected. We can also see that the size of the detection box is different between configurations (c) and (d), for which the vehicle type is the same but the colors of the vehicle and pedestrian pants are different. Our observation is that the characterization of the errors is generally not as simple as saying that all white colored cars are not detected. Instead, the errors arise from some combination of subsets of discrete parameters. Because of this combinatorial aspect of the problem, covering arrays may be a good fit to test DNN-based object detection and classification algorithms. In Section 4.4, we describe how Sim-ATAV combines covering arrays to explore discrete and discretized parameters with falsification on continuous parameters.

4.3 Requirements

In this section, we provide five STL requirements intended for the automated driving system. Each requirement is used to target specific aspects of safety and performance. Also, we describe how analysis results related to each of the requirements can be used to enhance either the controller design or testing phases of the development process.

4.3.1 STL Requirements

The following describes each of the requirements that we use in the sequel to evaluate the automated driving system design with our virtual framework. We provide





		Color Combinations	
		Blue car, blue pants	White car, white pants
Vehicle Type	A	 (a)	 (b)
	B	 (c)	 (d)

Figure 29. Specific configurations impacting DNN performance.

these requirements to illustrate how STL can be used to describe four different types of behavior expectations for an automated driving system: *system-level* safety, *subsystem-level* performance, *subsystem-to-system* safety, and system-level performance (driving comfort) requirements.

Requirement $\varphi 1$: Vehicle should not collide with an object.

This requirement is an example of a *system-level safety requirement*. It is used to ensure that the ego vehicle does not collide with any object in the environment. Behaviors that do not satisfy this requirement correspond to an unsafe performance from the autonomous vehicle. These cases are valuable to identify in simulation, as they can be communicated back to the control designers so that the control algorithms can be improved.

The following provides the STL requirement.

$$\varphi 1_i = \Box(\neg\pi_{i,coll})$$

where

$$\pi_{i,coll} = dist(i, ego) < \epsilon_{dist}$$

In the above specification, i corresponds to an object in the environment, such as an agent vehicle or a pedestrian. $dist(i, ego)$ gives the minimum Euclidean distance between the boundaries of the Ego vehicle and the boundaries of object i . The specification basically indicates that the Ego vehicle should not collide with object i .

In practice, we consider a unique requirement for each object in the environment. When the object we are considering is clear from the context, we drop the index i and refer to the requirement $\varphi 1$.

Requirement $\varphi 2$: Sensor should detect visible obstacles.

This requirement is an example of a *subsystem-level requirement*; this particular example can be considered as a requirement on the sensor or perception subsystems.

The requirement indicates that the perception system or a specific sensor should not fail to detect an object for an excessive amount of time.

The requirement is as follows.

$$\varphi_{2,i,s} = \Box((W(i,s) \wedge \neg D(i,s)) \implies \Diamond_{[0,t1]}(D(i,s) \vee \neg W(i,s)))$$

Here, we use $W(i,s)$ to mean that object i is physically *visible* to sensor s . For our framework, $s \in \{CCD, LIDAR, radar, combined\}$, where *combined* represents the total perception system, which is a fusion of available sensors. Function $D(i,s)$ evaluates to true when sensor s detects object i .

A description of this requirement in natural language could be “it is always true that for any time when object i is visible and not detected by sensor s , there exists an instant, within 0 to $t1$ seconds, that object i is either detected or invisible to the sensor”.

When the object i and sensor s are clear from the context, we drop the indices and refer to the requirement φ_2 .

Requirement φ_3 : Localization error should not be too high for too long.

This requirement is another sensor-level requirement and specifies that the localization of an object that is based on a particular sensor should provide sufficient accuracy, within an adequate time after the object becomes visible to the sensor.

The following is the requirement.

$$\begin{aligned} \varphi_{3,i,s} = & \Box((W(i,s) \wedge (\neg D(i,s) \vee E(i,s) > \epsilon_{err})) \\ & \implies \Diamond_{[0,t1]}(\neg W(i,s) \vee (D(i,s) \wedge E(i,s) < \epsilon_{err}))) \end{aligned}$$

In the requirement, $E(i, s)$ is the difference between object i 's location and its location as estimated using information from sensor s . Constant ϵ_{err} is a threshold on the acceptable amount of error between the actual position of i and its estimated position.

To understand the requirement, consider the situation where either an object is not detected (i.e., $\neg D(i, s)$) or there is a large error in the localization of the object (i.e., $E(i, s) > \epsilon_{err}$), and call this a case of “*poor detection*” of the object. Then we can read the requirement as follows: “it is always true that whenever object i is visible to sensor s and is poorly detected by sensor s , there exists an instant, within a time period of 0 to $t1$ seconds, that either object i is invisible to sensor s or the object is detected and the localization error is small, as computed using information from sensor s ”.

This requirement basically limits the amount of time the sensor error can be greater than a given threshold. When the object i and sensor s are clear from the context, we drop the indices and refer to the requirement $\varphi3$.

Requirement $\varphi4$: A sensor-related fault should not lead to a system-level fault.

This is an example of a *subsystem-to-system requirement*. This requirement relates sensor-level behaviors to system-level behaviors. The purpose is to isolate behaviors where a sensor fault results in a collision. The expectation is that the system as a whole should be robust to the failure of a single sensor.

The requirement follows.

$$\varphi4_{i,s} = \Box \neg \left(\Box_{[0,t1]} (\neg \pi_{i,coll} \wedge W(i, s) \wedge (\neg D(i, s) \vee E(i, s) > \epsilon_{err})) \wedge \Diamond_{(t1,t2]} \pi_{i,coll} \right)$$

The above requirement designates that there should not be a period of $t1$ seconds where a visible object is not accurately detected and no collision occurs, followed

immediately by a period of length $t2 - t1$ seconds that contains a collision. In other words, the requirement indicates that a system level fault (collision) should not occur within a short time after a sensor fault. A behavior that violates this requirement does not necessarily indicate that the sensor fault *caused* the system fault, but it suggests a correlation, as it points to a behavior wherein the system fault occurs a short time after the sensor fault. Providing behavior examples that violate this requirement can help to pinpoint the cause of system-level faults.

When the object i and sensor s are clear from the context, we drop the indices and refer to the requirement $\varphi4$.

Requirement $\varphi5$: The vehicle should not do excessive braking unnecessarily or too often.

This is a *system-level performance (driving comfort)* requirement, in that it requires that the system not brake unnecessarily or too often, thereby causing discomfort for the passengers.

The requirement follows.

$$\varphi5 = \Box \left(\neg \Box_{[0,t1]} (B \wedge \neg C) \wedge \neg (edge \wedge \Diamond_{(0,t2]} (edge \wedge \Diamond_{(0,t2]} edge)) \right),$$

where

$$edge = B \wedge \bigcirc \neg B.$$

Here, C is a variable that is true when the Ego vehicle is estimated to collide with another object in the environment, based on a simplified model of future behaviors. The simplified model that we use for future trajectory estimation is the Constant Turn Rate and Velocity (CTRV) model [156]. B represents that the amount of braking

force applied by the controller exceeds half of the available braking force. The variable *edge* represents the event of a *true* value of *B* followed by a *false* value in the next time step.

To understand the meaning of requirement $\varphi 5$, consider the following part of the requirement:

$$\Box\left(\neg\Box_{[0,t1]}(B \wedge \neg C)\right),$$

which requires that the system not apply excessive braking for more than a specific amount of time (*t1*) while there is no collision predicted. This essentially stipulates that the system should not unnecessarily brake for a prolonged amount of time. Next, consider the second part of requirement $\varphi 5$:

$$\Box\left(\neg(edge \wedge \Diamond_{(0,t2]}(edge \wedge \Diamond_{(0,t2]}edge))\right),$$

which indicates that there should not be an “on-off” behavior, followed by another “on-off” behavior, followed by a third “on-off” behavior, with less than *t2* seconds between each other. This essentially requires that the brakes not be applied and released too often. Thus, this is a riding comfort requirement.

4.3.2 Development Process Support

We describe how requirements $\varphi 1$ through $\varphi 5$ can be used to support both the controller design and testing phases of the development process.

For all of the requirements, any detected violation (falsification) should be linked back to the conditions that caused the violation.

Consider the first scenario’s requirement, $\varphi 1$, “Vehicle should not collide with an object”: if the vehicle does collide with an object, then we would go back and see what conditions caused such an event, for example, whether the vehicle speed

trace exhibited an anomaly or whether the vehicle was moving erratically. Testing for collision avoidance is well established in the field of ADAS. Often inflatable and other destructible targets are employed for convenience; for example, see [111] and Fig. 30.



Figure 30. Robotic pedestrian surrogate target with a Toyota autonomous vehicle.

In the requirement “Sensor should detect visible obstacles” we focus on the detection of an obstacle as operational imperative. If the sensor fails to detect within a time interval, then the requirement will be violated. This is essentially the sensor-level requirement (visible but not detected), and test engineers can set a real-world experiment to verify it relatively easily because it is decoupled from others (one-term inequality, sensor by sensor).

The requirement “Localization error should not be too high for too long” is important to verify (falsify) for both ego-location and identification of positions of other agents in the environment. Placing an ego-vehicle in the correct pose on the road is usually not achieved by simply relying on GPS signal processing, due to the GPS tendency to “jump” unpredictably, but instead by estimating and dynamically refining the pose through landmark observations, such as road edges, vertical elements such as light poles, and signs. Assuming that the ego-vehicle localization is done with sufficient accuracy, the remaining task of localizing is to make sure that the

location of other agents, especially those in the planned path of the ego vehicle, are estimated with sufficient accuracy. Often a grid-based representation centered on the ego-vehicle is employed (e.g., [138]). Estimating $E(i, s)$ in $\varphi 3$ is not trivial, but practical approaches exist that can be used by test engineers (e.g., [73]).

The requirement “A sensor-related fault should not lead to a system-level fault” is a form of robustness requirement. This is similar to a requirement that the system should have no “single point of failure”, which enforces that the failure of any single component will not cause the system to fail (for example, see [92]). We make an important clarification that is practical but limiting in scope: that no failure should occur within the specified (short) time after the fault. Test engineers could readily use examples of behavior provided in the course of falsifying this requirement.

Lastly, the requirement “The vehicle should not apply brakes too often” is an example of a possible set of requirements designed to establish how comfortable the ride in the vehicle is. It is known that autonomous vehicles could induce motion sickness in passengers if the vehicle control system does not comply with human physiology [56], [74]. A better requirement may well be developed using fuzzy set theory and further refined for a specific target group of passengers (e.g., elderly people). An alternative requirement could be defined by counting the number of occurrences of an event within a total time period, instead of relating one occurrence to another. Such a requirement can be defined as a Timed Propositional Temporal Logic (TPTL) specification. TPTL is a variant of temporal logic and it is also supported in our framework [43].

4.4 Framework

We describe Sim-ATAV, an open-source framework for performing testing and analysis of automated driving systems in a virtual environment. The simulation environment used in Sim-ATAV includes a vehicle perception system, a vehicle controller, and a model of the physical environment. The perception system processes data from three sensor systems: CCD camera images, LIDAR, and radar. The framework uses freely available and low-cost tools and can be run on a standard desktop PC. Later, we demonstrate how Sim-ATAV can be used to implement traditional testing approaches, as well as advanced automatic test generation approaches that were not previously possible using existing frameworks.

Figure 31 shows an overview of the simulation environment. The environment consists of a Simulator and a Vehicle Control system. The Simulator contains models of the ego vehicle, agents, and other objects in the environment (e.g., roads, buildings). The Simulator outputs sensor data to the Vehicle Control system. The sensor data includes representations of CCD camera, LIDAR, and radar data. Simple models of the sensors are used to produce the sensor data. For example, synthetic CCD camera images are rendered by the Simulation system, as if they came from a camera mounted on the front of the ego vehicle. The Vehicle Control system contains models of the Perception System, which performs sensor data processing and sensor fusion. The Controller uses the output of the Perception System to make decisions about how to actuate the AV system. Actuation commands are sent from the Controller to the Simulator.

Simulations proceed iteratively. At each instant, sensor data is processed by the Vehicle Control, which then makes an actuation decision. The actuation decision is

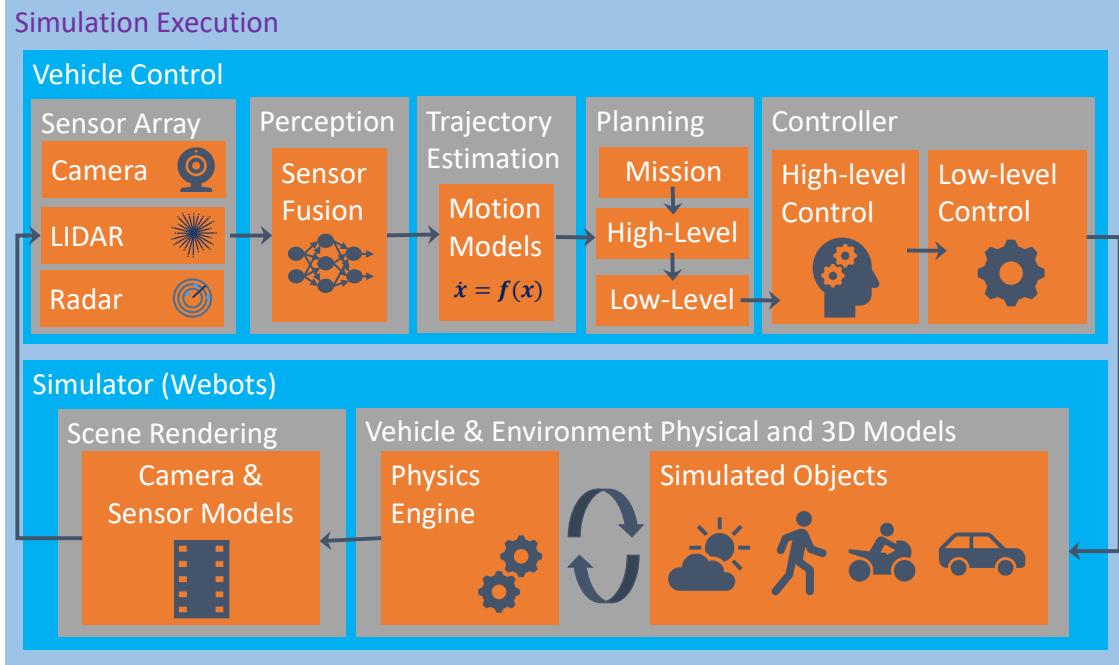


Figure 31. Overview of the simulation environment.

then transmitted back to the Simulator, which uses the actuation commands to update the physics for the next time instant. This process is repeated until a designated time limit has been reached.

The Vehicle Control system is implemented in Python. We use simplified algorithms to implement the subsystems of the vehicle control, which is sufficient in this case, as the purpose of this investigation is to evaluate new *testing* methodologies and not to evaluate a real AV control design; however, we note that it is straightforward to replace our algorithms with production versions to test real control designs.

To process CCD image data, we use a lightweight DNN, SqueezeDet, which performs object detection and classification [182]. SqueezeDet is implemented in TensorFlowTM[2], and it outputs a list of object detection boxes with corresponding class probabilities. This network was originally trained on real image data from the KITTI dataset [69] to achieve accuracy comparable to the popular AlexNet classifier

[107]. We further train this network on the virtual images generated in our framework. Figure 32 shows an example output from SqueezeDet, based on a synthetic image produced by our simulator. The image shows two vehicles correctly detected and classified, along with a portion of a shadow that is incorrectly classified as a vehicle.



Figure 32. Outputs from the SqueezeDet DNN, based on a synthesized camera image.

To process LIDAR point cloud data, we cluster the received points based on their positions and estimate existence and types of the objects based on the dimensions of the clusters. We implement a simple sensor fusion algorithm that relates and merges the object detections from a camera, a LIDAR, and a radar. It also utilizes the expected current positions of previously detected objects. The object states estimated

by the sensor fusion algorithm are used to estimate the future trajectories of the objects using the CTRV model [156].

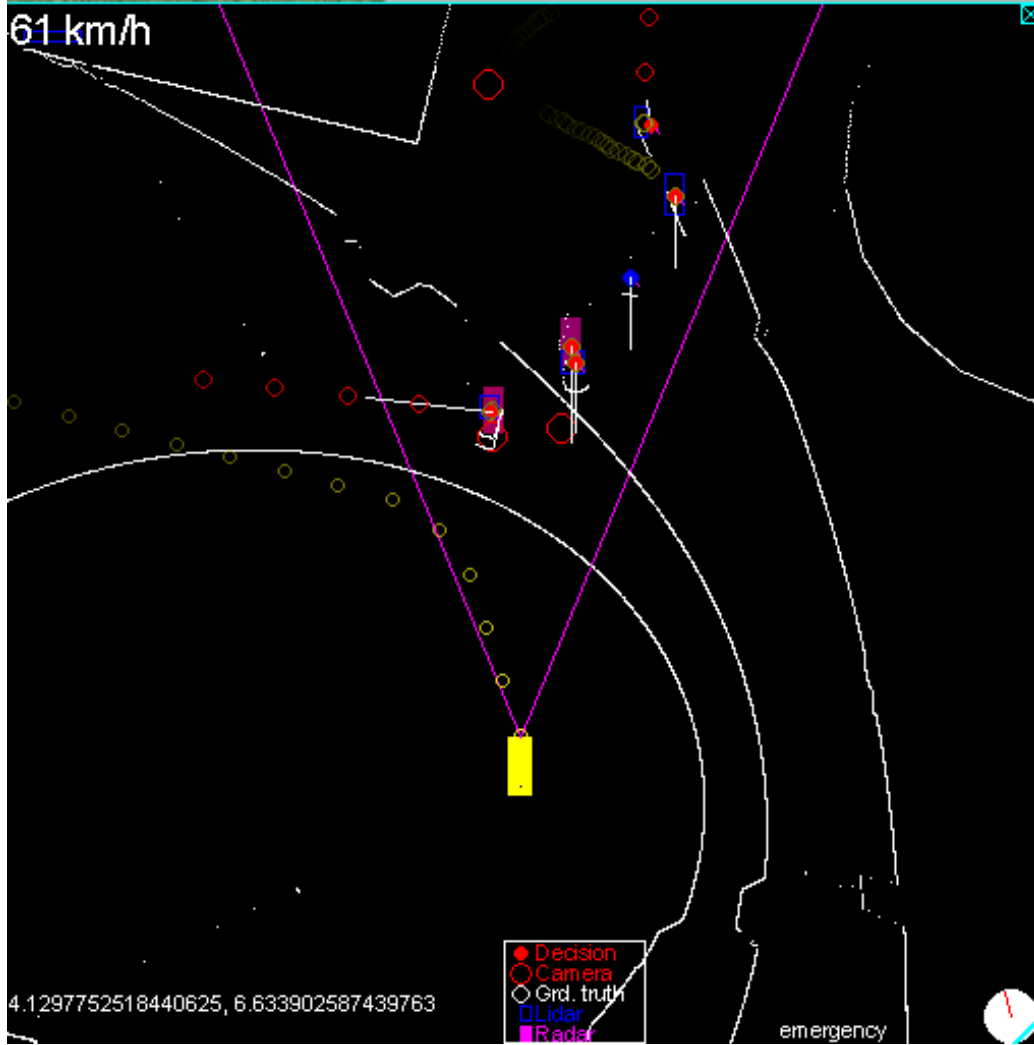


Figure 33. Sensor fusion outputs.

Figure 33 illustrates outputs from the sensor fusion system. In the figure, the solid yellow box in the middle represents the Ego vehicle. Yellow circles in front of the ego vehicle represent the estimated future trajectory of the Ego vehicle. Small white dots represent LIDAR point cloud data. The colored dots and rectangles represent

detected objects, with their estimated orientation indicated with a white line in front of them. Expected future positions of agent vehicles with respect to the ego vehicle are represented by red circles.

Our simple planner takes the high-level target path and target speed and the outputs of sensor fusion and trajectory estimation algorithms. It assigns collision risk levels to the target objects with a simple logic and outputs the risk assessments and a target speed, which depends on the target speed of the mission or other factors, such as the distance to a sharp turn ahead.

Our control algorithm implements simple path and speed tracking and *collision avoidance* features. The controller receives the outputs of the planner. When there is no collision risk, the controller drives the car with the target speed and on the target path. When a future collision with an object is predicted, it applies the brakes at a level commensurate with the risk assigned to the object.

The environment modeling framework is implemented in Webots [123], a robotic simulation framework that models the physical behavior of robotic components, such as manipulators and wheeled robots, and can be configured to model autonomous driving scenarios. In addition to modeling the physics, a graphics engine is used to produce images of the scenarios. In Sim-ATAV, the images rendered by Webots are configured to correspond to the image data captured from a virtual camera that is attached to the front of a vehicle.

The process used by Sim-ATAV for test generation and execution for discrete and discretized continuous parameters is illustrated by the flowchart shown in Fig. 34. Sim-ATAV first generates test cases that correspond to scenarios defined in the simulation environment using covering arrays as a combinatorial test generation approach. The scenario setup is communicated to the simulation interface using TCP/IP sockets.

After a simulation is executed, the corresponding simulation trace is received via socket communication and evaluated using a cost function. Among all discrete test cases, the most promising one is used as the initial test case for the falsification process shown in Fig. 35. For falsification, the result obtained from the cost function is used in an optimization setting to generate the next scenario to be simulated. For this purpose, we used S-TALIRO [60], which is a MATLAB toolbox for falsification of CPSs. Similar tools, such as Breach [46], can also be used in our framework for the same purpose.

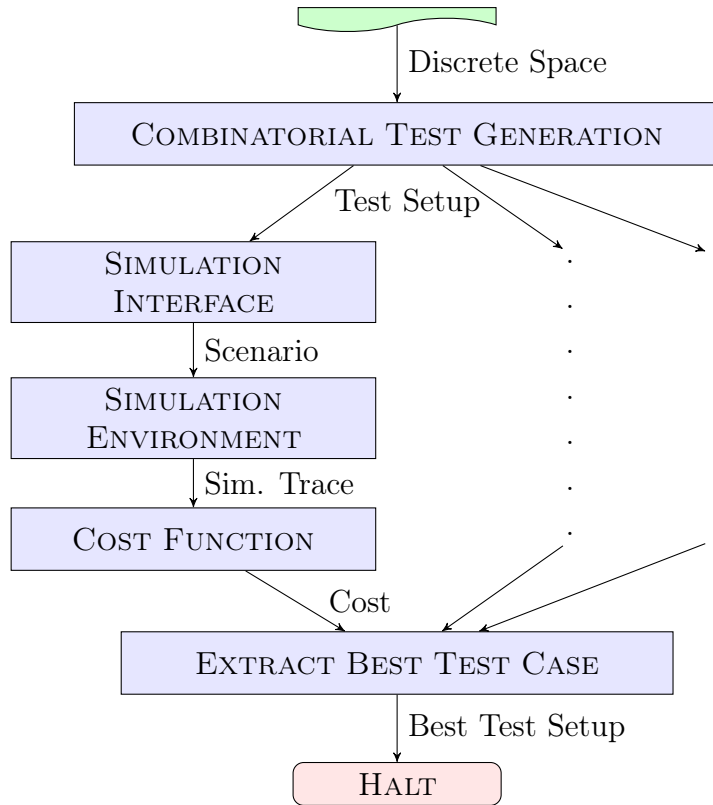


Figure 34. Flowchart illustrating the combinatorial testing.

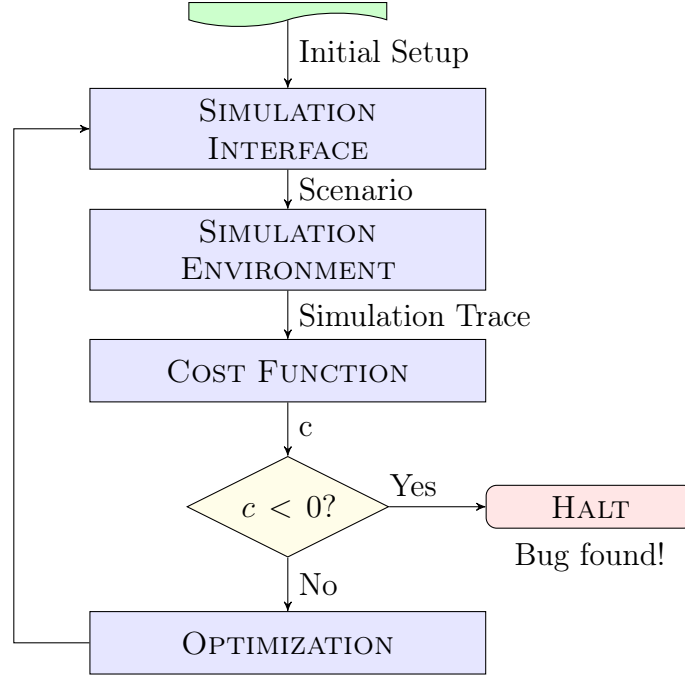


Figure 35. Flowchart illustrating the falsification.

4.5 Testing Application

In this section, we present an evaluation of the Sim-ATAV framework using three separate driving scenarios. The scenarios are selected both to be challenging for the automated driving system and also analogous to plausible driving scenarios experienced in real-world situations. In general, the system designers will need to identify crucial driving scenarios, based on intuition about challenging situations, from the perspective of the autonomous vehicle control system. A thorough simulation-based testing approach will include a wide array of scenarios that exemplify critical driving situations.

For each of the following scenarios, we consider a subset of the requirements presented in Section 4.3 and describe how to use the results to enhance the development process. We conclude the section with a summary of the results.

Scenario 1

The first scene that we consider is a straightaway section of a two-lane road, as illustrated in Fig. 36. Several cars are parked on the right-hand side of the road, and a pedestrian is jay-walking in front of one of the cars, passing in front of the Ego car from right to left. We call this driving scenario model \mathcal{M}_1 . The scenario simulates a similar setup to the Euro NCAP Vulnerable Road User (VRU) protection test protocols [164].

Several aspects of the driving scenario are parameterized, meaning that their values are fixed for any given simulation by appropriately selecting the model parameters. The parameters we use for this scenario are as follows:

- Initial speed and lateral position of the Ego vehicle inside its lane;
- Walking speed of the pedestrian;
- The model of Agent car 1, which is next to the pedestrian;
- R, G, B values for the colors of Agent car 1;
- R, G, B values for the pedestrian’s shirt and pants.

We choose the parameters such that their specific combinations could be challenging to a DNN-based pedestrian detection system that relies on CCD camera images. We also choose the ranges of some of the parameters so that the scenario is physically challenging for the brake performance.

We evaluate Model \mathcal{M}_1 against three of the requirements from Section 4.3: $\varphi1$, $\varphi2$, and $\varphi4$. These include the system-level requirement, the sensor-level requirements, and the sensor-to-system-level requirement. We use this collection of requirements for Model \mathcal{M}_1 to demonstrate how we can automatically identify each type of behavior using our framework.

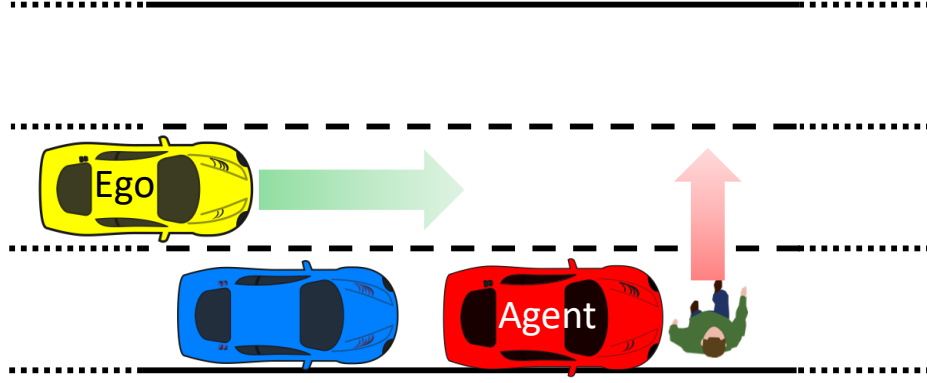


Figure 36. Overview of the scenario 1.

Scenario 2

The next scenario involves a left turn maneuver by the ego vehicle in a controlled intersection, as illustrated in Fig. 37. An agent vehicle (*Agent 1*) in the opposing lane unexpectedly passes through the intersection, against a red light, potentially causing a collision with the Ego vehicle. There is also another agent car (*Agent 2*), which is making a legal left turn from the opposing lane. It is incumbent on the Ego vehicle to take action to avoid colliding with the agent vehicles. We call the model of this scenario \mathcal{M}_2 .

For this experiment, we choose parameters such that the position of Agent 2, or trajectory followed by Agent 1, in combination with the behavior of the Ego, may result in poor performance from the sensor processing or trajectory estimation systems. The following variables are parameterized for this model:

- Ego vehicle initial speed and initial distance to the intersection;
- Agent 1 initial distance to the intersection, initial target speed, target speed when approaching the intersection, target speed inside the intersection, initial

lateral position w.r.t its lane center, target lateral position w.r.t its lane center when approaching the intersection, and target lateral position w.r.t its lane center inside the intersection;

- Agent 2 initial lateral position w.r.t its lane center, speed, and initial distance to the intersection.

We evaluate Model \mathcal{M}_2 against requirement φ_4 . The idea in using the sensor-to-system-level requirement is that it is relatively easy, in general, to find behaviors that result in a collision for Model \mathcal{M}_2 , but many collision cases are not interesting for the designers. This could be because, for example, the agent car is moving too quickly for the ego vehicle to avoid. This would be a behavior that is not necessarily caused by any specific incorrect behavior on the part of the ego vehicle. Instead, we use φ_4 to identify behaviors where there is a collision that is directly correlated to an unacceptable performance from the sensor processing system; in a sense, these are cases where the sensor data processing or future trajectory estimation system is at fault for the collision. These are more valuable cases, in that they can more easily be used to debug specific aspects of the ego vehicle control algorithms.

Scenario 3

In this last scenario, the ego vehicle is making a left turn through an intersection, while an agent vehicle in the opposing lane is also making a left turn. This scene is similar to the Scenario 2, as depicted in Fig. 37, except that Agent 1 is not present in this scenario, only Agent 2, which we refer to as the agent vehicle for this scenario.

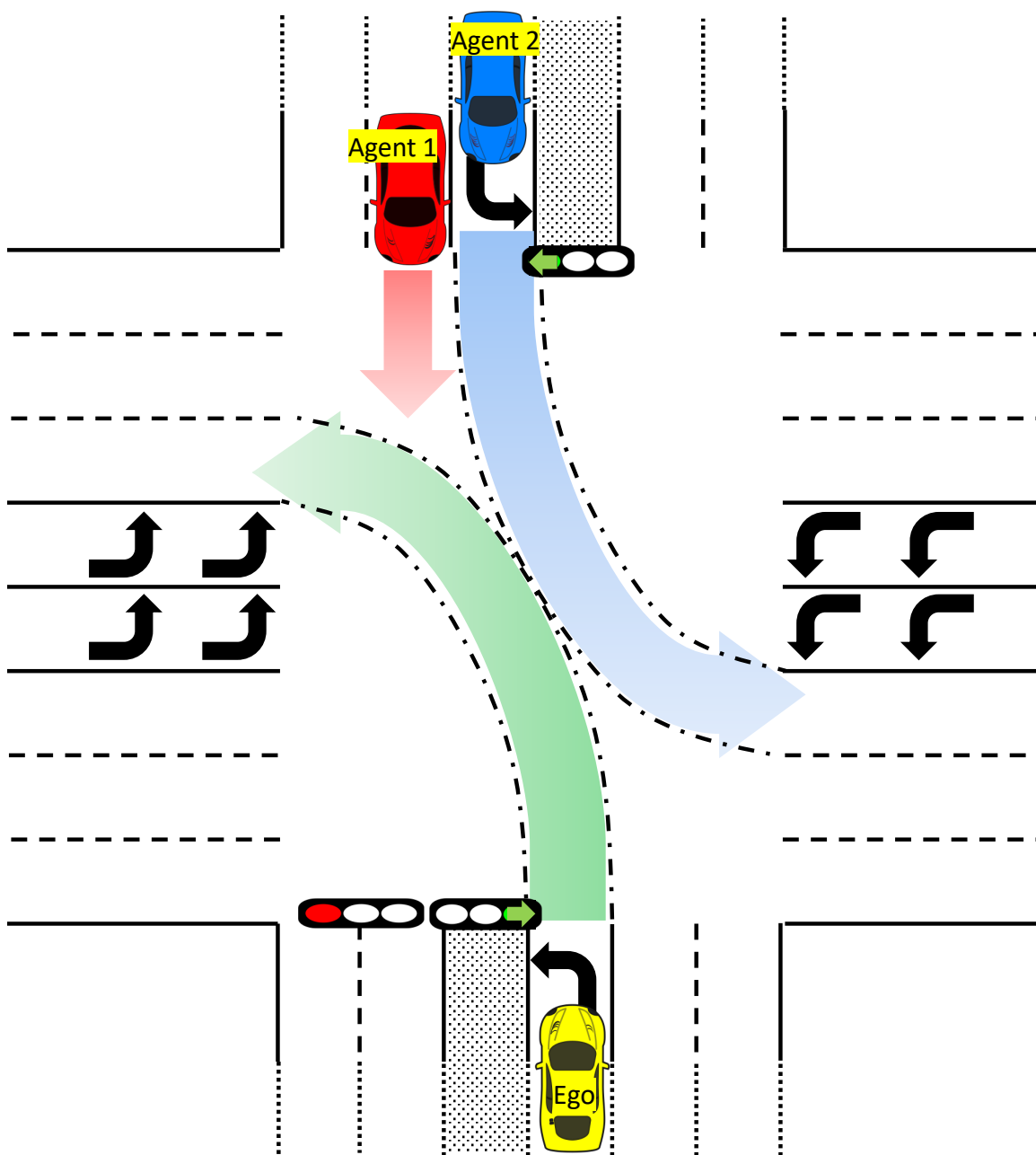


Figure 37. Overview of the scenario 2.

If both ego and agent vehicles are not accurately regulating their trajectories during this maneuver, a collision may occur. We call the model of this scenario \mathcal{M}_3 .

In this scenario, we search over target trajectories of the ego and agent vehicles. Below are the parameters that we use:

- Ego vehicle initial speed, target lateral position w.r.t its lane center when entering the intersection, distance traveled inside the intersection before starting its left turn, target lateral position w.r.t its lane center when exiting the intersection, and distance traveled inside the intersection after finishing its left turn;
- Agent vehicle speed, target lateral position w.r.t its lane center when entering the intersection, distance traveled inside the intersection before starting its left turn, target lateral position w.r.t its lane center when exiting the intersection, and distance traveled inside the intersection after finishing its left turn.

We evaluate Model \mathcal{M}_3 against requirement $\varphi 5$. The purpose of considering the performance requirement $\varphi 5$, in this case, is that scenario \mathcal{M}_3 is difficult to falsify. That is, due to the specific parameter ranges selected for the scenario, it is unlikely that the ego vehicle will collide with the agent vehicle. Instead, in this case, we are interested to identify situations where the emergency braking system unnecessarily decelerates the ego vehicle, causing unacceptable performance, from a ride-quality perspective. The scenario can easily lead to unnecessary braking, as the ego and agent vehicles momentarily move toward each other during their left turn maneuvers, which can cause the emergency braking algorithm to decide, incorrectly, that a collision is imminent. This type of case can be useful as feedback to designers, as it can highlight controller behaviors that are too conservative, at the expense of ride quality.

Summary of Test Results

We present results from experiments demonstrating the application of our framework to the scenarios and requirements described above. Table 3 summarizes the results. Indicated in the table for each case study are the requirements used to test each model, the testing approach used, the set of active sensors used, and a summary of the results. We describe the results in detail below.

Covering array and falsification on Model \mathcal{M}_1

In our previous work [169], we proposed and studied the effectiveness of a testing approach that first uses covering arrays to discover critical regions, based on a set of discrete parameters, then uses those results as the initial points for robustness guided falsification. In [169], we have empirically shown that our approach that utilizes covering arrays followed by falsification can achieve results that are closer to the global minimum compared to the uniform random testing or combination of uniform random testing with the falsification approach. Here, we apply that approach (covering arrays followed by falsification) on model \mathcal{M}_1 for 3 different requirements, φ_1 , φ_2 and φ_4 . In model \mathcal{M}_1 , we focus on the camera sensor and DNN-based object detection and classification algorithm. Because of this, most of our parameters are colors of pedestrian clothing and the agent vehicle, as described in Section 4.3. We first execute 195 covering array tests and collect simulation trajectories. Then, we compute the *robustness* values for those trajectories, with respect to the requirements φ_1 , φ_2 and φ_4 . Finally, for each requirement, starting from the case with the smallest positive robustness value, we try to find as many additional falsifications as possible,

	Model \mathcal{M}_1			Model \mathcal{M}_2	Model \mathcal{M}_3
Requirement	φ_1	φ_2	φ_4	φ_4	φ_5
Testing Modality	CA+ Falsification	CA+ Falsification	CA+ Falsification	Falsification	Falsification
Active Sensors	CCD	CCD	CCD	CCD, Radar, LIDAR	CCD, LIDAR
Computation Time	CA: 2h, 10min.			2h, 3min.	9h, 40min.
	Fals.:3h, 33min.	3h, 35min.	3h 34min.		
Number of Simulations	CA: 195			58	232
	Fals.:300	300	300		
Falsification Obtained	67 by CA + 5 by falsification	65 by CA + 8 by falsification	67 by CA + 12 by falsification	✓	✓
Application of Results	Lowest robustness cases used to create critical tests.			Falsifying cases relate to processing of specific sensor; aids in controller design improvement.	Poor performance cases used to improve controller design in modeling phase.

Table 3. Results from autonomous driving tests using virtual framework.

within a maximum of 300 extra simulations, by using a falsification approach that uses simulated annealing to perform the optimization.

For requirement φ_1 , 67 cases were falsified from the covering array tests (i.e., 67 of the 195 cases did not satisfy φ_1). Starting from 7 of the remaining (non-falsifying) cases from the covering array tests, 5 additional falsifying cases were discovered using falsification. For requirement φ_2 , 65 cases were falsified from the covering array cases, with an additional 8 cases discovered during the falsification step. For requirement φ_4 , 67 cases were falsified during the covering array step, with 12 more cases discovered during the falsification step.

These results demonstrate that we can automatically identify test cases that violate specific sensor-level, system-level, and sensor-to-system level requirements. These test cases can be fed back to the designers to improve the perception or control design or can be used as guidance to identify challenging scenarios to be used during the testing phase.

Analysis of robustness values on the falsification of Model \mathcal{M}_2

The *robustness* value, which is described in Section 4.2, for a trajectory with respect to the requirement is automatically computed in Sim-ATAV. This computation is performed by the S-TALIRO tool [63] and is used to guide the test cases towards a falsification.

We use the results of falsification on Model \mathcal{M}_2 to show, in Fig. 38, how the robustness value changes over time and finally becomes negative, which indicates falsification of the requirement. In this case, Sim-ATAV was able to find a falsifying example in 58 simulations. Because the cost function gradients are not computable,

we use a stochastic global optimization technique, Simulated Annealing (SA). The blue line shows the robustness value for each simulation. We can observe that the robustness value per simulation run is not monotonic. This is due to the stochastic nature of the optimizer; however, the achieved minimum robustness up to the current simulation is a non-increasing function, which shows the best robustness achieved after each simulation. As soon as the framework finds a test case that causes a negative robustness value, it stops the search and reports the falsifying example.

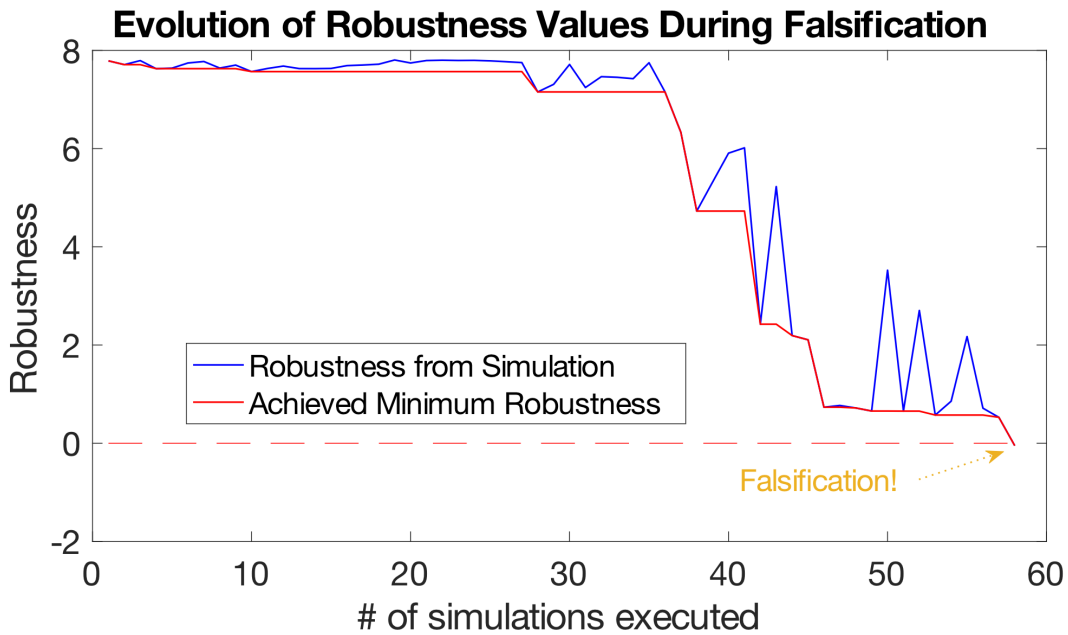


Figure 38. Robustness guided falsification utilizes global optimization techniques to guide the test cases toward falsification.

Figure 39 shows images from the simulation execution of a falsifying example for model \mathcal{M}_2 with respect to the requirement φ_2 . Between the time corresponding to Fig. 39-(a) to Fig. 39-(b), the red car approaching from the opposite side is driving on a path such that there will be a future collision with the Ego vehicle; however, due to incorrect localization of the agent vehicle, the Ego vehicle is not able to correctly

predict the future trajectory of the agent vehicle, and so it does not predict a collision. Hence it continues without taking action to avoid the collision. Starting from the moment shown on Fig. 39-(c), the Ego vehicle predicts the collision and starts applying emergency braking; however, because it takes action too late, the Ego vehicle cannot avoid a collision with the agent vehicle, as shown in Fig. 39-(d).

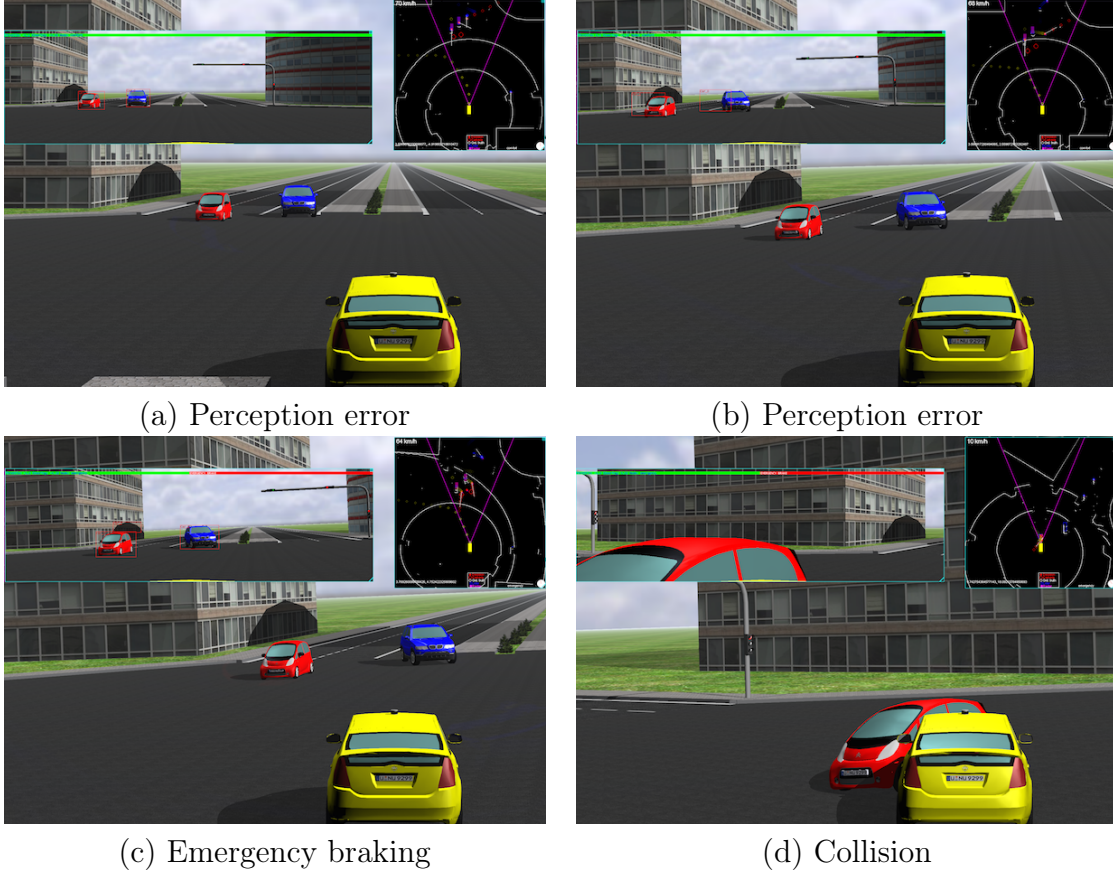


Figure 39. Time-ordered images from the falsifying example on model M_2 .

We note that, even for cases that are non-falsifying, the robustness values are useful for the system designer’s analysis, as behaviors with low robustness value are “close to” violating the requirement and therefore correspond to cases that may require closer attention.

A visual analysis of a falsifying simulation trajectory from Model M_3

As presented in Table 3, Sim-ATAV was able to find a falsifying example for model M_3 with respect to the STL requirement φ_5 in 232 simulations. We present a visual analysis of the falsifying test result. Note that this analysis is done automatically in the framework, and corresponding satisfaction/falsification of the requirement is returned to the user, along with the *robustness* value that shows the signed distance to the boundary of satisfaction or falsification. The type of visual analysis we present here may be useful for the system designers to understand the reason behind the falsification (or satisfaction) of a requirement, which can be helpful for debugging or improving the design. For this analysis, we use the definitions and notation introduced in Section 4.3.

Figure 40 shows a part of the simulation trajectory of Model M_3 for a time window around the falsification instance, together with the corresponding logic evaluations of the predicates related to the sub-formulas in Requirement φ_5 . In the top plot in Fig. 40, the red solid line is the estimated future minimum distance between the ego vehicle and Agent vehicle 1, with respect to the simulation time. This estimation is based on the ground truth information collected from the simulation and utilizes the CTRV model at each time step of the simulation to compute the expected estimation that is described in φ_5 . For this example, we define the variable C that is used in φ_5 as $(d_{f,min} < 0.5)$, where $d_{f,min}$ represents the expected minimum future distance; the dashed horizontal red line in Fig. 40 located at $0.5m$ is the threshold minimum future distance for a collision estimation. The values of t_1 and t_2 are respectively defined as 0.6 and 0.5 in this example. Since $d_{f,min}$ is never less than 0.5 in this case, the

collision estimation variable C , which is represented by the black solid line in the top plot, is always false.

The middle plot presents a similar evaluation for computing the variable B used in $\varphi 5$, which represents excessive braking. This evaluation uses the collected actual normalized brake power data, say br , from the simulation and computes the logical variable $B = (br > 0.5)$. The solid and dashed red lines represent br and the threshold value 0.5, respectively. The solid black line shows the value of B with respect to time.

The bottom plot in Fig. 40 shows the value of the variable $edge$ that is defined for the requirement $\varphi 5$ with respect to the simulation time.

The first part of the requirement $\varphi 5$, which was defined as $\Box\left(\neg\Box_{[0,t1]}(B \wedge \neg C)\right)$ in Section 4.3, would evaluate to false if and only if there would exist a time window of $t1$ seconds such that B is always true and C is always false. Focusing on the values of C and B from the top two plots in Fig. 40, we can see that although C is always false, because there is no time window of $t1 = 0.6s$ in which B is always true, the first part of the requirement evaluates to true. This means this execution of model \mathcal{M}_3 satisfies the first part of the requirement $\varphi 5$.

The second part of the requirement $\varphi 5$, which is defined as $\Box\left(\neg\left(edge \wedge \Diamond_{(0,t2]}(edge \wedge \Diamond_{(0,t2]}edge)\right)\right)$ evaluates to false if and only if there exists a series of three falling edges of B ($edge$), such that one occurrence of $edge$ follows another within a time window of $t2$ seconds. As we see in the bottom plot of Fig. 40, at time 5.6s it is true that there exists an $edge$ and it is also true that there exists another $edge$ within the time window of 0 to 0.5s following this moment (occurring at 5.85s). Hence the inner $(edge \wedge \Diamond_{(0,t2]}edge)$ inside the above formula evaluates to true at time 5.6s. If we call this event $e1$, the overall formula will evaluate to false if there exists an $edge$ that is followed by event $e1$ in a time window between 0 and $t2 = 0.5s$. This

happens at time 5.46s, which is the moment that there exists an *edge* followed by event *e1* at $0.14 \in (0, 0.5]$ seconds, *where the event e1 is defined as an edge followed by another edge within $t \in (0, 0.5]$ seconds*. Hence, the second part of the requirement φ_5 evaluates to false, and as a result, φ_5 evaluates to false at time 5.46s, since it is a conjunction of parts 1 and 2. In other words, the system falsifies (does not satisfy) the requirement φ_5 .

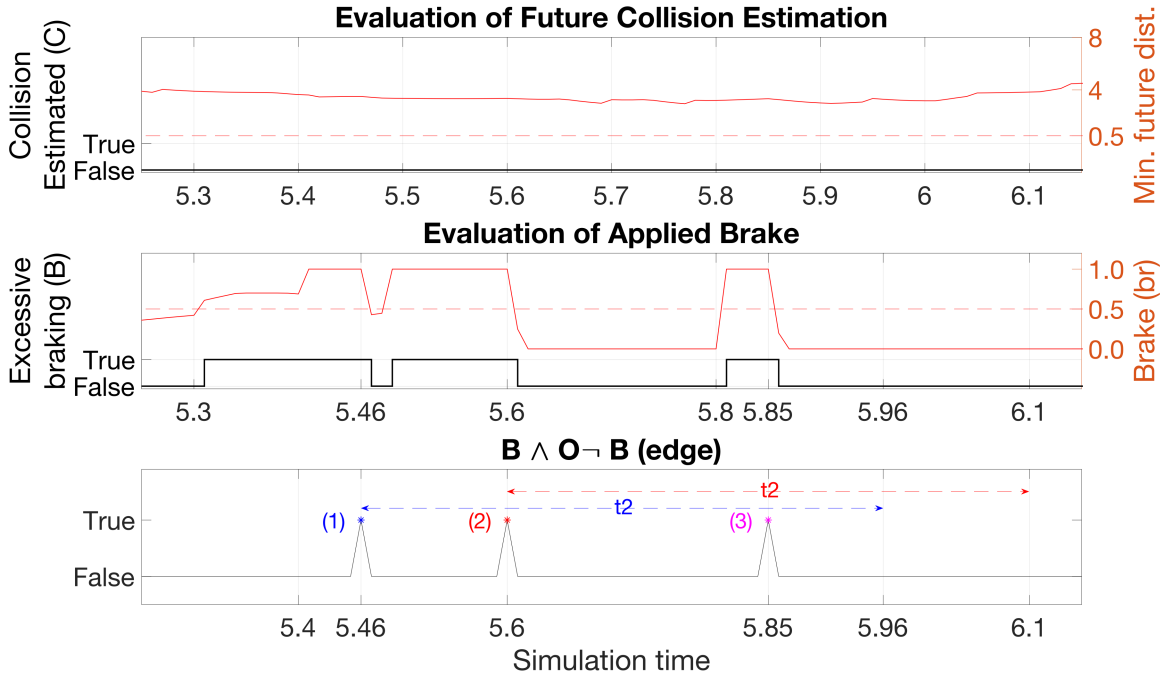


Figure 40. Analysis of falsification for Model M_3 .

4.6 Related work

Testing and evaluation methods for Autonomous¹ Vehicles (AVs) could be categorized into three major classes: (1) model based, (2) data-driven, and (3) scenario based. Scenario-based approaches utilize accident reports and driving conditions that are easily identifiable as challenging, producing specific test scenarios to be executed either in the real world or in a simulation environment. For example, Euro NCAP [164] and DOT [127] provide such scenarios. Data-driven approaches, on the other hand, typically utilize driving data [188] to generate probabilistic models of human drivers. Such models are then used for risk assessment and rare event sampling for AV algorithms under specific driving scenarios [189].

The aforementioned testing methods are important and necessary before AV deployment, but they cannot help with design exploration and automated fault detection at early development stages. Such problems are addressed by model-based verification [116, 12], model based test generation [173, 14, 174, 135, 102, 101], or a combination thereof [66, 134]. It is important to also highlight that these methods typically ignore or use simple models to abstract away proximity sensors and, especially, the vision systems. However, ignoring sensors or using simplified sensing models may be a dangerously simplifying assumption since it ignores the complex interactions between the dynamics of the vehicle and the sensors. For example, the effective sensing range of a sensor platform mounted on the roof of a vehicle is affected when the vehicle makes hard turns.

In addition, vision-based perception systems have become an integral component

¹We utilize the more general term “autonomous” as opposed to a more restricted “automated” since our methods could potentially apply to all levels of autonomy.

of the sensor platform of AVs, and in many cases, they constitute the only perception system. Currently, the winning algorithmic technology for image processing systems is utilizing Deep Neural Networks (DNN). For instance, by 2011, the DNN architecture proposed in [34] was already capable of classifying pre-segmented images of traffic signs with better accuracy than humans (99.46% vs 99.22%). Since then, there has been substantial progress with DNNs performing both segmentation and classification [94, 16]. Yet, in spite of the multiple impressive results using DNN, it is still also easy to devise methods that can produce (so-called adversarial) images that will fool them [166, 136, 72].

The latter (negative) result raises two important questions: (1) can we still generate adversarial inputs for DNN when we manipulate the physical properties and trajectories of the objects in the environment of the AV, and (2) how does the DNN accuracy affect the system level properties of an AV, that is, its functional safety? Exhaustive verification methods for NN in the loop are still in their infancy [54], and they cannot handle AV with DNN components in the loop. To address the two questions above, several model-based test generation methods have been proposed [51, 50, 52, 7, 169]. The procedure described in [51, 50, 52] analyzes the performance of the perception system using static images to identify candidate counterexamples, which are then checked using simulations of the closed-loop system to determine whether the AV exhibits unsafe behaviors. On the other hand, [7, 169] develop methods that directly search for unsafe behaviors of the closed-loop system by defining a cost function on the closed-loop behaviors. The differences between [7] and [169] are primarily on the search methods, the simulation environments, and the AVs, with [169] providing a more efficient method for combinatorial search.

In our framework, we perform adversarial test generation at the system level. We

demonstrate that our framework used for test generation for AV with multi-sensor systems as opposed to vision-only perception systems. Moreover, we demonstrate the importance and effectiveness of test generation methods guided by system-level requirements as well as perception-level requirements.

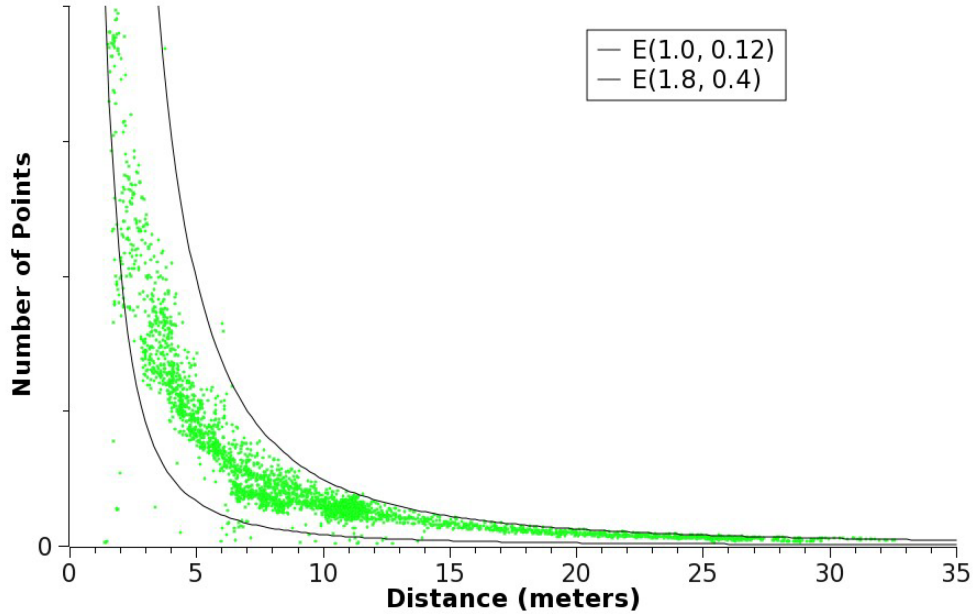


Figure 41. LIDAR reflection points versus distance to various pedestrian and pedestrian-like targets. Lines of expected number of points $E(h, w)$ are also shown.

Using our framework, we can formalize and test against requirements on the sensor performance, in the context of a driving scenario. For example, the LIDAR’s point cloud density drops significantly with the distance to the target object, for example, a pedestrian; Fig. 41 illustrates this point with experimental results showing LIDAR data point density as a function of object distance. Similar to this aspect of LIDAR behavior, the pixel count of a CCD camera would also decrease dramatically with the distance if it were to be used for pedestrian detection since the area of an observed object decreases as the square of the distance to the object. This may complicate

testing for long-range observation conditions. Our framework supports testing these aspects of sensor performance.

4.7 Conclusions

We demonstrated Sim-ATAV, a simulation-based adversarial test generation framework for automated driving systems. The framework works in a closed-loop fashion, where the system evolves in time with the feedback cycles from the autonomous vehicle’s controller. The framework includes models of LIDAR and radar sensor behaviors, as well as a model of the CCD camera sensor inputs. CCD camera images are rendered synthetically by our framework and processed using a pre-trained deep neural network (DNN). Using our framework, we demonstrated a new effective way of finding critical vehicle behaviors by using 1) covering arrays to test combinations of discrete parameters and 2) simulated annealing to find corner-cases.

Our framework allows the automatic identification of high-level descriptions of test scenarios in open environments. For instance, in Scenario 2 which is presented in Section 4.5, the initial lane numbers of the vehicles, the direction they enter the intersection (including possible wrong-way driving behavior), and whether they are driving straight, making a left turn or a right turn in the intersection can be parameterized by utilizing covering arrays. Our framework allows such flexibility in the automated generation of the test scenarios.

Future work will also include using identified counterexamples to retrain and improve the DNN-based perception system. Additionally, the scene rendering will be made more realistic by using other scene rendering tools, such as those based on state-of-the-art game engines.

Also, we note that the formal requirements that we considered were provided as an example of the type used when employing a requirements-driven development approach based on a temporal logic language, which is a formalism that may be unfamiliar to many test engineers. Future research will include investigating ways to automatically produce formal requirements based on requirements given in more traditional forms or in a visual language for expressing requirements such as ViSpec [88].

A TUTORIAL ON SIM-ATAV

5.1 An Overview of Sim-ATAV

Sim-ATAV is a tool developed for experimenting with different test generation techniques [169] for research purposes as described in Chapter 4. It is mainly developed in Python and it uses the open-source robotics toolbox Webots for 3D scene generation, vehicle and sensor modeling and simulation [167, 39]. Sim-ATAV can be interfaced with *covering array* generation tools like ACTS [108] and with *falsification* tools like S-TALiRO [60], which is a MATLAB toolbox.

Figure 42 provides a high-level overview of the framework. The main functionality of Sim-ATAV is provided by **Simulation configurator** and **Simulation Supervisor** blocks. **Simulation configurator** block represents the Sim-ATAV API for the user script to create a simulation and to receive the results. **Simulation Supervisor** block represents the part of the framework that executes inside the robotics simulation toolbox Webots. It uses the Webots API to (1) modify the simulation environment, e.g., add/configure vehicles, roads, pedestrians, provide vehicle controllers, (2) execute the simulation, and (3) collect information, e.g., the evolution of vehicle states over the simulation time. **Simulation Supervisor** receives the requested simulation configuration from the **Simulation configurator** over socket communication. When the simulation environment is set up, **Simulation Supervisor** requests Webots to start the simulation. User-provided controllers control the motion of the vehicles and pedestrians. At the end of the

simulation, **Simulation Supervisor** sends the collected simulation trace to **Simulation configurator**.

External tools like ACTS can be used to generate combinatorial tests with covering arrays. Sim-ATAV provides functions to read covering array test scenarios from `csv` files. For falsification, S-TALiRO is used to iteratively sample new configurations until a failure case is detected. Optionally, initial samples for the configurations can be read by S-TALiRO from the covering array `csv` files. The sampled configurations in S-TALiRO are passed as parameters to test generation functions using the Python interface provided in MATLAB, *i.e.*, by calling Python functions directly inside MATLAB. The test generation functions return the simulation trace back to MATLAB (and to S-TALiRO) as return values. More detail on *falsification* methods is available in [169, 60]. A running example of test generation is provided in the upcoming sections.

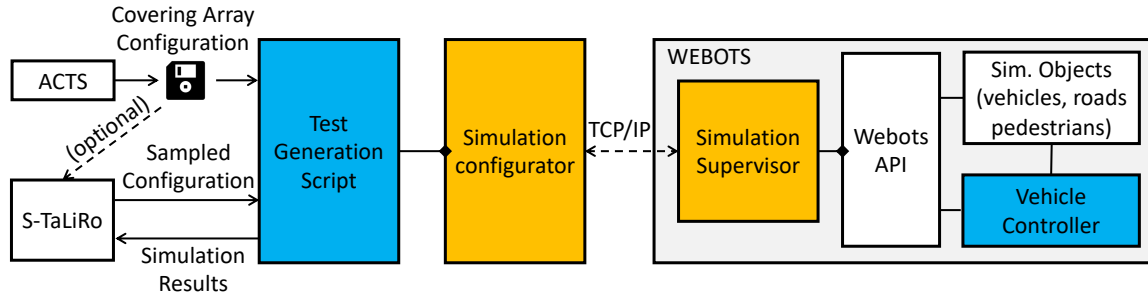


Figure 42. An Overview of Sim-ATAV.

Although they are not essential for the test generation purposes, Sim-ATAV also comes with some vehicle controller implementations as well as some basic perception system, sensor fusion, path planning and control algorithms that can be utilized by the user for Ego or Agent vehicle control.

5.2 Installation Instructions

Sim-ATAV requires Python 3.7 and Webots for basic functionality. For some controllers and test generation approaches, there are other requirements like MATLAB®, S-TALiRO, TensorFlow, SqueezeDet etc. The framework has been tested in Windows® 10 with specific versions of the required packages.

Firstly, Sim-ATAV should be either downloaded or cloned using a *git* client ². For Windows®, the preferred installation approach is to use `setup_for_windows.bat`. Once executed, it will guide the user through the installation process and automate the process as much as possible. This script is not advanced and may fail for some systems. If the script fails, please try the steps below for a manual installation.

Below are the steps for the manual installation. All the paths are given relative to the root folder for the Sim-ATAV distribution. In case any problems are experienced during the installation of the packages, most of the packages can also be found in Christoph Gohlke's website³:

1. Install Python 3.7-64 Bit
2. Install Webots r2019a.
3. (optional) If the system has CUDA-enabled GPU and it will be utilized for an increased performance:
 - a. Install CUDA Toolkit 10.0
 - b. Install CUDNN 7.3.1
4. Download `Python_Dependencies` from <http://www.public.asu.edu/~etuncali/>

²Sim-ATAV: <https://cpslab.assembla.com/spaces/sim-atav>

³Christoph Gohlke's website: <https://www.lfd.uci.edu/~gohlke/pythonlibs/>

downloads/ and unzip it next to this installation script. The Python wheel (.whl) files should be directly under ./Python_Dependencies/

5. Install commonly used Python packages:

- a. Install Numpy+MKL 1.14.6 either from Python_Dependencies folder or from Christoph Gohlke's website:

```
pip3 install --upgrade Python_Dependencies/numpy-1.14.6+  
mkl-cp37-cp37m-win_amd64.whl
```

- b. Install scipy 1.2.0:

```
pip3 install scipy==1.2.0
```

- c. Install scikit-learn:

```
pip3 install scikit-learn
```

- d. Install pandas:

```
pip3 install pandas
```

- e. Install Absl Py:

```
pip3 install absl-py
```

- f. Install matplotlib:

```
pip3 install matplotlib
```

- g. Install pykalman:

```
pip3 install pykalman
```

- h. Install Shapely:

```
pip3 install Shapely
```

! If any problems are experienced during the installation of Shapely, it can be installed from Python_Dependencies folder:

```
pip3 install Python_Dependencies/Shapely-1.6.4.  
post1-cp37-cp37m-win_amd64.whl
```

i. Install dubins:

```
pip3 install dubins
```

! If any problems are experienced when installing pydubins:

One option is to go into the folder `Python_Dependencies/pydubins` and execute `python setup.py install`. Another option is the following:

- (i) Download pydubins from github.com/AndrewWalker/pydubins.
- (ii) Do the following changes in `dubins/src/dubins.c`:

```
#ifndef M_PI  
#define M_PI 3.14159265358979323846  
#endif
```

- (iii) Call `python setup.py install` inside pydubins folder.

6. For controllers with DNN (Deep Neural Network) object detection:

! Currently, Python 3.7 support for Tensorflow is provided by a 3rd party (only for Windows). Installation wheels are provided under Python_Dependencies folder.

Check if the system CPU supports AVX2 (for increased performance) ⁴.

- a. If the system GPU has CUDA cores, CUDNN is installed and the system CPU supports AVX 2: Install Tensorflow-gpu with AVX2 support.

```
pip3 install --upgrade Python_Dependencies/tensorflow_gpu-1  
.12.0-cp37-cp37m-win_amd64.whl
```

⁴A list of CPUs with AVX2 is available at:
https://en.wikipedia.org/wiki/Advanced_Vector_Extensions#CPUs_with_AVX2

- b. If the system GPU has CUDA cores, CUDNN is installed and the system CPU does NOT support AVX 2: Install Tensorflow-gpu without AVX2 support.

```
pip3 install --upgrade
Python_Dependencies/sse2/tensorflow_gpu-1.12.0
-cp37-cp37m-win_amd64.whl
```

- c. If the system GPU does not have CUDA cores or CUDNN is not installed, and the system CPU supports AVX 2: Install Tensorflow with AVX2 support.

```
pip3 install --user --upgrade Python_Dependencies/tensorflow-1
.12.0-cp37-cp37m-win_amd64.whl
```

- d. If the system GPU does not have CUDA cores or CUDNN is not installed, and the system CPU does NOT support AVX 2: Install Tensorflow without AVX2 support.

```
pip3 install --user --upgrade
Python_Dependencies/sse2/tensorflow-1.12.0
-cp37-cp37m-win_amd64.whl
```

7. Install Python Dependencies of SqueezeDet (if the existing controllers that use SqueezeDet will be used). There is no need to install SqueezeDet separately, as it is provided in the framework.

- a. Install joblib:

```
pip3 install --upgrade joblib
```

- b. Install opencv:

```
pip3 install --upgrade opencv-contrib-python
```

- c. Install pillow:

```
pip3 install --upgrade Pillow
```

- d. Install easydict:

```
pip3 install --upgrade easydict==1.7
```

! If any problems are experienced while installing easydict, try the following:

```
cd Python_Dependencies/easydict-1.7\  
python setup.py install  
cd ../..
```

8. To design Covering Array Tests: Please request a copy and install ACTS from NIST⁵.
9. To do robustness-guided falsification, MATLAB[®] and S-TALiRO are needed:
- a. Install MATLAB from Mathworks(tested with r2017b).
 - b. Install S-TALiRO⁶.
10. After installation is finished, the Python package wheel files that are under `Python_Dependencies` folder can be deleted to save some disk space.

5.2.0.0.1 Setting to Utilize GPU:

If the system has a CUDA-enabled GPU, and CUDA Toolkit, CUDNN are installed, the variable `has_gpu` should be set to `True` in the following file to make the experiments use the system GPU for SqueezeDet:

`Sim_ATAV/classifier/classifier_interface/gpu_check.py`.

⁵ACTS tool can be requested from <https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software/downloadable-tools>

⁶S-TALiRO is available at <https://sites.google.com/a/asu.edu/s-taliro/s-taliro>

5.3 Reference Manual

5.3.1 Simulation Entities

Sim-ATAV starts building a testing scenario from an existing Webots world file provided by the user, which we will call as *base world file*. The user has the option to have all or some of the simulation entities, *i.e.*, roads, vehicles, etc., saved in the base world file before the test generation time. The user can use the functionality provided by Sim-ATAV to add more simulation entities to the world at the time of test generation. This functionality is especially useful when the search space for the tests contain some parameters of the simulation entities such as road width, number of lanes, positions of the vehicles.

This section describes the most commonly used simulation entities that can be programmatically added into the simulation world at the time of test generation. Note that Sim-ATAV may not provide the functionality to add all possible types of simulation objects that are supported by Webots. In this section, the class names and properties with their default values for the simulation entities supported by Sim-ATAV are provided. **Test Generation Script** which is developed by the user typically creates instances of required simulation entities and uses the provided functions to add those entities into the test scenario. For modifying the simulation environment beyond the capabilities of Sim-ATAV, the user can do the changes manually and save in the base world file or modify the source code of Sim-ATAV to add or change the capabilities as needed. For a deeper understanding of the possibilities, the reader is advised to get familiar with the Webots simulation environment and details of available simulation entities [39].

The class properties for the simulation entities are provided in the tables below. The first row of each table gives the class name. Other rows start with the property name, gives the default value of the property (the value used if not explicitly changed by the user), and a brief description of the property. For each class, a related source code snippet from a running example is provided. The original source code for the running example can be found as `tests/tutorial_example_1.py` in the Sim-ATAV distribution.

Figure 43 is an image from the scenario described in the running example. A yellow Ego vehicle is behind a pedestrian walking in the middle of a 3-lane road, and an agent vehicle is approaching from the opposite direction in the next lane. There is a bumpy road surface for a short distance, and a stop sign placed on the right side of the road. This is only a simple example to illustrate how to use Sim-ATAV.



Figure 43. A view from the generated scenario for the running example.

5.3.1.0.1 General information:

In Webots, all simulation entities are kept in a simulation tree and they can be easily accessed using their **DEF_NAME** property. **Position** fields are 3D arrays as $[x, y, z]$, keeping the value of the position at each axis in Webots coordinate system, where y is typically the vertical axis to the ground (in practice, this depends on the Webots world file provided by the user). **Rotation** fields are 4D arrays as $[x, y, z, \theta]$, where x, y, z represents a rotation vector and θ represents the rotation around this vector in clockwise direction.

5.3.1.1 Simulation Environment

Once an object is created for a simulation entity, the user can utilize the corresponding function provided by Sim-ATAV to add the entity to the scenario. An easier alternative is to utilize the **SimEnvironment** class provided by Sim-ATAV. An object of this class can be populated with the necessary simulation entities and passed to the simulation environment with a single function call. Table 4 summarizes the **SimEnvironment** class.

Table 4. Simulation Environment Class.

Class Name: SimEnvironment		
Property	Default	Description
fog	None	Keeps fog object.
heart_beat_config	None	The heartbeat configuration.
view_follow_config	None	The viewpoint configuration.
ego_vehicles_list	[]	The list of Ego vehicles.
agent_vehicles_list	[]	The list of agent vehicles.
pedestrians_list	[]	The list of pedestrians.
road_list	[]	The list of roads.
road_disturbances_list	[]	The list of road disturbances.
generic_sim_objects_list	[]	The list of generic simulation objects.
control_params_list	[]	The list of controller parameters that will be set in the run time.
initial_state_config_list	[]	The list of initial state configurations.
data_log_description_list	[]	The list of data log descriptions.
data_log_period_ms	None	Data log period (ms).

Example 1 *For the running example, we start with an empty simulation environment.*

*Listing 5.1 creates an empty **SimEnvironment** object that will later keep the required simulation entities.*

```

1 from Sim_ATAV.simulation_configurator.sim_environment |
2     import SimEnvironment
3
4 sim_environment = SimEnvironment()
```

Listing 5.1. Source code for creating a simulation environment.

5.3.1.2 Road

A user-configurable road structure to use in the simulation environment is given in Table 5.

Table 5. Simulation Entity: Road.

Class Name: WebotsRoad		
Property	Default	Description
def_name	“STRROAD”	Name as it appears in the simulation tree.
road_type	“StraightRoadSegment”	Road type name as used by Webots.
rotation	[0, 1, 0, math.pi/2]	Rotation of the road
position	[0, 0.02, 0]	Starting position.
number_of_lanes	2	Number of lanes.
width	number_of_lanes * 3.5	Road width (m).
length	1000	Road length (m).

Example 2 *Listing 5.2 provides a source code snippet that creates a 3-lane straight road segment lying between 1000m and -1000m along the x-axis.*

```
1 from Sim_ATAV.simulation_control.webots_road import WebotsRoad
2
3 road = WebotsRoad(number_of_lanes=3)
4 road.rotation = [0, 1, 0, -math.pi / 2]
5 road.position = [1000, 0.02, 0]
6 road.length = 2000.0
7
8 # Add the road into simulation environment object:
9 sim_environment.road_list.append(road)
```

Listing 5.2. Source code for creating a road.

5.3.1.3 Vehicle

A user-configurable vehicle class to use in the simulation environment in Table 6.

Table 6. Simulation Entity: Vehicle.

Class Name: WebotsVehicle		
Property	Default	Description
def_name	""	Name as it appears in Webots simulation tree.
vhc_id	0	Integer ID for referencing to the vehicle.
vehicle_model	"AckermannVehicle"	Vehicle model can be any model name available in Webots.
rotation	[0, 1, 0, 0]	Rotation of the object.
current_position	[0, 0.3, 0]	x,y,z values of the position.
color	[1, 1, 1]	R,G,B values of the color in the range [0,1].
controller	"void"	Name of the vehicle controller.
is_controller_name_absolute	False	Indicates where to find the vehicle controller. Find the details below.
vehicle_parameters	[]	Additional parameters for the vehicle object.
controller_parameters	[]	Parameters that will be sent to the vehicle controller.
controller_arguments	True	Arguments passed to the vehicle controller executable.
sensor_array	[]	List of sensors on the vehicle (WebotsSensor objects).

The `vehicle_model` field of a vehicle object should match with the models available in Webots (or any custom models added to Webots by the user). Webots r2019a version provides vehicle model options *ToyotaPrius*, *CitroenCZero*, *BmwX5*, *RangeRoverSportSVR*, *LincolnMKZ*, *TeslaModel3* as well as truck, motorcycle and

tractor models. When `is_controller_name_absolute` is set to true, Webots will load the given controller from `Webots_Projects/controllers` folder, otherwise, it will load the controller named `vehicle_controller` which is located under the same folder but will take the controller name as an argument.

Example 3 *We can now create vehicles and place them on the road that was created above. Listing 5.3 provides an example source code snippet that creates an Ego vehicle at the position $x = 20\text{m}$, $y = 0$, and an agent vehicle at the position $x = 300\text{m}$, $y = 3.5\text{m}$, vehicles facing toward each other. The controllers for the vehicles are set and the controller arguments are provided. The arguments accepted are controller-specific. The vehicle controllers used in this example can be found under `Webots_Projects/controllers` folder.*

```

1 from Sim_ATAV.simulation_control.webots_vehicle import WebotsVehicle
2
3 # Ego vehicle
4 ego_x_pos = 20.0 # Setting the x position of the Ego vehicle in a
   variable.
5
6 vhc_obj = WebotsVehicle()
7 vhc_obj.current_position = [ego_x_pos, 0.35, 0.0]
8 vhc_obj.current_orientation = math.pi/2
9 vhc_obj.rotation = [0.0, 1.0, 0.0, vhc_obj.current_orientation]
10 vhc_obj.vhc_id = 1
11 vhc_obj.color = [1.0, 1.0, 0.0]
12 vhc_obj.set_vehicle_model('ToyotaPrius')
13 # Name of our controller python file is '
   automated_driving_with_fusion2':
14 vhc_obj.controller = 'automated_driving_with_fusion2'
15 # Controller will be found directly under controllers folder:

```

```

16 vhc_obj.is_controller_name_absolute = True
17 # Below is a list of arguments specific to this controller.
18 # For reference, the arguments are: car_model, target_speed_kmh,
    target_lat_pos,
19 # self_vhc_id, slow_at_intersection, has_gpu, processor_id
20 vhc_obj.controller_arguments.append('Toyota')
21 vhc_obj.controller_arguments.append('70.0')
22 vhc_obj.controller_arguments.append('0.0')
23 vhc_obj.controller_arguments.append('1')
24 vhc_obj.controller_arguments.append('True')
25 vhc_obj.controller_arguments.append('False')
26 vhc_obj.controller_arguments.append('0')
27
28 # Agent vehicle:
29 vhc_obj2 = WebotsVehicle()
30 vhc_obj2.current_position = [300.0, 0.35, 3.5]
31 vhc_obj2.current_orientation = 0.0
32 vhc_obj2.rotation = [0.0, 1.0, 0.0, -math.pi/2]
33 vhc_obj2.vhc_id = 2
34 vhc_obj2.set_vehicle_model('TeslaModel3')
35 vhc_obj2.color = [1.0, 0.0, 0.0]
36 vhc_obj2.controller = 'path_and_speed_follower'
37 vhc_obj2.controller_arguments.append('20.0')
38 vhc_obj2.controller_arguments.append('True')
39 vhc_obj2.controller_arguments.append('3.5')
40 vhc_obj2.controller_arguments.append('2')
41 vhc_obj2.controller_arguments.append('False')
42 vhc_obj2.controller_arguments.append('False')
43
44 # Here, we don't save the vehicles into simulation environment yet

```

45 *# because we will add some sensors on the vehicles below.*

Listing 5.3. Source code for creating a vehicle.

5.3.1.4 Sensor

Table 7 provides an overview of the **WebotsSensor** class that can be used to describe a sensor. Webots vehicle models have specific sensor slots on the vehicles. These are typically TOP, CENTER, FRONT, RIGHT, LEFT. The property `sensor_type` accepts the type of the sensor which should match the type used by Webots. The *translation* field of the sensor can be used to place the sensor to a different position relative to its corresponding sensor slot. As sensors can vary a lot in terms or parameters, `sensor_fields` property is provided to accept names and values of the desired parameters as a list of **WebotsSensorField** objects for flexible configuration of sensors.

Table 7. Simulation Entity: Sensor.

Class Name: WebotsSensor		
Property	Default	Description
<code>sensor_type</code>	""	Type of the sensor as defined in Webots.
<code>sensor_location</code>	FRONT	Sensor slot enumeration. <FRONT, CENTER, LEFT, RIGHT, TOP>
<code>sensor_fields</code>	[]	List of WebotsSensorField objects to customize the sensor.
Class Name: WebotsSensorField		
<code>field_name</code>	""	Name of the field that will be set.
<code>field_val</code>	""	Value of the field.

Example 4 *An example source code of adding sensors to vehicles is provided in Listing 5.4. In this example, we add a compass and a GPS device that are used by*

the controllers for path following. The receiver device added to the vehicles is used by the controllers to receive new commands from *Simulation Supervisor*. We add the receivers because we will later need them to update target paths of the vehicles. Note that although the necessary infrastructure for this approach is provided by *Sim-ATAV*, it is implementation specific and not mandated. In this example, we also add a radar device to *Ego* vehicle for collision avoidance.

```

1 from Sim_ATAV.simulation_control.webots_sensor import WebotsSensor
2
3 # Add a radio receiver to the center sensor slot
4 # with the name field set to 'receiver'.
5 # This is optional and will be used to communicate with the controller
   at the run-time.
6 vhc_obj.sensor_array.append(WebotsSensor())
7 vhc_obj.sensor_array[-1].sensor_location = WebotsSensor.CENTER
8 vhc_obj.sensor_array[-1].sensor_type = 'Receiver'
9 vhc_obj.sensor_array[-1].add_sensor_field('name', '"receiver"')
10
11 # Add a compass to the center slot with the name field set to 'compass
   '.
12 vhc_obj.sensor_array.append(WebotsSensor())
13 vhc_obj.sensor_array[-1].sensor_location = WebotsSensor.CENTER
14 vhc_obj.sensor_array[-1].sensor_type = 'Compass'
15 vhc_obj.sensor_array[-1].add_sensor_field('name', '"compass"')
16
17 # Add a GPS receiver to the center sensor slot.
18 vhc_obj.sensor_array.append(WebotsSensor())
19 vhc_obj.sensor_array[-1].sensor_location = WebotsSensor.CENTER
20 vhc_obj.sensor_array[-1].sensor_type = 'GPS'
21

```

```

22 # Add a Radar to the front sensor slot with the name field set to '
    radar'.
23 vhc_obj.sensor_array.append(WebotsSensor())
24 vhc_obj.sensor_array[-1].sensor_type = 'Radar'
25 vhc_obj.sensor_array[-1].sensor_location = WebotsSensor.FRONT
26 vhc_obj.sensor_array[-1].add_sensor_field('name', '"radar"')
27
28 # Finally, add the vehicle to the simulation environment as an Ego
    vehicle.
29 sim_environment.ego_vehicles_list.append(vhc_obj)
30
31 # Similar for the agent vehicle:
32 vhc_obj2.sensor_array.append(WebotsSensor())
33 vhc_obj2.sensor_array[-1].sensor_location = WebotsSensor.CENTER
34 vhc_obj2.sensor_array[-1].sensor_type = 'Receiver'
35 vhc_obj2.sensor_array[-1].add_sensor_field('name', '"receiver"')
36
37 vhc_obj2.sensor_array.append(WebotsSensor())
38 vhc_obj2.sensor_array[-1].sensor_location = WebotsSensor.CENTER
39 vhc_obj2.sensor_array[-1].sensor_type = 'Compass'
40 vhc_obj2.sensor_array[-1].add_sensor_field('name', '"compass"')
41
42 vhc_obj2.sensor_array.append(WebotsSensor())
43 vhc_obj2.sensor_array[-1].sensor_location = WebotsSensor.CENTER
44 vhc_obj2.sensor_array[-1].sensor_type = 'GPS'
45
46 # Add the agent vehicle to the simulation environment
47 sim_environment.agent_vehicles_list.append(vhc_obj2)

```

Listing 5.4. Source code for adding sensor to a vehicle.

5.3.1.5 Pedestrian

The user-configurable pedestrian class, **WebotsPedestrian**, to use in the simulation environment is described in Table 8. Target speed and target path (trajectory) of the pedestrian are automatically passed as arguments to the given controller.

Table 8. Simulation Entity: Pedestrian.

Class Name: WebotsPedestrian		
Property	Default	Description
def_name	“PEDESTRIAN”	Name as it appears in Webots simulation tree.
ped_id	0	Integer ID for referencing to the pedestrian.
rotation	[0, 1, 0, math.pi/2.0]	Rotation of the object.
current_position	[0, 0, 0]	x,y,z values of the position.
shirt_color	[0.25, 0.55, 0.2]	R,G,B values of the shirt color in the range [0, 1].
pants_color	[0.24, 0.25, 0.5]	R,G,B values of the pants color in the range [0, 1].
shoes_color	[0.28, 0.15, 0.06]	R,G,B values of the shoes color in the range [0, 1].
controller	“void”	Name of the pedestrian controller.
target_speed	0.0	Walking speed of the pedestrian.
trajectory	[]	Walking path of the pedestrian.

Example 5 *Listing 5.5 provides an example code snippet to create a pedestrian object, and provide a target speed and a path to define the motion of the pedestrian.*

```
1 from Sim_ATAV.simulation_control.webots_pedestrian import
   WebotsPedestrian
2
3 pedestrian_speed = 3.0 # Setting the pedestrian walking speed in a
   variable.
4
5 pedestrian = WebotsPedestrian()
6 pedestrian.ped_id = 1
7 pedestrian.current_position = [50.0, 1.3, 0.0]
8 pedestrian.shirt_color = [0.0, 0.0, 0.0]
9 pedestrian.pants_color = [0.0, 0.0, 1.0]
10 pedestrian.target_speed = pedestrian_speed
11 # Pedestrian trajectory as a list of x1, y1, x2, y2, ...
12 pedestrian.trajectory = [50.0, 0.0, 80.0, -3.0, 200.0, 0.0]
13 pedestrian.controller = 'pedestrian_control'
14
15 # Add the pedestrian into the simulation environment.
16 sim_environment.pedestrians_list.append(pedestrian)
```

Listing 5.5. Source code for creating a pedestrian.

5.3.1.6 Fog

User-configurable fog class to use in the simulation environment is summarized in Table 9.

Table 9. Simulation Entity: Fog.

Class Name: WebotsFog		
Property	Default	Description
def_name	“FOG”	Name as it appears in Webots simulation tree.
fog_type	“LINEAR”	Defines the type of the fog gradient.
color	[0.93, 0.96, 1.0]	R,G,B values of the fog color in the range [0, 1].
visibility_range	1000	Visibility range of the fog (m).

Example 6 *No camera is involved in this scenario, hence fog will not impact the performance of the controller. However, an example source code snippet is provided in Listing 5.6 for reference.*

```
1 from Sim_ATAV.simulation_control.webots_fog import WebotsFog
2
3 # Creating fog with 700m visibility and adding it to the simulation
  environment.
4 sim_environment.fog = WebotsFog()
5 sim_environment.fog.visibility_range = 700.0
```

Listing 5.6. Source code for creating fog.

5.3.1.7 Road Disturbance

Road disturbance objects are solid triangular objects placed on the road surface to emulate a bumpy road surface. **WebotsRoadDisturbance**, which is described in

Table 10, contains the properties to describe how the solid objects will be placed to create a bumpy road surface.

Table 10. Simulation Entity: Road Disturbance Object.

Class Name: WebotsRoadDisturbance		
Property	Default	Description
disturbance_id	1	Object ID for later reference.
disturbance_type	INTERLEAVED	Enumerated type of the disturbance. <INTERLEAVED, FULL_LANE_LENGTH, ONLY_LEFT, ONLY_RIGHT>
rotation	[0, 1, 0, 0]	Rotation of the object.
position	[0, 0, 0]	x,y,z values of the position.
length	100	Length of the bumpy road surface (m).
width	3.5	Width of the corresponding lane.
height	0.06	Height of the disturbance (m).
surface_height	0.02	Height of the corresponding road surface (m).
inter_object_spacing	1.0	Distance between repeating solid objects on the road (m).

Example 7 An example road disturbance object creation is given in Listing 5.7.

```

1 from Sim_ATAV.simulation_control.webots_road_disturbance import
   WebotsRoadDisturbance
2
3 # Create bumpy road for 3m where there are road disturbances on both
   side of the lane
4 # of height 4cm, each separated with 0.5m.
5 road_disturbance = WebotsRoadDisturbance()
6 road_disturbance.disturbance_type = WebotsRoadDisturbance.
   TRIANGLE_DOUBLE_SIDED #i.e., INTERLEAVED
7 road_disturbance.rotation = [0, 1, 0, -math.pi / 2.0] # Same as the
   road
8 road_disturbance.position = [40, 0, 0]

```

```

9 road_disturbance.width = 3.5
10 road_disturbance.length = 3
11 road_disturbance.height = 0.04
12 road_disturbance.inter_object_spacing = 0.5
13
14 # Add road disturbance into the simulation environment object.
15 sim_environment.road_disturbances_list.append(road_disturbance)

```

Listing 5.7. Source code for creating road disturbance.

5.3.1.8 Generic Simulation Object

Generic simulation object is for adding any type of object into the simulation which is not covered above. For these objects, there are no checks performed or there are no limitations on the field values. The user can manually create any possible Webots object by setting all of its non-default field values.

Table 11. Simulation Entity: Generic Simulation Object.

Class Name: WebotsSimObject		
Property	Default	Description
def_name	""	Name as it appears in Webots simulation tree.
object_name	"Tree"	Type name of the object. Must be same as the name used by Webots.
object_parameters	[]	List of (field name, field value) tuples as strings. Names must be same as the field names used by Webots.

Example 8 *In Listing 5.8, although it is not expected to have an impact on the controller performance, a Stop Sign object is created as a generic simulation object example for reference.*

```

1 from Sim_ATAV.simulation_control.webots_sim_object import
    WebotsSimObject
2
3 sim_obj = WebotsSimObject()
4 sim_obj.object_name = 'StopSign' # The name of the object as defined
    in Webots.
5 # The field names and format as they are used by Webots.
6 sim_obj.object_parameters.append(('translation', '40 0 6'))
7 sim_obj.object_parameters.append(('rotation', '0 1 0 1.5708'))
8
9 # Add the stop sign as a generic item into the simulation environment
    object.
10 sim_environment.generic_sim_objects_list.append(sim_obj)

```

Listing 5.8. Source code for adding a stop sign to the simulation.

5.3.2 Configuring the Simulation Execution

5.3.2.1 Additional Controller Parameters

Depending on the application and implementation details, controller parameters can be directly given to the controllers or they can be sent in the run-time. To emulate runtime inputs, such as human commands, Sim-ATAV can transmit controller commands over virtual radio communication. The controller should be able to read those commands from a radio receiver and a *receiver* object should be added to one of the sensor slots of the vehicles. This is an optional approach and the user is free to use other approaches such as reading data from a file, using socket communications etc.

Table 12. Controller parameters.

Class Name: WebotsControllerParameter		
Property	Default	Description
vehicle_id	None	ID of the corresponding vehicle.
parameter_name	" "	Name of the parameter as string.
parameter_data	[]	Parameter data.

Example 9 *An example controller parameter creation.*

```

1 from Sim_ATAV.simulation_control.webots_controller_parameter |
2     import WebotsControllerParameter
3
4 # ----- Controller Parameters:
5 # Ego Target Path:
6 target_pos_list = [[-1000.0, 0.0], [1000.0, 0.0]]
7
8 # Add each target position as a controller parameter for Ego vehicle.
9 for target_pos in target_pos_list:
10     sim_environment.controller_params_list.append(
11         WebotsControllerParameter(vehicle_id=1,
12             parameter_name='target_position',
13             parameter_data=target_pos))
14
15 # Agent Target Path:
16 target_pos_list = [[1000.0, 3.5], [145.0, 3.5], [110.0, -3.5],
17     [-1000.0, -3.5]]
18 # Add each target position as a controller parameter for agent vehicle
19 .
20 for target_pos in target_pos_list:
21     sim_environment.controller_params_list.append(
22         WebotsControllerParameter(vehicle_id=2,

```

```

22     parameter_name='target_position',
23     parameter_data=target_pos))

```

Listing 5.9. Source code for creating controller parameters.

5.3.2.2 Heartbeat Configuration

Simulation Supervisor can periodically report the status of the simulation execution to Simulation Configurator with heartbeats. Simulation Configurator can further modify the simulation environment on the run time by responding to the heartbeats.

Table 13. Heartbeat Configuration.

Class Name: WebotsSimObject		
Property	Default	Description
sync_type	NO_HEART_BEAT	NO_HEART_BEAT: Do not report simulation status. WITHOUT_SYNC: Report the status and continue simulation. WITH_SYNC: Wait for new commands after each heartbeat.
period_ms	10	Period of the simulation status reporting.

Example 10 *An example heartbeat configuration.*

```

1 from Sim_ATAV.simulation_control.heart_beat import HeartBeatConfig
2
3 # Create a heartbeat configuration that will make Simulation
4   Supervisor report
5 # simulation status at every 2s and continue execution without waiting
6   for a new
7   command.
8
9 sim_environment.heart_beat_config = \

```

```

7  HeartBeatConfig(sync_type=HeartBeatConfig.WITHOUT_SYNC, period_ms
    =2000)

```

Listing 5.10. Source code for creating a heartbeat configuration.

5.3.2.3 Data Log Item

Data log items are the states that will be collected into the simulation trace by Simulation Supervisor.

Table 14. Data Item Description.

Class Name: ItemDescription		
Property	Default	Description
item_type	None	Type of the corresponding simulation entity. <TIME, VEHICLE, PEDESTRIAN>
item_index	None	Index of the corresponding simulation entity.
item_state_index	None	Index of the state that will be recorded.

Example 11 *An example list of data log items for simulation trajectory generation.*

```

1  # ——— Data Log Configurations:
2  # First entry in the simulation trace will be the simulation time:
3  sim_environment.data_log_description_list.append(
4      ItemDescription(item_type=ItemDescription.ITEM_TYPE_TIME,
5                      item_index=0, item_state_index=0))
6
7  # For each vehicle in Ego and Agent vehicles list,
8  # record x,y positions, orientation and speed:
9  for vhc_ind in range(len(sim_environment.ego_vehicles_list) + len(
10     sim_environment.agent_vehicles_list)):

```

```

11     ItemDescription(item_type=ItemDescription.ITEM_TYPE_VEHICLE,
12         item_index=vhc_ind,
13         item_state_index=WebotsVehicle.STATE_ID_POSITION_X))
14 sim_environment.data_log_description_list.append(
15     ItemDescription(item_type=ItemDescription.ITEM_TYPE_VEHICLE,
16         item_index=vhc_ind,
17         item_state_index=WebotsVehicle.STATE_ID_POSITION_Y))
18 sim_environment.data_log_description_list.append(
19     ItemDescription(item_type=ItemDescription.ITEM_TYPE_VEHICLE,
20         item_index=vhc_ind,
21         item_state_index=WebotsVehicle.STATE_ID_ORIENTATION))
22 sim_environment.data_log_description_list.append(
23     ItemDescription(item_type=ItemDescription.ITEM_TYPE_VEHICLE,
24         item_index=vhc_ind,
25         item_state_index=WebotsVehicle.STATE_ID_SPEED))
26
27 # For each pedestrian, record x and y positions:
28 for ped_ind in range(len(sim_environment.pedestrians_list)):
29     sim_environment.data_log_description_list.append(
30         ItemDescription(item_type=ItemDescription.ITEM_TYPE_PEDESTRIAN,
31             item_index=ped_ind,
32             item_state_index=WebotsVehicle.STATE_ID_POSITION_X))
33     sim_environment.data_log_description_list.append(
34         ItemDescription(item_type=ItemDescription.ITEM_TYPE_PEDESTRIAN,
35             item_index=ped_ind,
36             item_state_index=WebotsVehicle.STATE_ID_POSITION_Y))
37
38 # Set the period of data log collection from the simulation.
39 sim_environment.data_log_period_ms = 10
40

```

```

41 # Create Trajectory dictionary for later reference.
42 # Dictionary will be used to relate received simulation trace to
   object states.
43 sim_environment.populate_simulation_trace_dict()

```

Listing 5.11. Source code for creating data log items.

5.3.2.4 Initial State Configuration

Initial state configuration objects are for setting an initial state of a simulation entity.

Table 15. Initial State Configuration.

Class Name: InitialStateConfig		
Property	Default	Description
item	None	Data item for the corresponding state as an ItemDescription object.
value	None	Initial value of the corresponding state.

Example 12 *An example initial state configuration.*

```

1 from Sim_ATAV.simulation_control.initial_state_config import
   InitialStateConfig
2 from Sim_ATAV.simulation_control.item_description import
   ItemDescription
3
4 ego_init_speed_m_s = 10.0 # Keeping the Ego initial speed in a
   variable
5
6 # Create and add an initial state configuration into simulation
   environment object.

```



```

7 sim_environment.initial_state_config_list.append(
8   InitialStateConfig(item=ItemDescription(
9     item_type=ItemDescription.ITEM_TYPE_VEHICLE, # State of a vehicle
10    item_index=0, # Vehicle index 0 (first added vehicle)
11    item_state_index=WebotsVehicle.STATE_ID_VELOCITY_X), # Speed
12    value=ego_init_speed_m_s))

```

Listing 5.12. Source code for setting an initial state value.

5.3.2.5 Viewpoint Configuration

Webots viewpoint can automatically follow a simulation entity throughout the simulation. Viewpoint configuration is used to describe which object to follow if desired.

Table 16. Viewpoint Configuration.

Class Name: ViewFollowConfig		
Property	Default	Description
item_type	None	Type of the corresponding simulation entity. <TIME, VEHICLE, PEDESTRIAN>
item_index	None	Index of the corresponding simulation entity.
position	None	$[x, y, z]$ values of the initial position of the viewpoint.
rotation	None	$[x, y, z, \theta]$ Rotation of the viewpoint.

Example 13 *An example viewpoint configuration.*

```

1 from Sim_ATAV.simulation_configurator.view_follow_config import
   ViewFollowConfig

```

```

2
3 # Create a viewpoint configuration to follow Ego vehicle (vehicle
   indexed as 0 as it
4 # is the first added vehicle). Viewpoint will be positioned 15m behind
   and 3m above
5 # the vehicle.
6 sim_environment.view_follow_config = |
7   ViewFollowConfig(item_type=ItemDescription.ITEM_TYPE_VEHICLE,
8     item_index=0,
9     position=[sim_environment.ego_vehicles_list[0].current_position[0]
   - 15.0,
10    sim_environment.ego_vehicles_list[0].current_position[1] + 3.0,
11    sim_environment.ego_vehicles_list[0].current_position[2]),
12    rotation=[0.0, 1.0, 0.0, |
13      -sim_environment.ego_vehicles_list[0].current_orientation])

```

Listing 5.13. Source code for creating a viewpoint configuration.

5.3.2.6 Simulation Configuration

Simulation configuration provides the information necessary to execute a simulation through TCP/IP connection between Simulation Configurator and Simulation Supervisor.

Table 17. Simulation Configuration.

Class Name: SimulationConfig		
Property	Default	Description
world_file	"../Webots_Projects/ worlds/test_world_1.wbt"	Name of the base Webots world file.
server_port	10021	Port number for connecting to Simulation Supervisor .
server_ip	"127.0.0.1"	Port number for connecting to Simulation Supervisor .
sim_duration_ms	50000	Simulation duration (ms).
sim_step_size	10	Simulation time step (ms).
run_config_arr	[]	List of run configurations for supporting multiple parallel simulation executions (RunConfig objects).
Class Name: RunConfig		
Property	Default	Description
simulation_run_mode	FAST_NO_GRAPHICS	Webots run mode (simulation speed). REAL_TIME: Real-time speed, FAST_RUN: As fast as possible with visualization, FAST_NO_GRAPHICS: As fast as possible without visualization.

Example 14 *An example simulation configuration creation.*

```

1 from Sim_ATAV.simulation_configurator import sim_config_tools
2
3 sim_config = sim_config_tools.SimulationConfig(1)
4 sim_config.run_config_arr.append(sim_config_tools.RunConfig())
5 sim_config.run_config_arr[0].simulation_run_mode = SimData.
    SIM_TYPE_REAL_TIME
6 sim_config.sim_duration_ms = 15000 # 15s simulation
7 sim_config.sim_step_size = 10

```

```
8 sim_config.world_file = '../Webots_Projects/worlds/empty_world.wbt'
```

Listing 5.14. Source code for creating a simulation configuration.

5.3.3 Executing the Simulation

To start execution of a scenario, it is advised to first start Webots with the base world file manually. If Webots crashes or communication is lost, Sim-ATAV can restart Webots when necessary (only if Webots is executing in the same system). To start the simulation, **Simulation Configurator** should be used to connect to the **Simulation Supervisor**, send the simulation environment and configuration details, start the simulation, and finally collect the simulation trace. Below is an example source code for this step.

Example 15 *An example for execution of the simulation.*

```
1 from Sim_ATAV.simulation_configurator.sim_environment_configurator  
   import SimEnvironmentConfigurator  
2  
3 # Create a Simulation Configurator with the previously defined sim.  
   configuration.  
4 sim_env_configurator = SimEnvironmentConfigurator(sim_config=  
   sim_config)  
5  
6 # Connect to the Simulation Supervisor.  
7 # Try maximum of 3 (max_connection_retry) times to connect.  
8 (is_connected, simulator_instance) = sim_env_configurator.connect(  
   max_connection_retry=3)  
9 if not is_connected:  
10     raise ValueError('Could not connect!')
```

```

11
12 # Setup the scenario with previously populated Simulation Environment
   object.
13 sim_env_configurator.setup_sim_environment(sim_environment)
14
15 # Execute the simulation and get the simulation trace.
16 trajectory = sim_env_configurator.run_simulation_get_trace()

```

Listing 5.15. Source code for executing the scenario.

5.4 Combinatorial Testing

Sim-ATAV provides basic functionality to read test scenarios from `csv` files, in which, each row represents a test case and each column holds the values of a test parameter. This feature can be used to automatically execute a set of predefined test cases. The user can create `csv` files that contain a set of test cases either manually or by using a tool. Table 18 provides a list of methods provided in `covering_array_utilities.py`.

<hr/> load_experiment_data (file_name, header_line_count=6, index_col=None) <hr/>	
<i>Description:</i>	Loads test cases from a csv file.
<i>Arguments:</i>	file_name: Name of the csv file header_line_count: Number of header lines to skip. (default: 6) index_col: Index of the column that is used for data indexing. (default: None)
<i>Return:</i>	A <i>pandas</i> data frame (a tabular data structure) containing all test cases.
<hr/> get_experiment_all_fields (exp_data_frame, exp_ind) <hr/>	
<i>Description:</i>	Returns all parameters for a test case (one row of the test table).
<i>Arguments:</i>	exp_data_frame: Data frame that was read using load_experiment_data exp_ind: Index of the experiment
<i>Return:</i>	A row from the test table that contains the requested test case.
<hr/> get_field_value_for_current_experiment (cur_experiment, field_name) <hr/>	
<i>Description:</i>	Returns the value of the requested test parameter.
<i>Arguments:</i>	cur_experiment: Current test case that was read using get_experiment_all_fields . field_name: Name of the test parameter.
<i>Return:</i>	Value of the requested parameter for the given test case.
<hr/>	

Table 18. A list of Sim-ATAV methods for loading test cases from csv files.

For combinatorial testing, ACTS from NIST [108] can be used as a tool for generating covering arrays. Providing a complete guide on covering arrays and how to use ACTS is out of the scope of this chapter. The user of ACTS creates a system definition by defining the system parameters and providing the possible values for each parameter. The system definition is saved as an `xml` file. ACTS can generate a covering array of desired strength and export the outputs in a `csv` file. An example of a system definition and a corresponding 2-way covering array are provided in Sim-ATAV distribution in `tests/examples/TutorialExampleSystemACTS.xml` and `tests/examples/TutorialExample_CA_2way.csv` files, respectively. Below is an example of reading test cases from a `csv` file and running each test one by one. Note

that, in the example, simulation trajectories are not evaluated. Outputs of each test case should be evaluated to decide the test result.

Example 16 *An example for running covering array test cases.*

In this example, a 2-way covering array for the test parameters is generated in ACTS. Table 19 shows the original test parameter name from the file `tutorial_example_1.py`, the name used in ACTS, and the possible values for each parameter (corresponding ACTS file is `TutorialExampleSystemACTS.xml`).

Table 19. Describing test parameters for creating a covering array.

Parameter	Name in ACTS	Possible Values
ego_init_speed_m_s	ego_init_speed	0, 5, 10, 15
ego_x_pos	ego_x_position	15, 20, 25
pedestrian_speed	pedestrian_speed	2,3,4,5

After entering the parameter descriptions in Table 19 to ACTS, a 2-way covering array is generated and exported as a csv file. Listing 5.16 shows the contents of the output csv file. The file is available as: `tests/examples/TutorialExample_CA_2way.csv`.

```
# ACTS Test Suite Generation: Mon Jan 14 22:46:33 MST 2019
# '*' represents don't care value
# Degree of interaction coverage: 2
# Number of parameters: 3
# Maximum number of values per parameter: 4
# Number of configurations: 16
ego_init_speed,ego_x_position,pedestrian_speed
0,20,2
0,25,3
0,15,4
0,20,5
5,25,2
5,15,3
```

```

5,20,4
5,25,5
10,15,2
10,20,3
10,25,4
10,15,5
15,20,2
15,25,3
15,15,4
15,15,5

```

Listing 5.16. CSV file containing a set of test cases generated by ACTS.

Below is the source code that is reading the test cases from the csv file using the functions listed in Table 18 and running each test case one by one.

```

1 import time
2 from Sim_ATAV.simulation_configurator import covering_array_utilities
3
4 def run_test(ego_init_speed_m_s=10.0, ego_x_pos=20.0, pedestrian_speed
   =3.0):
5     """Runs a test with the given arguments"""
6     .
7     .
8     .
9     # This function is creating the test scenario, executing it and
   returning trajectory as described in previous section. Content is
   not repeated here for space considerations.
10    .
11    .
12    .
13    return trajectory

```



```

14
15 def run_covering_array_tests():
16     """Runs all tests from the covering array csv file"""
17     exp_file_name = 'TutorialExample_CA_2way.csv' # csv file
18     containing the tests
19
20     # Read all experiment into a table:
21     exp_data_frame = covering_array_utilities.load_experiment_data(
22     exp_file_name, header_line_count=6)
23
24     # Decide number of experiments based on the number of entries in
25     the table.
26
27     num_of_experiments = len(exp_data_frame.index)
28
29     trajectories_dict = {} # A dictionary data structure to keep
30     simulation traces.
31
32     for exp_ind in range(num_of_experiments): # For each test case
33         # Read the current test case
34         current_experiment = covering_array_utilities.
35         get_experiment_all_fields(
36         exp_data_frame, exp_ind)
37
38         # Read the parameters from the current test case:
39         ego_init_speed = float(
40         covering_array_utilities.
41         get_field_value_for_current_experiment(
42         current_experiment, 'ego_init_speed'))
43         ego_x_position = float(

```

```

37         covering_array_utilities.
get_field_value_for_current_experiment(
38         current_experiment, 'ego_x_position'))
39     pedestrian_speed = float(
40         covering_array_utilities.
get_field_value_for_current_experiment(
41         current_experiment, 'pedestrian_speed'))
42
43     # Execute the test case and record the resulting simulation
trace:
44     trajectories_dict[exp_ind] = run_test(ego_init_speed_m_s=
ego_init_speed,
45     ego_x_pos=ego_x_position, pedestrian_speed=pedestrian_speed)
46     time.delay(2) # Give Webots some time to reload the world
47     return trajectories_dict

```

Listing 5.17. Source code for executing the covering array test cases.

5.5 Falsification / Search-based Testing

For performing falsification, a CPS falsification tool like S-TALiRO [60] can be used. As S-TALiRO is in MATLAB, we need to call the Sim-ATAV test cases which are developed in Python from MATLAB. In this section, first a simple approach is described to call the test cases from MATLAB, then a simple S-TALiRO setup is described to perform falsification. This chapter does not provide a complete guide to S-TALiRO. Reader is referred to S-TALiRO website for further details ⁷.

⁷S-TALiRO iwebsite: <https://sites.google.com/a/asu.edu/s-taliro/s-taliro>

5.5.1 Connecting to MATLAB

MATLAB[®] has built-in support to call Python functions. However, type conversions are required both in Sim-ATAV and in the MATLAB code ⁸. Sim-ATAV provides basic functionality to convert simulation trace to a format (single dimensional list) that can be easily interpreted in MATLAB. Listing 5.18 shows the necessary updates on the Python side. First, `sim_duration` argument is added to `run_test` function allow S-TALiRO run different length of simulations. Next, `for_matlab` argument is added to tell the function is called from MATLAB so that `experiment_tools.numpyArray2Matlab` method from Sim-ATAV is used to convert simulation trace to a single list for MATLAB.

```
1 def run_test(ego_init_speed_m_s=10.0, ego_x_pos=20.0, pedestrian_speed
    =3.0, sim_duration=15000, for_matlab=False):
2     """Runs a test with the given arguments"""
3     ...
4     # This function is creating the test scenario, executing it and
    returning trajectory as described in previous section. Content is
    not repeated here for space considerations.
5     ...
6     if for_matlab:
7         trajectory = experiment_tools.numpyArray2Matlab(trajectory)
8     return trajectory
```

Listing 5.18. Source code for executing the covering array test cases.

On the MATLAB side, we need a wrapper function to call the function `run_test`. Listing 5.19 provides an example to such a wrapper func-

⁸The user is free to use a different approach than what is described here.

tion. This MATLAB code can be found in Sim-ATAV distribution in the file `/tests/examples/run_tutorial_example_from_matlab.m`. Unused return parameters `YT`, `LT`, `CLG`, `GRD` and input arguments `steptime`, `InpSignal` are to maintain compatibility with S-TALIRO function call format.

```
1 function [T, XT, YT, LT, CLG, GRD] = run_tutorial_example_from_matlab(  
    XPoint, sim_duration_s, steptime, InpSignal)  
2 %run_tutorial_example_from_matlab Run Webots simulation with the  
    parameters in XPoint.  
3 % XPoint contains: [ego_init_speed_m_s, ego_x_pos, pedestrian_speed]  
4  
5 % Run the simulation and receive the trajectory:  
6 traj = py.tutorial_example_1.run_test(XPoint(1), XPoint(2), XPoint(3),  
    int32(sim_duration_s*1000.0), true);  
7  
8 % Convert trajectory to matlab array  
9 mattraj = Core_py2matlab(traj); % Core_py2matlab is from Matlab  
    fileexchange, developed by Kyle Wayne Karhohs  
10 YT = [];  
11 LT = [];  
12 CLG = [];  
13 GRD = [];  
14 if isempty(mattraj)  
15     T = [];  
16     XT = [];  
17 else  
18     % Separate time from the simulation trace:  
19     T = mattraj(:,1)/1000.0; % Also, convert time to s from ms  
20     XT = mattraj(:, 2:end); % Rest of the trace  
21 end
```

22 **end**

Listing 5.19. Matlab code as a wrapper to execute Python test execution function.

5.5.2 Connecting to S-TALiRO

To perform falsification with S-TALiRO, an MTL specification should be defined, ranges for test parameters and the MATLAB code, which is given in Listing 5.19, for running a simulation should be provided to S-TALiRO as the model under test. Listing 5.20 gives a very simple example for falsification with an MTL requirement that checks x and y coordinates of Ego and agent vehicles to decide a collision. This MATLAB code can be found in Sim-ATAV distribution in `/tests/examples/run_falsification.m`. The requirement is not to collide onto the agent vehicle. Note that, this requirement is very simplified to provide a simple example source code. A more complete check for collision would require incorporating further vehicle details and/or extracting collision information from the simulation.

```
1 % Indices of states in simulation trace:
2 cur_traj_ind = 1;
3 EGO_X = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
4 EGO_Y = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
5 EGO_THETA = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
6 EGO_V = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
7 AGENT_X = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
8 AGENT_Y = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
9 AGENT_THETA = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
10 AGENT_V = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
11 PED_X = cur_traj_ind; cur_traj_ind = cur_traj_ind + 1;
12 PED_Y = cur_traj_ind;
```

```

13 NUM_ITEMS_IN_TRAJ = cur_traj_ind;
14
15 % Predicates for MIL requirement:
16 ii = 1;
17 preds(ii).str='y_check1';
18 preds(ii).A = zeros(1, NUM_ITEMS_IN_TRAJ);
19 preds(ii).A(AGENT_Y) = 1;
20 preds(ii).A(EGO_Y) = -1;
21 preds(ii).b = 1.5;
22
23 ii = ii+1;
24 preds(ii).str='y_check2';
25 preds(ii).A = zeros(1, NUM_ITEMS_IN_TRAJ);
26 preds(ii).A(AGENT_Y) = -1;
27 preds(ii).A(EGO_Y) = 1;
28 preds(ii).b = 1.5;
29
30 ii = ii+1;
31 preds(ii).str='x_check1';
32 preds(ii).A = zeros(1, NUM_ITEMS_IN_TRAJ);
33 preds(ii).A(AGENT_X) = 1;
34 preds(ii).A(EGO_X) = -1;
35 preds(ii).b = 8;
36
37 ii = ii+1;
38 preds(ii).str='x_check2';
39 preds(ii).A = zeros(1, NUM_ITEMS_IN_TRAJ);
40 preds(ii).A(AGENT_X) = -1;
41 preds(ii).A(EGO_X) = 1;
42 preds(ii).b = 0;

```

```

43
44 % Metric Temporal Logic Requirement:
45 phi = '[](! (y_check1 /\ y_check2 /\ x_check1 /\ x_check2))';
46
47 % Ranges for test parameters (ego_init_speed_m_s, ego_x_pos,
    pedestrian_speed):
48 init_cond = [0.0, 15.0;
49              15.0, 25.0;
50              2.0, 5.0];
51
52 % Provide our Matlab wrapper function for running the tests as the model
    .
53 model = @run_tutorial_example_from_matlab;
54 opt = staliro_options();
55 opt.runs = 1; % Do falsification only once.
56 opt.black_box = 1; % Because we use a custom Matlab function as the
    model.
57 opt.SampTime = 0.010; % Sample time. Same as Webots world time step.
58 opt.spec_space = 'X'; % Requirements are defined on state space.
59 opt.optimization_solver = 'SA_Taliro'; % Use Simulated Annealing
60 opt.taliro = 'dp_taliro'; % Use dp_taliro to compute robustness
61 opt.map2line = 0;
62 opt.falsification = 1; % Stop when falsified
63 opt.optim_params.n_tests = 100; % maximum number of tries
64 sim_duration = 15.0;
65
66 disp(['Running S-TaLiRo ... '])
67 [results, history] = staliro(model, init_cond, [], [], phi, preds,
    sim_duration, opt);
68

```

```

69 res_filename = ['results_', datestr(datetime("now"), 'yyyy_mm_dd__HH_MM'
    ), '.mat'];
70 save(res_filename)
71 disp(["Results are saved to: ", res_filename])

```

Listing 5.20. Matlab code for running falsification with S-TaLiRo.

5.6 Other Remarks

A user guide for Sim-ATAV is provided with a running example. The source code for the running example used in this chapter is available in Sim-ATAV distribution under `tests/examples` folder. As Sim-ATAV is a research tool that is not directly targeted for production-level systems special caution should be taken before using it for testing any critical functionality. Sim-ATAV is still evolving and it may contain a number of bugs or parts that are open to optimization. Here, only major functionality provided by Sim-ATAV is discussed. It provides further functionality like a number of sample vehicle controller subsystems, additional computations on simulation trajectory such as collision detection, and functionality toward evaluating perception-system performance. For a deeper understanding of Sim-ATAV's capabilities, the reader is encouraged to go through the source code for examples and experiments that are used as case studies for publications.

BIBLIOGRAPHY

- [1] E. H. Aarts and P. J. Van Laarhoven. Statistical cooling: A general approach to combinatorial optimization problems. *Philips J. Res.*, 40(4):193–226, 1985.
- [2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint:1603.04467*, 2016.
- [3] H. Abbas and G. Fainekos. Convergence proofs for simulated annealing falsification of safety properties. In *Communication, Control, and Computing (Allerton), 2012 50th Annual Allerton Conference on*, pages 1594–1601. IEEE, 2012.
- [4] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95, 2013.
- [5] H. Abbas, B. Hoxha, G. Fainekos, and K. Ueda. Robustness-guided temporal logic testing and verification for stochastic cyber-physical systems. In *The 4th Annual IEEE International Conference on Cyber Technology in Automation, Control and Intelligent*, pages 1–6. IEEE, 2014.
- [6] H. Abbas, H. Mittelman, and G. Fainekos. Formal property verification in a conformance testing framework. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 12th ACM/IEEE International Conference on*, pages 155–164. IEEE, 2014.
- [7] H. Abbas, M. O’Kelly, A. Rodionova, and R. Mangharam. Safe at any speed: A simulation-based test harness for autonomous vehicles. In *7th International Workshop on Cyber-Physical Systems (CyPhy)*, 2017.
- [8] H. Abbas, A. Winn, G. Fainekos, and A. A. Julius. Functional gradient descent method for metric temporal logic specifications. In *American Control Conference (ACC), 2014*, pages 2312–2317. IEEE, 2014.
- [9] M. A. Abramson and C. Audet. Convergence of mesh adaptive direct search to second-order stationary points. *SIAM Journal on Optimization*, 17(2):606–619, 2006.
- [10] A. Adimoolam, T. Dang, A. Donzé, J. Kapinski, and X. Jin. Classification and coverage-based falsification for embedded control systems. In *International Conference on Computer Aided Verification*, pages 483–503. Springer, 2017.
- [11] M. Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

- [12] M. Althoff, D. Althoff, D. Wollherr, and M. Buss. Safety verification of autonomous vehicles for coordinated evasive maneuvers. In *Intelligent vehicles symposium (IV), 2010 IEEE*, pages 1078–1083. IEEE, 2010.
- [13] M. Althoff and J. M. Dolan. Online verification of automated road vehicles using reachability analysis. *Robotics, IEEE Transactions on*, 30(4):903–918, 2014.
- [14] M. Althoff and S. Lutz. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *IEEE Intelligent Vehicles Symposium (IV)*, 2018.
- [15] R. Alur. *Principles of cyber-physical systems*. MIT Press, 2015.
- [16] A. Angelova, A. Krizhevsky, and V. Vanhoucke. Pedestrian detection with a large-field-of-view deep network. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '15)*, 2015.
- [17] Y. S. R. Annapureddy and G. E. Fainekos. Ant colonies for temporal logic falsification of hybrid systems. In *IECON 2010-36th Annual Conference on IEEE Industrial Electronics Society*, pages 91–96. IEEE, 2010.
- [18] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan. S-TaLiRo: A tool for temporal logic falsification for hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257. Springer, 2011.
- [19] N. Aréchiga, S. M. Loos, A. Platzer, and B. H. Krogh. Using theorem provers to guarantee closed-loop system properties. In *2012 American Control Conference (ACC)*, pages 3573–3580. IEEE, 2012.
- [20] E. Asarin, O. Bournez, T. Dang, and O. Maler. Approximate reachability analysis of piecewise-linear dynamical systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 20–31. Springer, 2000.
- [21] E. Asarin, T. Dang, and A. Girard. Reachability analysis of nonlinear systems using conservative approximation. In *International Workshop on Hybrid Systems: Computation and Control*, pages 20–35. Springer, 2003.
- [22] C. Audet and J. E. Dennis Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on optimization*, 17(1):188–217, 2006.
- [23] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Nickovic, and S. Sankaranarayanan. Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 128–168. Springer, 2018.

- [24] K. Bengler, K. Dietmayer, B. Farber, M. Maurer, C. Stiller, and H. Winner. Three decades of driver assistance systems: Review and future perspectives. *Intelligent Transportation Systems Magazine, IEEE*, 6(4):6–22, 2014.
- [25] T. Bock, M. Maurer, and G. Farber. Validation of the vehicle in the loop (VIL); a milestone for the simulation of driver assistance systems. In *Intelligent Vehicles Symposium, 2007 IEEE*, pages 612–617. IEEE, 2007.
- [26] M. S. Branicky, M. M. Curtiss, J. Levine, and S. Morgan. Sampling-based planning, control and verification of hybrid systems. *IEE Proceedings-Control Theory and Applications*, 153(5):575–590, 2006.
- [27] M. Buehler, K. Iagnemma, and S. Singh. *The DARPA urban challenge: autonomous vehicles in city traffic*, volume 56. springer, 2009.
- [28] W. Burgard, U. Franke, M.ENZWEILER, and M. Trivedi. The Mobile Revolution - Machine Intelligence for Autonomous Vehicles (Dagstuhl Seminar 15462). *Dagstuhl Reports*, 5(11):62–70, 2016.
- [29] J. Campbell, C. E. Tuncali, P. Liu, T. P. Pavlic, U. Ozguner, and G. Fainekos. Modeling concurrency and reconfiguration in vehicular systems: A π -calculus approach. In *Automation Science and Engineering (CASE), 2016 IEEE International Conference on*, pages 523–530. IEEE, 2016.
- [30] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.
- [31] R. Chen, J. Xu, S. Zhang, C.-H. Chen, and L. H. Lee. An effective learning procedure for multi-fidelity simulation optimization with ordinal transformation. In *Automation Science and Engineering (CASE), 2015 IEEE International Conference on*, pages 702–707. IEEE, 2015.
- [32] L. Chi and Y. Mu. Deep steering: Learning end-to-end driving model from spatial and temporal visual cues. *arXiv preprint arXiv: 1708.03798*, 2017.
- [33] A. Christensen, A. Cunningham, J. Engelman, C. Green, C. Kawashima, S. Kiger, D. Prokhorov, L. Tellis, B. Wendling, and F. Barickman. Key considerations in the development of driving automation systems. In *24th Enhanced Safety of Vehicles Conferences*, 2015.
- [34] D. Cireşan, U. Meier, J. Masci, and J. Schmidhuber. A committee of neural networks for traffic sign classification. In *Proceedings of the 2011 International Joint Conference on Neural Networks (IJCNN)*, pages 1918–1921, 2011.

- [35] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.
- [36] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.
- [37] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.
- [38] P. Corke. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised*, volume 118. Springer, 2017.
- [39] Cyberbotics. Webots user manual and reference manual. <https://www.cyberbotics.com/support>. Accessed: 2019-02-28.
- [40] T. Dang, A. Donzé, O. Maler, and N. Shalev. Sensitive state-space exploration. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 4049–4054. IEEE, 2008.
- [41] R. De Maesschalck, D. Jouan-Rimbaud, and D. L. Massart. The mahalanobis distance. *Chemometrics and intelligent laboratory systems*, 50(1):1–18, 2000.
- [42] A. Dokhanchi, B. Hoxha, and G. Fainekos. Formal requirement debugging for testing and verification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2):34, 2018.
- [43] A. Dokhanchi, B. Hoxha, C. E. Tuncali, and G. Fainekos. An efficient algorithm for monitoring practical tptl specifications. In *Formal Methods and Models for System Design (MEMOCODE), 2016 ACM/IEEE International Conference on*, pages 184–193. IEEE, 2016.
- [44] A. Dokhanchi, A. Zutshi, R. T. Sriniva, S. Sankaranarayanan, and G. Fainekos. Requirements driven falsification with coverage metrics. In *Proceedings of the 12th International Conference on Embedded Software*, pages 31–40. IEEE Press, 2015.
- [45] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *International Conference on Computer Aided Verification*, pages 167–170. Springer, 2010.
- [46] A. Donze and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In *Formal Modelling and Analysis of Timed Systems*, volume 6246 of *LNCS*. Springer, 2010.

- [47] M. Dorigo, V. Maniezzo, A. Colorni, et al. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, man, and cybernetics, Part B: Cybernetics*, 26(1):29–41, 1996.
- [48] T. Dozat. Incorporating nesterov momentum into adam. In *International Conference on Learning Representations (ICLR) Workshop*, 2016.
- [49] T. Dreossi, T. Dang, A. Donzé, J. Kapinski, X. Jin, and J. V. Deshmukh. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *7th International Symposium NASA Formal Methods (NFM)*, volume 9058 of *LNCS*, pages 127–142. Springer, 2015.
- [50] T. Dreossi, A. Donze, and S. A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. In *NASA Formal Methods (NFM)*, volume 10227 of *LNCS*, pages 357–372. Springer, 2017.
- [51] T. Dreossi, S. Ghosh, A. Sangiovanni-Vincentelli, and S. A. Seshia. Systematic testing of convolutional neural networks for autonomous driving. In *Reliable Machine Learning in the Wild (RMLW)*, 2017.
- [52] T. Dreossi, S. Jha, and S. A. Seshia. Semantic adversarial deep learning. *arXiv:1804.07045v2*, 2018.
- [53] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
- [54] S. Dutta, S. Jha, S. Sankaranarayanan, and A. Tiwari. Learning and verification of feedback control systems using feedforward neural networks. In *Analysis and Design of Hybrid Systems*, 2018.
- [55] M. Egerstedt and C. Martin. *Control Theoretic Splines: Optimal Control, Statistics, and Path Planning*. Princeton University Press, 2009.
- [56] M. Elbanhawi, M. Simic, and R. Jazar. In the passenger seat: Investigating ride comfort measures in autonomous cars. *IEEE Intelligent Transportation Systems Magazine*, 7(3):4–17, Fall 2015.
- [57] Y. Eldar, M. Lindenbaum, M. Porat, and Y. Y. Zeevi. The farthest point strategy for progressive image sampling. *IEEE Transactions on Image Processing*, 6(9):1305–1315, 1997.
- [58] J. M. Esposito, J. Kim, and V. Kumar. Adaptive RRTs for validating hybrid robotic control systems. In *Algorithmic Foundations of Robotics VI*, pages 107–121. Springer, 2004.

- [59] D. J. Fagnant and K. Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, 2015.
- [60] G. Fainekos, S. Sankaranarayanan, K. Ueda, and H. Yazarel. Verification of automotive control applications using S-TaLiRo. In *Proceedings of the American Control Conference*, 2012.
- [61] G. E. Fainekos and K. C. Giannakoglou. Inverse design of airfoils based on a novel formulation of the ant colony optimization method. *Inverse Problems in Engineering*, 11(1):21–38, 2003.
- [62] G. E. Fainekos, A. Girard, and G. J. Pappas. Temporal logic verification using simulation. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 171–186. Springer, 2006.
- [63] G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Testing and Runtime Verification*, volume 4262 of *LNCS*, pages 178–192. Springer, 2006.
- [64] G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous-time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.
- [65] P. Falcone, F. Borrelli, J. Asgari, H. E. Tseng, and D. Hrovat. Predictive active steering control for autonomous vehicle systems. *IEEE Transactions on control systems technology*, 15(3):566–580, 2007.
- [66] C. Fan, B. Qi, and S. Mitra. Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features. *IEEE Design Test*, 35(3):31–38, June 2018.
- [67] S.-K. S. Fan and E. Zahara. A hybrid simplex search and particle swarm optimization for unconstrained optimization. *European Journal of Operational Research*, 181(2):527–548, 2007.
- [68] F. N. Fritsch and R. E. Carlson. Monotone piecewise cubic interpolation. *SIAM Journal on Numerical Analysis*, 17(2):238–246, 1980.
- [69] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012.
- [70] General Motors. Self-driving safety report, 2018.

- [71] A. Girard and G. J. Pappas. Approximate bisimulations for nonlinear dynamical systems. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 684–689. IEEE, 2005.
- [72] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
- [73] B. Grabe, T. Ike, and M. Hötter. Evidence based evaluation method for grid-based environmental representation. In *FUSION*, pages 1234–1240. IEEE, 2009.
- [74] P. Green. Motion sickness and concerns for self-driving vehicles: A literature review. <http://umich.edu/~driving/publications/Motion-Sickness--Report-061616pg-sent.pdf>. Accessed: 2019-02-28.
- [75] R. Grosu and S. A. Smolka. Monte carlo model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 271–286. Springer, 2005.
- [76] I. Haghghi, A. Jones, Z. Kong, E. Bartocci, R. Gros, and C. Belta. SpaTeL: a novel spatial-temporal logic and its applications to networked systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 189–198. ACM, 2015.
- [77] D. L. Hall and J. Llinas. An introduction to multisensor data fusion. *Proceedings of the IEEE*, 85(1):6–23, 1997.
- [78] K.-W. Han and C.-H. Chang. Gain margins and phase margins for control systems with adjustable parameters. *Journal of guidance, control, and dynamics*, 13(3):404–408, 1990.
- [79] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.
- [80] A. Hartman. Software and hardware testing using combinatorial covering suites. *Graph theory, combinatorics and algorithms*, 34:237–266, 2005.
- [81] J. C. Hayward. Near-miss determination through use of a scale of danger. *Highway Research Record*, (384), 1972.
- [82] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What’s decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998.
- [83] T. A. Henzinger and J.-F. Raskin. Robust undecidability of timed and hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 145–159. Springer, 2000.

- [84] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005.
- [85] G. Hinton, N. Srivastava, and K. Swersky. Neural networks for machine learning, lecture 6a: Overview of mini-batch gradient descent. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Accessed: 2019-03-13.
- [86] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun. Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In *American Control Conference*, pages 2296–2301, 2007.
- [87] B. Hoxha, H. Bach, H. Abbas, A. Dokhanchi, Y. Kobayashi, and G. Fainekos. Towards formal specification visualization for testing and monitoring of cyber-physical systems. In *International Workshop on Design and Implementation of Formal Tools and Systems*, 2014.
- [88] B. Hoxha, N. Mavridis, and G. Fainekos. ViSpec: A graphical tool for elicitation of MTL requirements. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3486–3492. IEEE, 2015.
- [89] E. Huang, J. Xu, S. Zhang, and C.-H. Chen. Multi-fidelity model integration for engineering design. *Procedia Computer Science*, 44:336–344, 2015.
- [90] M. Huang, H. Nakada, S. Polavarapu, R. Choroszuca, K. Butts, and I. Kolmanovsky. Towards combining nonlinear and predictive control of diesel engines. In *American Control Conference (ACC), 2013*, pages 2846–2853. IEEE, 2013.
- [91] C. Igel, T. Suttorp, and N. Hansen. A computational efficient covariance matrix update and a $(1+1)$ -cma for evolution strategies. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 453–460. ACM, 2006.
- [92] International Organization for Standardization. ISO Functional Safety. https://en.wikipedia.org/wiki/ISO_26262. Accessed: 2018-02-28.
- [93] L. Jaillet, J. Cortés, and T. Siméon. Transition-based RRT for path planning in continuous cost spaces. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2145–2150. IEEE, 2008.
- [94] V. John, K. Yoneda, B. Qi, Z. Liu, and S. Mita. Traffic light recognition in varying illumination using deep learning and saliency map. In *Proceedings of the 2014 IEEE 17th International Conference on Intelligent Transportation Systems (ITSC)*, 2014.

- [95] A. A. Julius, G. E. Fainekos, M. Anand, I. Lee, and G. J. Pappas. Robust test generation and coverage for hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 329–342. Springer, 2007.
- [96] J. Kapinski, J. Deshmukh, X. Jin, H. Ito, and K. Butts. Simulation-guided approaches for verification of automotive powertrain control systems. In *2015 American Control Conference (ACC)*, pages 4086–4095. IEEE, 2015.
- [97] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts. Simulation-based approaches for verification of embedded control systems: An overview of traditional and advanced modeling, testing, and verification techniques. *IEEE Control Systems Magazine*, 36(6):45–64, 2016.
- [98] J. Kapinski, B. H. Krogh, O. Maler, and O. Stursberg. On systematic simulation of open continuous systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 283–297. Springer, 2003.
- [99] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.
- [100] J. Kennedy. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2010.
- [101] B. Kim, A. Jarandikar, J. Shum, S. Shiraishi, and M. Yamaura. The SMT-based automatic road network generation in vehicle simulation environment. In *International Conference on Embedded Software (EMSOFT)*, pages 18:1–18:10. ACM, 2016.
- [102] B. Kim, Y. Kashiba, S. Dai, and S. Shiraishi. Testing autonomous vehicle software in the virtual prototyping environment. *Embedded Systems Letters*, 9(1):5–8, 2017.
- [103] J. Kim, J. M. Esposito, and V. Kumar. An rrt-based algorithm for testing and validating multi-robot controllers. Technical report, MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA GRASP LAB, 2005.
- [104] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [105] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.
- [106] P. Koopman and M. Wagner. Challenges in autonomous vehicle testing and validation. *SAE Int. J. Trans. Safety*, 4:15–24, 04 2016.

- [107] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [108] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to combinatorial testing*. CRC press, 2013.
- [109] T. Kunz, A. Thomaz, and H. Christensen. Hierarchical rejection sampling for informed kinodynamic planning in high-dimensional spaces. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 89–96. IEEE, 2016.
- [110] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.
- [111] D. J. LeBlanc, M. Gilbert, S. Stachowski, D. Blower, C. A. C. Flannagan, S. Karamihas, and W. T. B. and Rini Sherony. Advanced surrogate target development for evaluating pre-collision systems. In *23rd Enhanced Safety of Vehicles Conferences*, 2013.
- [112] J. Lehman and K. O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.
- [113] H. Li, Y. Li, L. H. Lee, E. P. Chew, G. Pedrielli, and C.-H. Chen. Multi-objective multi-fidelity optimization with ordinal transformation and optimal sampling. In *Proceedings of the Winter Simulation Conference*, pages 3737–3748. IEEE Press, 2015.
- [114] S. Li, K. Li, R. Rajamani, and J. Wang. Model predictive multi-objective vehicular adaptive cruise control. *IEEE Transactions on Control Systems Technology*, 19(3):556–566, 2011.
- [115] P. Liu and U. Ozguner. Predictive control of a vehicle convoy considering lane change behavior of the preceding vehicle. In *American Control Conference (ACC), 2015*, pages 4374–4379. IEEE, 2015.
- [116] S. M. Loos, A. Platzer, and L. Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *International Symposium on Formal Methods*, pages 42–56. Springer, 2011.
- [117] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.
- [118] MATLAB. *version 9.0.0 (R2016a)*. The MathWorks Inc., Natick, Massachusetts, 2016.

- [119] M. Maurer, J. C. Gerdes, B. Lenz, H. Winner, et al. *Autonomous driving*, volume 10. 2016.
- [120] M. Maurer and H. Winner. *Automotive systems engineering*. Springer, 2013.
- [121] Mechanical Simulation. CarSim, 2016.
- [122] A. Melikyan, N. Hovakimyan, and Y. Ikeda. Dynamic programming approach to a minimum distance optimal control problem. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, 2003.
- [123] O. Michel. Cyberbotics ltd. Webots: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.
- [124] O. Michel. WebotsTM: Professional mobile robot simulation. *arXiv preprint cs/0412052*, 2004.
- [125] F. A. Mullakkal-Babu, M. Wang, B. van Arem, and R. Happee. Design and analysis of full range adaptive cruise control with integrated collision avoidance strategy. In *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on*. IEEE, 2016.
- [126] B. Nagy and A. Kelly. Trajectory generation for car-like robots using cubic curvature polynomials. *Field and Service Robots*, 11, 2001.
- [127] W. G. Najm, R. Ranganathan, G. Srinivasan, J. S. Toma, E. Swanson, and A. B. D. Smith. Description of light-vehicle pre-crash scenarios for safety applications based on vehicle-to-vehicle communications. Technical report, DOT HS 811 731, 2013.
- [128] E. Narby. Modeling and estimation of dynamic tire properties. Master’s thesis, Linköpings Universitet, Linköping, 2006.
- [129] National Highway Traffic Safety Administration. NHTSA Federal Automated Vehicles Policy. <https://www.transportation.gov/AV/federal-automated-vehicles-policy-september-2016>. Accessed: 2018-02-28.
- [130] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.
- [131] Y. Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

- [132] T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancić, A. Gupta, and G. J. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 211–220. ACM, 2010.
- [133] Nvidia. Self-driving safety report, 2018.
- [134] M. O’Kelly, H. Abbas, S. Gao, S. Shiraishi, S. Kato, and R. Mangharam. APEX: Autonomous vehicle plan verification and execution. In *SAE World Congress*, 2016.
- [135] M. O’Kelly, H. Abbas, and R. Mangharam. Computer-aided design for safe autonomous vehicles. In *2017 Resilience Week (RWS)*, 2017.
- [136] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *ACM Asia Conference on Computer and Communications Security*. ACM, 2017.
- [137] Pegasus Research Project. Pegasus. <https://www.pegasusprojekt.de/en/home>. Accessed: 2018-02-28.
- [138] A. Petrovskaya and S. Thrun. Model based vehicle detection and tracking for autonomous urban driving. *Autonomous Robots*, 26(2):123–139, Apr 2009.
- [139] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Falsification of LTL safety properties in hybrid systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 368–382. Springer, 2009.
- [140] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Hybrid systems: from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*, 34(2):157–182, 2009.
- [141] A. Platzer. *Logical analysis of hybrid systems: proving theorems for complex dynamics*. Springer Science & Business Media, 2010.
- [142] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In *International Joint Conference on Automated Reasoning*, pages 171–178. Springer, 2008.
- [143] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
- [144] D. A. Pomerleau. Alvin: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems*, pages 305–313, 1989.
- [145] N. Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

- [146] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.
- [147] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.
- [148] A. Rizaldi, F. Immler, B. Schürmann, and M. Althoff. A formally verified motion planner for autonomous vehicles. In *International Symposium on Automated Technology for Verification and Analysis*, pages 75–90. Springer, 2018.
- [149] A. Rizk, G. Batt, F. Fages, and S. Soliman. On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In *International Conference on Computational Methods in Systems Biology*, pages 251–268. Springer, 2008.
- [150] R. Y. Rubinstein and D. P. Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer Science & Business Media, 2013.
- [151] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [152] SAE. Guidelines for safe on-road testing of sae level 3, 4, and 5 prototype automated driving systems (ADS), document j3018_201503. https://www.sae.org/standards/content/j3018_201503/. Accessed: 2018-02-28.
- [153] SAE International. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems, 2014.
- [154] S. Sankaranarayanan and G. Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *ACM International Conference on Hybrid Systems: Computation and Control*, 2012.
- [155] W. H. Schilders, H. A. Van der Vorst, and J. Rommes. *Model order reduction: theory, research aspects and applications*, volume 13. Springer, 2008.
- [156] R. Schubert, E. Richter, and G. Wanielik. Comparison and evaluation of advanced motion models for vehicle tracking. In *2008 11th International Conference on Information Fusion*, pages 1–6, June 2008.
- [157] S. A. Seshia, D. Sadigh, and S. S. Sastry. Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514*, 2016.

- [158] L. F. Shampine and S. Thompson. Solving DDEs in Matlab. *Applied Numerical Mathematics*, 37(4):441–458, 2001.
- [159] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.
- [160] J. E. Stellet, M. R. Zofka, J. Schumacher, T. Schamm, F. Niewels, and J. M. Zöllner. Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions. *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 1455–1462, 2015.
- [161] M. Strickland, G. Fainekos, and H. B. Amor. Deep predictive models for collision risk assessment in autonomous driving. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.
- [162] R. Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.
- [163] P. Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.
- [164] The European New Car Assessment Programme. Euro NCAP. <https://www.euroncap.com/en>. Accessed: 2018-02-28.
- [165] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. MIT press, 2005.
- [166] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *40th International Conference on Software Engineering (ICSE)*, 2018.
- [167] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. Sim-ATAV: Open-Source Simulation-based Adversarial Test Generation Framework for Autonomous Vehicles, 2018.
- [168] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. Sim-ATAV: Simulation-based adversarial testing framework for autonomous vehicles. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 283–284. ACM, 2018.
- [169] C. E. Tuncali, G. Fainekos, H. Ito, and J. Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *IEEE Intelligent Vehicles Symposium (IV)*, 2018.

- [170] C. E. Tuncali, G. Fainekos, and Y.-H. Lee. Automatic parallelization of simulink models for multi-core architectures. In *2015 IEEE 17th International Conference on Embedded Software and Systems (ICESS)*, pages 964–971. IEEE, 2015.
- [171] C. E. Tuncali, B. Hoxha, G. Ding, G. Fainekos, and S. Sankaranarayanan. Experience report: Application of falsification methods on the UxAS system. In *NASA Formal Methods Symposium*, pages 452–459. Springer, 2018.
- [172] C. E. Tuncali, J. Kapinski, H. Ito, and J. V. Deshmukh. Reasoning about safety of learning-enabled components in autonomous cyber-physical systems. *IEEE 56th Design Automation Conference (DAC)*, 2018.
- [173] C. E. Tuncali, T. P. Pavlic, and G. Fainekos. Utilizing S-TaLiRo as an automatic test generation framework for autonomous vehicles. In *IEEE Intelligent Transportation Systems Conference*, 2016.
- [174] C. E. Tuncali, S. Yaghoubi, T. P. Pavlic, and G. Fainekos. Functional gradient descent optimization for automatic test case generation for vehicle controllers. In *IEEE International Conference on Automation Science and Engineering*, 2017.
- [175] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.
- [176] W. Wachenfeld, P. Junietz, R. Wenzel, and H. Winner. The worst-time-to-collision metric for situation identification. In *2016 IEEE Intelligent Vehicles Symposium, IV 2016, Gotenburg, Sweden, June 19-22, 2016*, pages 729–734, 2016.
- [177] G. Walsh, D. Tilbury, S. Sastry, R. Murray, and J.-P. Laumond. Stabilization of trajectories for systems with nonholonomic constraints. *Automatic Control, IEEE Transactions on*, 39(1):216–222, 1994.
- [178] J. Ward, G. Agamennoni, S. Worrall, and E. Nebot. Vehicle collision probability calculation for general traffic scenarios under uncertainty. In *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*, pages 986–992. IEEE, 2014.
- [179] Waymo. Waymo safety report: On the road to fully self-driving, 2017.
- [180] A. K. Winn and A. A. Julius. Optimization of human generated trajectories for safety controller synthesis. In *American Control Conference (ACC)*, pages 4374–4379. IEEE, 2013.

- [181] H. Winner, S. Hakuli, F. Lotz, and C. Singer. *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Springer, 2015.
- [182] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *arXiv preprint arXiv:1612.01051*, 2016.
- [183] J. Xu, S. Zhang, E. Huang, C.-H. Chen, L. H. Lee, and N. Celik. An ordinal transformation framework for multi-fidelity simulation optimization. In *Automation Science and Engineering (CASE), 2014 IEEE International Conference on*, pages 385–390. IEEE, 2014.
- [184] J. Xu, S. Zhang, E. Huang, C.-H. Chen, L. H. Lee, and N. Celik. MO2TOS: Multi-fidelity optimization with ordinal transformation and optimal sampling. *Asia-Pacific Journal of Operational Research*, 33(03), 2016.
- [185] S. Yaghoubi and G. Fainekos. Hybrid approximate gradient and stochastic descent for falsification of nonlinear systems. In *American Control Conference (ACC)*, 2017.
- [186] M. D. Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.
- [187] K. Zhang, J. Sprinkle, and R. G. Sanfelice. A hybrid model predictive controller for path planning and path following. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, pages 139–148. ACM, 2015.
- [188] D. Zhao, Y. Guo, and Y. J. Jia. TrafficNet: An open naturalistic driving scenario library. In *20th IEEE International Conference on Intelligent Transportation Systems, ITSC*, 2017.
- [189] D. Zhao, H. Lam, H. Peng, S. Bao, D. J. LeBlanc, K. Nobukawa, and C. S. Pan. Accelerated evaluation of automated vehicles safety in lane-change scenarios based on importance sampling techniques. *IEEE Transactions on Intelligent Transportation Systems*, 18(3):595–607, 2017.
- [190] M. R. Zofka, M. Essinger, T. Fleck, R. Kohlhaas, and J. M. Zöllner. The sleepwalker framework: Verification and validation of autonomous vehicles by mixed reality lidar stimulation. In *SIMPAR*, pages 151–157. IEEE, 2018.
- [191] A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski. Multiple shooting, cegar-based falsification for hybrid systems. In *Proceedings of the 14th International Conference on Embedded Software*, page 5. ACM, 2014.