Autonomous System Testing: Incorporating Physical Environments and Physical Semantics.

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Carl Hildebrandt

April 2023

Acknowledgements

While some may argue that pursuing a Ph.D. is an isolating experience, I believe it takes a village to complete the journey. Therefore, although I have listed a few individuals below who have been instrumental, I recognize that this does not encompass everyone who contributed. To those not mentioned, I extend my heartfelt thanks.

First and foremost, I would like to express my profound gratitude to my advisor, Sebastian. Thank you for taking a chance on me, despite my less-than-stellar test scores. Thank you for the countless hours spent guiding me, questioning me, listening to my ideas, refining them, and reminding me to step back when I was lost in the details. I want you to know that while you set out only to change the way I view and approach research, you have fundamentally shaped how I think about life and the world—a gift I could never repay. Although this marks the end of our work together, your influence will remain with me for the rest of my life, and for that, I am forever grateful!

I also want to thank my family. I would like to thank my dad, Deon, who I am sure would have loved to read this. My mom, Diane, who has been a rock in my life - someone I can complain to, ask advice from, and always rely on (except to get me to a sporting event on time). You are an inspiration to me, and I only hope I can be as amazing as you in the years to come. To my wife, Riley, thank you for your unwavering patience through the many highs and lows. Without your understanding, love, support, and ability to make me laugh no matter the circumstance, I would have never been able to finish the last year and a half of my Ph.D. - especially since I have been saying this is the last year and a half for four years. To my brother, Timothy, you have grown into an

incredible human. Thank you for not only being my brother but also my friend. To my stepfather, Jannie, thank you for never making me feel like you were my stepfather, but rather just my other dad. To my sister-in-law, Megan, I couldn't have dreamed of a better sister. I can't wait to be a part of your life, sharing in your endless energy and adventurous spirit. To my parents-in-law, Sue and Robert, thank you for accepting me into your home like a son. Your love and kindness to each other is something Riley and I strive for every day.

Next, I want to thank the people I have worked with. Bobby, thank you for your level-headed approach to everything, your kindness, and your guidance. You introduced me to the world outside academia, and my only regret is that I didn't get to work with you longer. To Matt, thank you for being the yin to Sebastian's yang. The LESS Lab has been a wonderful place to work, and it would not be the place it is without you. To the people of the LESS and Nimbus labs: To David and Ajay, I have no idea how you know so much about Python and robotics, respectively. Thank you for always being willing to sit down and help me with the countless problems I have encountered on this journey. To Trey and Will, not only for answering countless questions but also for somehow knowing everything there is to know about the most arbitrary facets of life; you made me look forward to coming into the lab. To Felipe, thank you for reminding me to always be enthusiastic, despite, in your words, "my old age". Thank you to the countless other lab mates who have influenced my life, both big and small. Thank you to Mitch, John-Paul, Meriel, Soneya, Dong, Adam (a.k.a. Commander P-Rad), Pedro, Chandima, Chris, Nusrat, Nick, Michael, Mira, and Rory. Thank you, not only for inspiring me but also for being my friend.

Next, I want to thank some of my lifelong friends who, despite the inconvenience of me moving to the US, have been willing to put up with time zones and many phone calls to be a part of my life. Thank you to Struan. Your perseverance and ability to set your mind on something and achieve it are unparalleled. Thank you for reminding me that how the world views you is in no way correlated to both who you are and what you can achieve. To James, the man I have never won a bet against, there is so much I can say here. Thank you for sharing your Ph.D. journey with me and for reminding me that even someone as smart and brilliant as you, experiences both highs and lows. Your empathy, kindness, and willingness to share and listen have helped me handle the pressures of

my own journey with a lighter heart. Thank you for inspiring me every day to be a better person, for your ability to listen to the worst of me, and for reminding me of the best parts of me. Most importantly, thank you for always putting up with my terrible book and movie recommendations, and in turn, actually giving me good ones. To Kai, thank you for consistently being down to play video games and chat into the late hours of the night, which, while it may seem like not much, made this journey just that much easier. To Dux, our lives only intersected for a short four years, but in that time, you shaped a part of who I am today. Thank you for always reminding me not to take life too seriously, to accept our flaws, and to remember that it will always work out. Finally, I want to thank all my friends at UVA, with a special place for the Club Field Hockey team. You accepted me into the team and gave me a place outside the lab. Each of you is so talented and wonderful, with such bright futures; I am glad I got to meet each and every one of you, and I will miss you all.

Finally I want to thank my Ph.D. committee. Bobby, Matt, Seongkook, Nicola, and Sebastian, thank you for both working with me, and taking the time to read over my contribution to autonomous system testing, which I have been working on for the last six years. Despite this being the tangible evidence of such a process, my only goal when starting was to better understand how to tackle some of the world's hardest problems, so that I could make the world, ever so slightly, a better place. Something that each of you does every single day.

Abstract

As the integration of autonomous systems becomes increasingly common in our everyday lives, their shortcomings and failures become more apparent. Therefore, rigorous validation to ensure their safety and reliability is paramount. Since autonomous systems behavior is predominantly driven by software, and software validation has achieved significant success in validating applications billions of people use today, it seems natural to attempt to apply current software validation to autonomous systems. Such application, however, requires overcoming two key challenges introduced by the differences between traditional software and autonomous systems, namely the *physical environment* and the systems *physical semantics*. Without considering these differences, traditional software testing techniques struggle to cope with a large unbounded input space and to effectively target areas of the software that drive the behaviors of the autonomous system. This work introduces techniques grounded in traditional software analysis that overcome these challenges spanning the entire testing pipeline: test generation, test execution, and test adequacy assessment.

In the area of Test Generation, I investigated techniques to produce tests based on a vehicle's kinematics to ensure they aligned with the *physical semantics* of the autonomous system, all while using parametrizable scoring models to identify tests that stress an autonomous system. Moreover, I leveraged a vast array of existing sensor data from real-world *physical environments* to identify performance discrepancies across different versions of an autonomous system. The sensor data that yielded discrepancies were then compared against the autonomous systems Operational Design Domain to determine their relevance.

In Test Execution, I have devised a mixed-reality strategy that bridges the gap between simu-

lation and real-world testing. Recognizing that real-world testing, while ideal, is often impractical, hazardous, and expensive, my approach integrates virtual elements into real *physical environments*. This allows for validating performance and safety while reducing both cost and time. Additionally, I designed a haptic suit for drones, enabling us to test the *physical semantics* of a drone by applying forces to the drone in the real world.

Regarding Test Coverage, I created Physical Coverage, one of the first coverage metrics for autonomous systems, which considers both the *physical environment* and *physical semantics* of the autonomous system. Utilizing physical reachability analysis and geometric vectorization, this metric offers a quantifiable measure of test suite effectiveness. It has proven instrumental in identifying missing scenarios and redundant tests in datasets such as Waymo's Open Perception dataset.

By addressing these challenges across the entire testing pipeline, this dissertation takes a significant step toward creating safer and more reliable autonomous systems.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

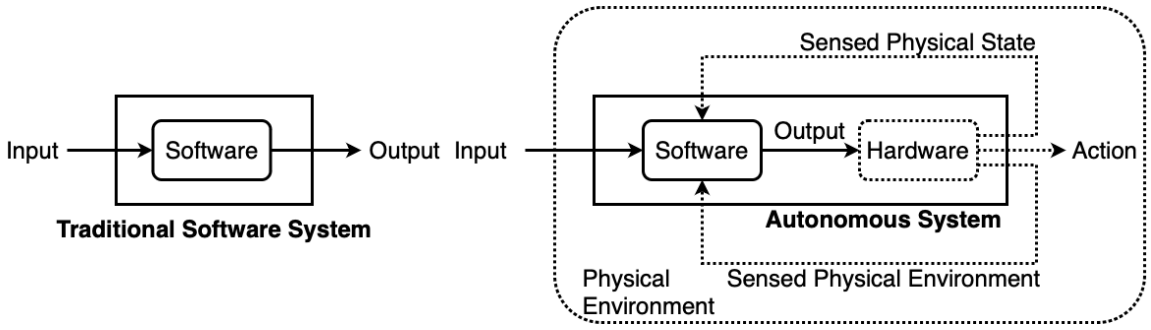
Autonomous systems are increasingly being adopted into society, reflecting a significant shift toward automation [236]. One key area of autonomous systems development is advanced driver-assistance systems, which are now commonly sighted on our roads [369, 370, 76, 263, 84]. However, this trend extends beyond ground transportation. In Switzerland, the national post office has experimented with advanced unmanned aerial vehicles for delivery services [337], and in Rwanda, fixed-wing autonomous aircraft play a crucial role in the distribution of blood and medical supplies throughout the country [408]. In aquatic environments, the European Union's INTCATCH project [157] leverages autonomous surface vehicles for mapping water quality and pollution levels [330] . Meanwhile, autonomous underwater vehicles are instrumental in creating detailed seafloor maps [229], investigating aircraft crash sites [190], and conducting deep-sea research in conditions where high pressure poses a significant risk to humans [383, 356]. The adoption rate of autonomous systems shows that soon they will be the norm.

As these systems integrate into our everyday lives, their operation in the real world exposes complex challenges. Recorded incidents across various applications have highlighted the risks associated with these technologies operating in a highly complex and dynamic environment [198, 364]. These incidents have resulted in the destruction of the autonomous systems, significant property damage, and the tragic loss of life [285, 160, 47, 243, 156, 186]. It is becoming increasingly evident that as we

move towards a future heavily reliant on autonomous systems, we need comprehensive approaches to detect and prevent faults that are general enough to be applied across a diverse array of autonomous systems.

Autonomous systems are composed of sophisticated hardware and cutting-edge software. On the hardware side, there have been notable improvements in sensor technologies [10, 75], the miniaturization of hardware [238, 234], and significant increases in computational power [250, 48]. Such advancements have led to enhanced environment analysis capabilities, improved efficiency, and faster processing times. Yet, the landscape of software has seen even more profound advancements, spurred by the leaps in computational capacity. This evolution has enabled the real-time execution of extremely complex software, capable of processing vast amounts of data. Central to these software advancements are fields such as artificial intelligence and machine learning [254, 4], cloud computing and big data [42], advancements in algorithms and control [388, 177], and higher fidelity simulations [64, 80]. These advancements have greatly enhanced the performance, and functionality of autonomous systems, now equipped to make decisions with minimal human input. Although hardware has seen its share of progress, it is the software innovations that have emerged as the pivotal component driving the autonomous behaviors of these systems.

Given this premise, focusing on software validation emerges as a key area for detecting and preventing faults in autonomous systems. Moreover, adopting such an approach can leverage advancements from decades of research and development in software validation, where significant progress has been made in foundational areas. Consider a traditional and somewhat simplified testing pipeline that starts with test generation, executes those tests, and then performs test adequacy to determine how well a system has been tested. Looking into each of these stages, there is already a large existing body of work from which we could draw inspiration from or modify to better test autonomous systems. For example, in test generation, techniques have evolved to automatically create test cases that cover a wide range of scenarios [299, 245, 17, 28], significantly reducing manual effort and increasing the comprehensiveness of testing. In test execution, innovations have improved the efficiency and automation of running tests [279, 77], allowing for rapid identification and rectification of faults. Finally, test adequacy has seen advancements in metrics and methods to assess the extent and effec-



(a) Traditional Software System. (b) Autonomous System.

Figure 1.1: Contrasting traditional software systems with autonomous systems. Dotted lines indicate differences

tiveness of testing [389], ensuring that tests are capable of uncovering potential issues. Throughout these areas, these techniques have demonstrated effectiveness across a diverse range of applications, underscoring their potential value in enhancing the reliability and safety of autonomous systems.

However, the direct application of current software validation methodologies to autonomous systems is challenging [237, 197, 109], primarily due to fundamental differences between traditional software systems and autonomous systems. To understand these differences, let us first examine a traditional software system as depicted in Figure 1.1a. A traditional software system is typically a transformational system accepting inputs, performing transformations on them to produce outputs. These systems primarily perform input/output operations, occasionally prompting a user for additional information.

In contrast, autonomous systems are closer to sophisticated reactive systems [126], as shown in Figure 1.1b. They operate within a *physical environment* that encompasses the entire system. This environment is not only complex and dynamic but also governed by physical laws and potentially shared with humans, amplifying the consequences of any failure. Additionally, while autonomous systems consume input, their output is not the final step. The output is consumed by hardware that inherently possesses *physical semantics*. This hardware enables the autonomous system to perform actions that not only change its physical state but can also potentially alter the physical environment itself. Furthermore, the autonomous system hardware generates two critical types of

feedback: a representation of the system's sensed physical state and a representation of its sensed physical environment. The former potentially includes approximations for example, the system's position, velocity, and orientation — elements deeply intertwined with its physical semantics. The latter might consist, for example, of visual data (like RGB values) or spatial data (such as point clouds), offering an approximation of the surrounding physical environment. These complex feedback loops have a critical influence on the software's output and system actions.

However, traditional software validation methodologies were not designed to account for the nuanced complexities of the *physical environment* and *physical semantics* inherent in autonomous systems. As a result, direct application of these methodologies to autonomous systems might overlook significant aspects of their operation, potentially including behaviors that could be unsafe.

To address these challenges, we propose adapting techniques from traditional software testing, taking into account both the *physical environment* and *physical semantics* of autonomous systems. Through this, this dissertation aims to show cost-effective ways to overcome these two challenges and provide a set of techniques as starting points for validating autonomous systems.

1.1 Challenges of Validating Autonomous System

Building on the differences between autonomous systems and traditional software [237, 197, 109], this section delves deeper into the unique complexities and challenges associated with the physical environments and physical semantics of autonomous systems.

1.1.1 Physical Environments

A fundamental difference between traditional software and autonomous systems is the physical environment, which presents two unique challenges. The first challenge is the sheer size of real physical environments. Traditional software testing techniques often rely on the ability to effectively abstract the input space, using models or symbolic representations to manage their complexity. The input space for autonomous systems are sensor readings which offer an abstraction of the environment. While the number of possible sensor readings is finite in theory, the combination

of different sensor inputs, their possible values, and their time-dependent nature makes the space of possible sensor readings overwhelmingly large. For example, consider trying to enumerate all possible camera readings for a single Full HD camera at any given point in time. A Full HD camera has a total of $1920 \times 1080 = 2,073,600$ pixels, where each RGB pixel has three channels ranging between 0-255. That means each pixel could be one of $256^3 = 16,777,216$ values, making for a total of $16,777,216^{2,073,600}$ possible image combinations, more images than there are estimated atoms in the observable universe [282]. The size of this input space is something that traditional software testing techniques are ill-equipped to handle.

The second challenge the environment poses is that it is governed by a set of physical laws. Specific scenarios that might appear in a purely theoretical enumeration of sensor data are physically infeasible. For instance, a tree cannot grow in mid-air without any contact to the ground. Despite this, a long tail of extraordinary scenarios exists, which are rare but feasible. For example, a tree might momentarily be airborne due to an explosion occurring beneath it. Determining the boundary between feasible and infeasible scenarios is a non-trivial task.

This enormous space of physical environments, and the inability to determine feasible from infeasible scenarios, make the direct application of traditional software analysis insufficient.

1.1.2 Physical Semantics

Another fundamental difference lies in the physical semantics of autonomous systems, which introduce two additional challenges. The first challenge is related to the system state. In the system's software, this state can be represented as variable-value pairs in memory, but in autonomous systems some of these variables are implicitly tied to physical quantities in the real world. This connection means that these variables are subject to physical limits, temporal dynamics, interdependencies, and specific units. For instance, while a car's velocity and acceleration are influenced by its motor speed and torque, a car lacks the capability to directly control its altitude. Conversely, a quadrotor's altitude is determined by its motor speed, while its velocity and acceleration are influenced by its pose. Despite these differences, in both cases, motor speed, velocity, acceleration, and altitude are stored as variables in the code. Each variable carries an implicit type, has implicit physical units, and is

limited by a range that reflects the physical capabilities. The interplay between these variables and their constraints often becomes obscured or neglected when viewed solely as software variables.

The second challenge relates to the gap between the software and physical semantics of the system to sense and change its physical state. Abstracting these two processes for use in traditional software validation is often difficult, considering the inherent uncertainty in these processes. Sensor data is often noisy, meaning that the system must combine multiple sensor readings using advanced state estimation techniques to simply understand its current state. Additionally, precise actuation in the real world can be extremely challenging, given the myriad factors and forces dictating the final result. For example, a quadrotor generally uses multiple sensors, such as a barometer, gyroscope, accelerator, magnetometer, and GPS, to estimate its current pose in the physical world. Similarly, a car wanting to stop, which may be simply setting velocity to zero in software, could result in a variety of outcomes in the physical world, such as locking wheels and skidding, resulting in a significantly different velocity reduction than expected by the software or its validation techniques.

Overall, the unique challenges introduced by the physical semantics of autonomous systems make testing them significantly more complex, underscoring the need for innovative approaches that specifically target these characteristics of the system.

1.2 Outline and Contribution

This dissertation advances the field of autonomous systems testing, significantly extending traditional software validation methods with unique insights pertinent to autonomous systems. The guiding principle of this research has been the recognition that the behavior of autonomous systems is intricately shaped by their *physical environment* and *physical semantics*. This realization necessitated the incorporation of these critical elements into standard testing strategies, thereby enhancing their effectiveness and relevance.

This dissertation encompasses a comprehensive view of the testing process. The work is structured around a conceptual three-stage testing pipeline. This pipeline encompasses test generation, test execution, and test adequacy, forming a framework for autonomous system testing. In the

Table 1.1: An overview of the completed work, proposed work, and which significant challenge it tackles.

Section	Approach	Physical Environment	Physical Semantics	Status
Generation	Feasible and Stressful Trajectory Generation		✓	Completed [138]
	Differential Testing with Real Data	✓		Completed Under Review
Execution	World-in-the-Loop Simulation	✓		Completed [135]
	Mimicking Real Forces on a Drone Through a Haptic Suit		✓	Completed [143]
Adequacy	Physical Coverage	✓	✓	Completed [140]

test generation phase, we systematically design test cases to explore the operational space of the autonomous system, aiming to expose potential failures under varied conditions. During test execution, these test cases are applied to the autonomous system in a controlled environment to observe its responses and behavior under the test's conditions. Lastly, the test adequacy phase evaluates the thoroughness of the testing process, to assess to what extent the range of potential behaviors and conditions have been explored.

Focusing on test generation, detailed in Chapter 3, the discussion is centered around two key contributions. First, *Feasible and Stressful Trajectory Generation* [138], focuses on generating tests which take the physical semantics of autonomous systems into account. This work employs the kinematic and dynamic models of an autonomous system (described in Section 2.3.3) to create trajectories that align with the system's physical semantics. It utilizes parameterizable scoring models of autonomous system stress, such as trajectory deviation, which are either defined by users or learned through existing data, to identify the most stressful trajectories. Second, *Differential Testing with Real Data* (portion under review, see ODD-DiLLMma below) addresses the challenge of integrating the physical environment into test generation. At its core, this method identifies failure-inducing test cases from the massive amounts of real-world sensor data that have previously been recorded in actual physical environments. Our approach begins with differential testing, which compares the behaviors of two or more systems expected to respond identically under the same inputs. Specifically it uses real-world data to test multiple systems against diverse scenarios, identifying behavior discrepancies that reveal potential failure inducing scenarios. However, as the data was

collected independently of the autonomous systems' operations, some scenarios identified may be outside the systems' operational design domain (ODD), which is the set of conditions the systems are designed to handle [158]. To tackle this, our strategy incorporates an ODD filtering mechanism, named ODD-diLLMma (under review), to sift out test cases falling outside the ODD. This results in the isolation of failures that are not only pertinent but also rooted in the real physical environment the autonomous system is designed to operate in.

Test execution, detailed in Chapter 4, features two key contributions. First, *World-in-the-Loop Simulation* [135] directly tackles the difficulty of replicating complex or costly physical environments. Specifically it addresses the simulation-reality gap, which causes differences in autonomous system behaviors when moving from simulation to real physical environments. Our approach creates a mixed-reality environment that merges elements of simulation with aspects of the real world. This allows developers to vary the degree of real and simulated environment included in during the testing execution phase, allowing developers to identify discrepancies in behaviors in safer more cost effective settings. Second, *Mimicking Real Forces on a Drone Through a Haptic Suit* [143], introduces a both a device and software system designed to apply real world forces onto a drone, allowing developers to explore the physical semantics of a drone in the real world, while minimizing costs, setup complexity, and space requirements. Uniquely, this device is mounted directly onto the drone, eliminating the need for tethers or any external equipment. It is capable of generating a diverse array of programmable forces that accurately mimic real-world scenarios, including wind resistance, added weight, or the dynamics of a swinging pendulum. By accurately synthesizing these environmental forces, our device not only simplifies the test execution process but also allows for a more complete exploration of the drone's physical semantics.

Test adequacy, in Chapter 5, introduces Physical Coverage [140], a novel approach for measuring the adequacy of test suites for autonomous systems. This approach integrates both the physical environment and its physical semantics into its adequacy criteria. It begins by identifying the portions of the physical environment most relevant to the autonomous system, based on its current state. This is done by utilizing kinematic and dynamic models to determine all possible future states of the system over a defined time horizon. These future states allow the approach to identify portions of the envi-

ronment that may be useful in the future, while simultaneously excluding regions the autonomous system could never interact with. The reduced environment is then geometrically approximated, resulting in an abstraction that represents an autonomous system's current environment-state pair. These abstractions serve as a basis for computing the adequacy of test suites, providing a robust and holistic assessment of test suite effectiveness.

The methodologies developed in this dissertation effectively address key challenges posed by both *physical environments* and *physical semantics* in the validation of autonomous systems. The successful completion of the proposed work has resulted in a suite of innovative and effective techniques that enhance each stage of the testing pipeline, significantly improving the validation process for autonomous systems. This work makes substantial contributions to the field of autonomous system testing as follows:

- **Conceptual Contributions: Incorporating Physical Environments and Physical Semantics into Autonomous System Validation**. This work underscores the pivotal roles of the *physical environment* and *physical semantics* in influencing autonomous systems' behavior. It presents 5 innovative strategies for incorporating both of these elements into the test generation, test execution, and test adequacy, showcasing validation across all phases of the autonomous system testing pipeline.

- **Empirical Studies: Demonstrating the Impact of Physical Semantics and Environments in Autonomous System Testing**. Our research presents evidence of the advantages of integrating *physical semantics* and *physical environments* into autonomous system validation. For example, our trajectory generation method creates a greater number of physically feasible trajectories that were on average 55.9% more stressful than trajectories that did not use our approach, highlighting the importance of physical semantics. Differential Testing on existing data reveals that 9.8% of sensor readings given to three commercial autonomous driving systems resulted in steering differences of 10 degrees or more. Additionally, we demonstrate that this approach can identify existing continuous streams of sensor readings of up to 2.5 seconds, which are relevant with respect to the systems ODD, that consistently produce steering

differences of more than 45 degrees. This emphasizes the potential value of data captured in real physical environments. Our mixed-reality framework preemptively identifies failures across 6 distinct scenarios before real-world execution, showcasing the advantages of integrating simulated with physical environments. Similarly, using our haptic suit for drone testing provides a cost-effective method to explore a drone's physical semantics, allowing the replication of 5 real-world scenarios with significantly reduced time and cost. Finally, our physical coverage metric, accounting for both physical semantics and environments, surpasses traditional coverage metrics by producing equivalence classes that are up to 57% more consistent than currently used metrics. Collectively, these findings highlight the critical role of both the *physical environment* and *physical semantics* in testing autonomous systems.

- **Implementation Contributions: Development of Tools and Artifacts.** This research has generated a suite of tools and artifacts to support and validate our methods, each made publicly available to broaden their application and facilitate further study. One artifact, received a distinguished artifact award at ISSTA 2020 [138, 139], highlighting our work's practical utility and significance. Hosted on publicly available platforms like GitHub and Zenodo, these tools are not only comprehensive and easily accessible but also promote wider adoption and research [139, 142, 136, 144, 141]. Demonstrations on real-world commercial autonomous systems, such as the Parrot Anafi Quadrotor [275], Waymo's Autonomous Vehicle (via their Open Perception Dataset) [333], and comma.ai's openpilot Automatic Lane Centering (ALC) [66], showcase our approaches applicability to real-world problems. This combination of tools and real-world demonstrations underlines our contributions' broader impact and relevance, bridging academic innovation with practical application and pushing forward the field of autonomous system validation.

Chapter 2

Background

This section lays the groundwork for understanding this dissertation. First, to ensure clarity and consistency in the subsequent chapters, Section 2.1 defines the nomenclature that will be used for the remainder of this dissertation. Next, Section 2.2 provides a comprehensive overview of the current state of the art in testing pipelines for autonomous systems. It discusses the key components, namely: test generation in Section 2.2.1, test execution in Section 2.2.2, and test adequacy in Section 2.2.3. Then, we delve into the mathematics underpinning the physical semantics of systems, examining system movement and physical modeling in Section 2.3. Specifically, we discuss a system's state in Section 2.3.1, kinematic and dynamic (KD) models in Section 2.3.2, and the prediction and determination of reachable sets in Section 2.3.3.

2.1 Nomenclature

This nomenclature provides an overview of the terminology used in subsequent chapters.

- $\mathcal{W}$: The world, which encompasses all possible environment configurations in which an autonomous system might operate.
- w: A scenario, which is one possible environment configuration from the world, $w \in \mathcal{W}$.

- c: An individual scene from the scenario, such that $c \in w$ and $w = \{c_1, c_2, \ldots, c_n\}$.
- $\mathcal{S}$: The set of all possible system states.
- s: An instance of a system state, such that $s \in \mathcal{S}$.
- sen: A function representing sensors in the autonomous system, designed to perceive some real-world scene and produce an approximation of it.
- c^{sen}: Is the sensed scene, such that $sen(c) = c \pm \delta_c$, where δ_c quantifies the difference between actual and sensed scene.
- s^{sen}: Is the sensed state, produced through $sen(s) = s \pm \delta_s$, where δ_s quantifies the difference between actual and sensed state.
- a: An autonomous systems action capable of changing both c and s.
- AS: An autonomous system. They have the ability to consume sensor readings and output actions $a = AS(c^{sen}, s^{sen})$.
- b: A behavior is a sequence of autonomous systems actions $b = \langle a_1, a_2, \ldots, a_m \rangle$.
- o^{sen}: A sequence of observed sensed and scenes state pairs $o^{sen} = \langle (c_1^{sen}, s_1^{sen}), (c_2^{sen}, s_2^{sen}), \ldots, (c_m^{sen}, s_m^{sen}) \rangle$ such that $b = AS(o^{sen})$.
- $\mathcal{O}^{sen}$: is the set of all possible observable sensed scene and state pairs $o^{sen} \in \mathcal{O}^{sen}$.
- $\mathcal{B}$: Set of all possible behaviors of the autonomous system, $b \in \mathcal{B}$.
- $\mathcal{T}$: A test suite composed of individual tests, denoted as τ, where each $\tau \in \mathcal{T}$.
- $\tau_k = (input, oracle, context)$: Represents a single test, characterized by an *input*, an *oracle*, and *context*.
- *input*: We define an *input* in the broadest sense possible, encompassing explicit test inputs from a test, such as trajectories to follow, environmental factors like w, and system states s that can affect execution and testing.

- *oracle*: Takes in both an autonomous system's input, o^{sen}, and output, b, and maps them to a boolean, denoted as $oracle : (o^{sen}, b) \mapsto \mathbb{B}$. Essentially, a test *oracle* distinguishes between correct and incorrect behaviors given some input of the system under test.

- *context*: Describes how closely the scenario w and the autonomous system AS are mocked during execution. Essentially, it spans the range from point simulation, where both w and AS are extremely abstract, to real-world deployment where both w and AS are fully realized.

2.2 The Testing Pipeline

Validation seeks to build confidence that the system meets defined requirements or specifications through empirical evidence [5]. It achieves this by running a series of tests on the system under test to either identify violations or accumulate instances of compliance with the requirements. By addressing these violations and confirming compliance, developers build confidence that the system will meet the requirements in its intended deployment. This is one of the ways developers can ensure that a system will behave as expected in the real world.

Another way to think about validation is through a traditional and somewhat simplified testing pipeline, as shown in Figure 2.1. This testing pipeline has three main components. The first component is test generation, where each of the $\tau \in \mathcal{T}$ is generated. The next component involves executing the tests on, in our case, autonomous systems, collecting the output, and checking for correctness. Finally, some form of test adequacy is performed; this is a quantitative method for determining confidence in both the $\mathcal{T}$ and the range of outputs exposed. Using this information, developers can either deploy the system if they have sufficient confidence in it, or generate more tests to expose the system to previously unseen inputs or uncomputed outputs. Below we discuss

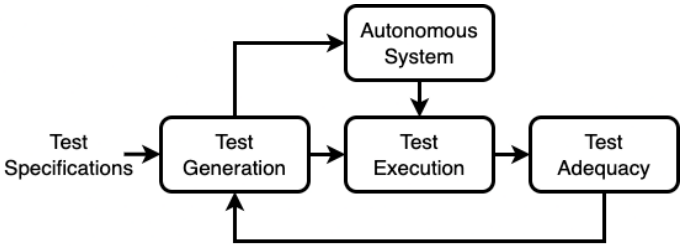


Figure 2.1: A typical testing pipeline.

each of the components in more detail.

2.2.1 Test Generation for Autonomous Systems

To create a test, developers need to consider three elements: the *input*, *oracle*, and *context*. Test generation primarily focuses on the *input* and *oracle*, as these elements often vary for each $\tau \in \mathcal{T}$. The *context* is more closely aligned with test execution and is therefore discussed later, in Section 2.2.2.

2.2.1.1 Determining a Tests Input

Generating *input* for tests, can generally be grouped into three categories: random input generation, search-based generation, and model-based generation. Random input generation uses stochastic processes to create a diverse range of *input*s. It prioritizes speed and breadth over specificity. Given the vastness of the input space, highlighted in Section 1.1, many techniques sample inputs from a specific portion of the autonomous system input space. For instance, some focus on just the scenario, using random procedural generation to create entire scenarios from scratch [252, 324, 21]. Others focus on randomly varying aspects of existing scenarios such as weather and lighting [345], objects and obstacles [93], and textures [227]. Some consider just the agents, dynamic elements in the scenario such as cars, by randomly placing them in different locations [114], or by randomly generating their trajectories [25, 350]. Random input generation techniques however are not only limited to the scenario portion of the *input*. For example, some randomly sample the initial parameters of the autonomous systems such as the velocity and acceleration [112], or randomly generating invalid commands to ensure input validation systems are working [288] Overall, these techniques emphasize exploring large portions of a specified input space quickly, in the hopes that they find an *input* which results in a failure.

Next, search-based test generation approaches leverage heuristics and optimization algorithms for targeted exploration of the input space [246]. These heuristics and algorithms enable the techniques to systematically explore the input space by focusing on areas deemed important, with the aim of discovering regions more likely to reveal failures. The idea is that these techniques can balance

the breadth provided by random test generation with the depth achieved by thoroughly exploring interesting areas. Examples of these include using search algorithms to find entire scenarios that result in autonomous vehicle crashes [3], or traffic law violations [403]. Others search only parts of the scenario, such as looking for road typologies that meet specific road curvature criteria [182], or road networks that challenge the lane-keeping capabilities of advanced driver assistance systems [154, 106]. Beyond searching for scenario-based *input*, some methods use predefined scenarios and search for autonomous system configuration files [50], system states [36], or sensor data [399], which are likely to result in collisions. Others do not require scenarios at all and search for input validation bugs, failures due to missing or incorrect validation checks, that result in erroneous control outputs [184]. Search-based test generation sometimes does not yield concrete tests but rather leans towards identifying probability distributions. For example, Scenic uses probabilistic programming languages to define and subsequently search through a space of possible scenarios [102].

While random and search-based methods define how to explore the *input* space, they don't describe what that space is. Model-based testing distinguishes itself by employing models that describe the input space to generate tests. While a vast array of model types exist, a common form of model describes either the system's expected behavior or its operating environment. For instance, models can describe how the system under test should behave for different driving styles [82], or they can outline expected behaviors in specific scenarios, such as on highways [18] or at intersections [97, 336]. Alternative models describe the behaviors of other systems or agents in a scenario; for example, the expected behaviors of other road users [308] or people in a shopping mall [19]. Models have also been applied to environmental aspects of scenarios [35], describing specific elements of the environment, such as wind and obstacles [221], or the roads and scenery [23]. Others focus on attributes of the scenario itself, such as its feasibility in the real world [105, 103]. Once a model is defined, it can be used in various ways. For example, by defining conditions or behaviors that the model should exhibit, such as reaching a particular state [266] or making a specific transition [149]. If a property is violated, the method generates a counterexample, which can be used to create a test for the real system [278, 307]. Examples of these approaches have been applied to partial models of systems such as the emergency braking system [88], the perception system [116], as well as to the

whole system [91, 220].

An alternative to generating new *input*s for testing is to employ test selection strategies on a pre-existing set of test inputs. This is useful in cases where more tests can be generated than executed, or where a set of tests already exists and developers want to prioritize them from most to least interesting. The key to test case selection is to quantitatively measure the value of each test case. This can be achieved through various methods, including model-driven and coverage-based selections. Model-driven selection approaches utilize models to inform the selection process, echoing the strategies discussed previously. These models range from those based on data from previous executions [228], simulating which obstacles are likely to be detected by the autonomous system [176], to models that describe the certainty of an autonomous system's actions [362, 98]. Conversely, coverage-based selection focuses on maximizing specific coverage metrics within the test suite [338]. This includes selecting tests based on scenarios, such as those most likely to cause the vehicle to drive on parts of the road not yet driven [153], or based on some internal software-based coverage metric, such as neuron coverage [281]. While many of these coverage metrics have yet to be specifically applied to autonomous systems, many have been proposed for Deep Neural Network (DNN) testing, a key component of many autonomous systems. Examples include extensions of neuron coverage [281], modified condition or decision coverage [334, 386], and surprise coverage [183].

2.2.1.2 Defining a Test Oracle

For a test to be useful, developers need to know if the corresponding output produced by the system under test is correct or not. This ability to distinguish correct from incorrect behavior, given some input, is referred to as the "test oracle problem" [28]. A test *oracle* is a function or procedure that distinguishes between correct and incorrect behaviors of a system under test [162, 27]. More specifically, it takes a system's input and output and maps them to a Boolean value, $oracle(\text{input}, \text{output}) \mapsto \mathbb{B}$. Adequately defining an oracle such that it can automatically perform this mapping is a challenging yet vital part of the testing process. Without an automated test oracle, a human must determine whether the observed output for a given input is correct. This severely limits the number of tests that can be performed and, in some cases, is not even feasible

due to the speed, complexity, or volume of the input and output being processed. For an oracle to automatically perform this operation, it must have the ability to compare the observed output with a known expected output. The different methods of deriving the expected output and performing the comparison can be discussed in three broad categories: specification-based oracles, derived oracles, and implicit oracles.

A formal specification of a system is a mathematically-based language that can be used to describe a system in terms of its inputs and expected outputs [134]. Test *oracles* can then use these specifications to check if the system's output meets the expected output for the given input [9, 27]. The primary issue is that defining specifications that are known to be correct, complete, and unambiguously specified is extremely challenging for autonomous systems [197]. This challenge stems from the fact that autonomous systems have complex input types, which exhibit a long tail of possible rare inputs, each associated with many potential correct and incorrect behaviors. While oracles based on specifications have had success in traditional software [298, 331], their application is limited when applied to autonomous systems. To overcome the difficulties posed by precisely defining specifications for autonomous systems, current specification-based oracles for autonomous systems primarily focus on using partial specifications. These are specifications that are either weakly defined, for example using natural language where there is room for ambiguity, or defined for only a small portion of the input or output space.

First, let us consider specifications based on natural language. While some of these specifications, such as legal road rules, may appear complete and well-defined, they inherently rely on human intuition and understanding. For instance, Rule 163 of the UK driving code states, "overtake only when it is safe and legal to do so" by, for example, "not getting too close to the vehicle you intend to overtake" [314]. This rule requires some intuitive understanding, or what is "safe", what is "legal", and what is "too close", all of which could vary from person to person. While there is work aiming to convert these properties into checkable specifications [401], they still depend on humans to make judgements about how to implement one of these properties, while also relying on some ground truth information to measure their defined property. Another potential source of these types of specifications are today's Operational Design Domain (ODD) [196]. The ODD, as defined by the SAE

standard J3016, represents "operating conditions under which a given driving automation system or feature thereof is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, and/or the requisite presence or absence of certain traffic or roadway characteristics" [1]. Techniques capable of specifying and checking small portions of such an ODD have been developed, focusing on narrow aspects like geographic location [209] or sensor-specific anomalies [65]. However, checking any given arbitrary specification against ODD specifications remains an open problem, for which we present one of the first approaches to do so later in Section 3.2. Next, let's consider partial formal specifications defined over small portions of the input/output space. Examples of this type of work include oracles that can check if a system's output velocity meets specific specifications based on sensor inputs such as obstacle distance and road inclination [289]. Other oracles verify that input service requests, such as visiting specific locations or avoiding certain areas, result in output plans that are efficient, avoid obstacles, and adhere to specified tasks [58]. Some oracles check that internal communication messages between components of a robotic system never produce outputs that result in deadlocks between modules [37] or outputs that deviate from a specified order [94]. Additionally, oracles might monitor energy consumption to ensure that swarms of robots behave in a way that does not deplete their energy or power sources prematurely [194]. Some oracles ignore the input and focus solely on the output, such as checking that the movements of a robotic arm meet certain specifications regarding the execution order and ranges of motion [217].

Another type of oracle that uses partial specifications is found in a field known as metamorphic testing. These oracles define metamorphic relations, which are partial specifications concerning the system under test in relation to multiple inputs and their expected outputs [55, 313]. Specifically, by applying a known transformation to the input and running the same system on both the original and transformed inputs, we can detect failures by comparing the outputs to a corresponding known transformation function. When applied to autonomous systems [404], typical transformations define sets of changes that, when applied to the input, should not alter the system's output. For example, a camera image of a straight road taken during the day and at night should not affect the steering angle of an autonomous vehicle. Ranges of these exist, from the naive, following standard data

augmentation techniques such as affine transformations or adding noise [399, 265, 108], to more complicated examples such as changing weather, lighting, or specific objects in the world [345, 304, 361], to advanced scenarios that take into account the world's semantics, by being cognizant of what individual sensor readings mean, allowing for transformations such as adding other vehicles driving on the street [377, 61].

While efforts to formalize larger more complete sets of specifications that are unambiguous, and correct are being discussed [196, 100, 173], defining a user-friendly, precise, mathematically-based language capable of validating an entire autonomous system across its entire input/output domain remains an unsolved problem.

The next category of *oracle*s derives the expected output not from precisely defined specifications, but from alternative artifacts. One example of this is differential testing from traditional software engineering [244]. Here, the approach assumes that there are multiple systems implemented to meet the same specifications. Therefore, by comparing these systems' outputs, if there is a difference, we potentially have found a case where one of the systems violated the specifications. This approach has seen application across diverse autonomous systems, from comparing aviation software operations [122] to comparing autonomous vehicle behaviors with human drivers [312], and analyzing different neural network versions trained on identical data [387]. Another form of artifact against which an oracle can compare behavior is models of the system. For example, comparing relationships between different parts of the system to finite state automata [396], or assessing the system's general real-world behavior against its internal state [233].

The third and final category is implicit test *oracle*s. These rely on universally accepted principles to differentiate between correct and incorrect behavior. This method is grounded in the premise that certain outcomes are expected, such as an autonomous system should not crash [329, 107, 215], an autonomous car should not drive off a road [106], or an autonomous car should not leave the lane [189]. While these oracles might seem trivial, they can be extended, for example, using reachability analysis to assert that they maintain at least a minimum potential time or distance to collision [394].

While many types of oracles have been proposed and implemented for autonomous systems, most

papers continue to rely on basic ones. For example, when we examine the 26 papers cited in our background from the last two years, we find a total of 22 instantiated oracles[1][2]. Of those oracles, 4 are based on formal specifications, 3 are derived, and 15 are implicit. This finding suggests that the majority of papers still rely on implicitly set oracles based on a developer's domain knowledge. Furthermore, most of these oracles consider the output or behavior of the autonomous system independent of the input or sensor data. We suspect this is partly due to the difficulty of incorporating the *physical environment* and *physical semantics* into traditional test oracles, as described in Section 1.1. Specifically, we found that 15 of the 22 oracles use only output independent of the input, with the most common output being crashes (documented in 5 oracles). When oracles do consider particular inputs, they generally rely on either the road structure or the position of environmental obstacles. Specifically, 3 oracles compare the objects in the environment to generated trajectories, and 5 oracles compare the road to the autonomous system's current position. Our findings regarding the input/output most commonly used in oracles are consistent with those from a 2021 survey by Jahangirova et al. [162], which reviewed 238 papers on autonomous vehicle testing and found that most oracles consider output constraints applicable to all inputs. These inputs are almost always either velocity, position, steering, or collisions. When input is considered, it is primarily the road or objects in the environment, such as pedestrians, other vehicles, or traffic signs. Similar to our findings, input was almost exclusively compared to the position of the autonomous vehicle. In summary, although this is a crucial research area for autonomous systems, defining oracles over particular input remains challenging, and as a result, most oracles are relatively primitive.

2.2.2 Test Execution for Autonomous Systems

The next step in the testing pipeline is execution. This step, defined by the *context* of τ, the AS is executed to generate behaviors b. These b are then evaluated for their adherence to predefined correctness criteria, as defined through the *oracle*. The *context* encompasses the fidelity in which the world and autonomous system is executed. This execution environment significantly influences the

[1] We excluded survey papers and publications not related to autonomous system testing.
[2] See Appendix A.

testing process and can range from highly abstracted simulations to complex real-world scenarios. The choice between simulation and actual environments hinges on the test objectives, resource availability, and the level of fidelity required to validate the AS behavior.

2.2.2.1 Simulation Based Execution

The costs and potential impact of testing in the real world mean that the execution of tests produced by test generation for autonomous systems is still primarily done in simulation. These approaches focus on conducting extensive experiments to thoroughly explore the input space, aiming to uncover faults within both individual components [145, 81] and the system as a whole [258, 153]. This has resulted in many different simulation platforms for all types of autonomous systems including autonomous robots [64], drones [89] and vehicles [178]. These platforms offer a range of fidelities, making them a valuable tool at various stages of autonomous system development. For instance, low-fidelity simulators employ mathematical models to approximate the world and robot states [319, 214]. They are economical and practical for early testing stages. As fidelity increases, the simulation begins to model the world using a physics or graphics engine, enabling software-in-the-loop (SIL) simulation. High-fidelity physics simulators, paired with low-fidelity graphics, are especially useful for testing new robot designs and physical semantics [104, 277, 192]. High-fidelity graphics simulators with simple kinematic models are suitable for systems with rich sensor input, such as cameras and LiDARs [123, 397]. Some SIL simulators incorporate high-fidelity graphics and physics engines but tend to be more expensive and limited to specific autonomous systems or domains[87, 315, 33]. Hardware-in-the-loop (HIL) simulations combine the actual hardware with simulation [259, 264]. While they can yield more accurate system outputs, they also tend to be autonomous system specific, come with their own cost and scope limitations [208, 148]. However all simulations have a common limitation: they all suffer from the simulation-reality gap [163, 219]. That is the discrepancy between simulated environments and their real-world counterparts.

2.2.2.2 Real-World Based Execution

Due to the simulation-reality gap, real-world testing remains the gold standard for validating autonomous systems [368, 180]. This is evident when considering the efforts of autonomous car companies; for example, Tesla claimed to collect 1 million miles of data every 10 hours as of 2015 [323], comma.ai calimed to drive over 500,000 miles each week in 2021 [74], and Waymo claimed to have driven over 20 million miles on public roads since 2009 [311]. However, the physical world is vast, with an extremely long tail of rare scenarios that remain unexplored by even the largest autonomous system fleets [195]. This is evident as there are still reports of accidents and unusual or incorrect behaviors by these vehicles in the real world [243, 160, 198, 285, 47, 156].

However, conducting large-scale tests in the real world presents numerous challenges. These include high operational costs, substantial time and space requirements, significant labor for setup, and inherent risks, such as potential loss of life and damage to the system or other property [285, 160, 156, 243]. Consequently, a substantial portion of research is centered around conducting smaller-scale field studies within highly controlled environments. These experiments, ranging from flying drones [174, 321, 45], driving robots [251, 366, 341], to testing underwater vehicles [30, 384] in conditions isolated from external disturbances such as wind or currents and often utilizing high precision tracking systems like Vicon [359]. The tests yield essential insights and evidence, suggesting the systems' potential efficacy in the unpredictable real-world scenarios.

Despite their value, controlled field studies cannot fully mimic the real-world complexities faced by autonomous systems, underscoring the need for innovative, broad-scale testing methodologies. Emerging frameworks [31] show the research community's initial attempts to bridge this gap, by developing systems capable of efficiently switching between simulation and reality, yet they represent just the beginning of a longer journey. Fully addressing the breadth of real-world challenges encompassing cost, safety, and logistical feasibility, remains an open research question.

2.2.3 Test Adequacy Metrics for Autonomous Systems

Following the generation and execution of the test suite $\mathcal{T}$, and the analysis of the resulting behaviors b, the next essential step is to assess the effectiveness of $\mathcal{T}$. This process involves quantifying the comprehensiveness of the testing. Test adequacy metrics serve as a quantitative benchmark for this purpose, providing insights into the comprehensiveness of $\mathcal{T}$ and pinpointing areas that may require improvement [256, 406, 373, 374]. These metrics expose potential shortcomings within the test suite, thereby informing subsequent testing strategies. Broadly, test adequacy can be classified into four categories.

The first category is structural coverage [26, 131], commonly referred to as white box coverage [261]. Structural coverage involves examining the software's internal structures to determine which parts have been executed by the test suite [187]. Common instantiations of this metric include statement [257, 274], branch [376], path [168], and interprocedual path coverage [127], among others. We have seen broad adoption of these metrics [389], where they have been used across diverse domains and languages, including Python [14, 29], Android [300], Java [267], and C [151]. They have also proven useful in industry, where for example, structural coverage metrics are computed daily on over a billion lines of code at Google [159].

However, the effectiveness of these traditional metrics diminishes when applied to autonomous systems [197]. The first major issue is that many autonomous vehicle components are inherently statistical in nature, and thus result in non-deterministic behavior [344]. Emerging work on coverage criteria for learned components [231, 281, 334] aims to mitigate this challenge, but such components constitute just one of the sources of non-determinism. This can lead to identical inputs producing varied outputs, thereby artificially inflating the structural code coverage metric due to differing abstractions. Second, regardless of the test, many system components, such as those for control and planning, tend to have a linear control flow [202, 115] which causes structural coverage metrics to saturate quickly. Consequently, once an initial variety of inputs has saturated the metric, any additional unique inputs and behaviors fail to be effectively captured by these metrics.

The second category is functional coverage [222], often referred to as black box coverage [261].

This family of metrics concentrates on the software's inputs, outputs, and specifications while completely ignoring the software's internal operations. Historically, equivalence classes were employed to conceptualize this idea [121, 262, 372]. The foundational concept is that inputs within an equivalence class trigger similar behaviors. If this categorization is effective, the proportion of classes covered during testing can determine how thoroughly a test suite examines the system. Newer methodologies introduce a variety of other input models from which to derive such classes, accounting for system configurations [200], finite constraint sets [349], and grammars tailored for string-dependent systems [129].

The fundamental flaw with applying functional coverage to autonomous systems lies in its failure to determine input and output bounds. Treating data from sensors, such as camera feeds, as mere software inputs overlooks the environmental and physical limitations intrinsic to these systems [138]. This is evident when you consider the sheer volume of possible inputs a camera might produce—an HD camera can generate $16,777,216^{2,073,600}$ possible image combinations, as described earlier in Section 1.1.1. This number is more vast than the estimated number of atoms in the observable universe [282]. However, we know that a large portion of these images are not feasible, assuming the sensors are functioning properly. Without careful consideration of how to define the inputs for functional coverage, techniques inadvertently start to consider unrealistic scenarios, such as static noise, trees floating in the sky, or instantaneous velocity changes, as potentially valid inputs within the test coverage. Such inaccuracies distort the understanding of the system's interaction with the physical world, thereby affecting the perceived test adequacy. Ignoring these real-world and hardware constraints not only skews the operational limits and capabilities of the system but also risks safety-critical oversights, undermining the efficacy of functional coverage in ensuring the reliability and safety of autonomous systems.

One effort to mitigate this deficiency is scenario coverage [235], which incorporates the physical environment by building a situation graph containing the objects, their attributes, and their relationships in an environment. This approach is feasible as long as the ground truth graphs can be precomputed, which is difficult, severely curtailing its applicability beyond limited simulation environments. Additionally, while this approach can compute the number of scenarios covered, it has

no feasible way to compute the total number of possible scenarios. A refinement of this approach aims to measure the trajectory coverage within a given scenario [153], but it is limited in that it neglects the temporal nature of trajectories, and fails to account for the broader environment. Some alternative criteria based on both requirement coverage [158, 343, 297] and parameter coverage [207] have emerged. These can help define critical acceptance tests but remain inadequate for capturing the breadth of possible scenarios.

The third category is fault-based coverage [56]. Within this category, mutation coverage [211] is the most common technique. Mutation coverage aims to determine if the test suite can detect seeded faults of known types and generates a mutation score representing the percentage of seeded faults found by the test suite. While this coverage method has gained substantial traction in traditional software [167, 283], its exploration within the context of autonomous systems remains limited. The closest line of work related to autonomous systems involves the mutation of LiDAR scenarios for test generation in autonomous vehicles [61]. The most relevant research related to mutation coverage specifically for autonomous vehicles pertains to deep learning models that may be used in autonomous systems [232, 367].

The final category is model-based coverage [12], which uses principles similar to model-based test generation, as described in Section 2.2.1. Essentially, the software is represented as a model, and the tests' interaction with this model gauges the coverage. This method has been applied to traditional software [62, 11, 181]. However it has several limitations. These include the resource-intensive nature of model creation, model maintenance, and the potential for incomplete or inaccurate models, which may result in inadequate testing and undiscovered defects. The closest line of work that focuses purely on coverage, without including a generation component, does so for only a subset of the autonomous systems, the electronic control unit [78].

2.3 Physical Modeling and Analysis

Understanding the concepts of physical modeling and analysis is important within this dissertation, as they form the foundation for several validation techniques introduced [138, 140, 143]. One can

conceptualize physical modeling and analysis in three stages, as depicted in Figure 2.2.

The process begins with identifying the AS's current state s_i, which are its physical attributes and characteristics at a given instant i, depicted as a dark dot in Figure 2.2. These attributes may include variables such as position, velocity, and orientation. They offer a snapshot of the system's present condition.

Next, kinematic and dynamic (KD) models, sets of mathematical equations based on the physical attributes and capabilities of the AS, are applied to compute the AS's next state s_{i+1}. This computation is represented by a dashed arrow in Figure 2.2.

The final stage involves utilizing reachability analysis to compute all possible states the AS might encounter. This process can be thought of as calculating all subsequent s_{i+1}, for each potential action the AS could undertake, using the KD models. The resulting set, depicted in red in Figure 2.2, delineates the spaces or volumes that encompass all potential future states of the AS. Each of these sections is described in more detail below.

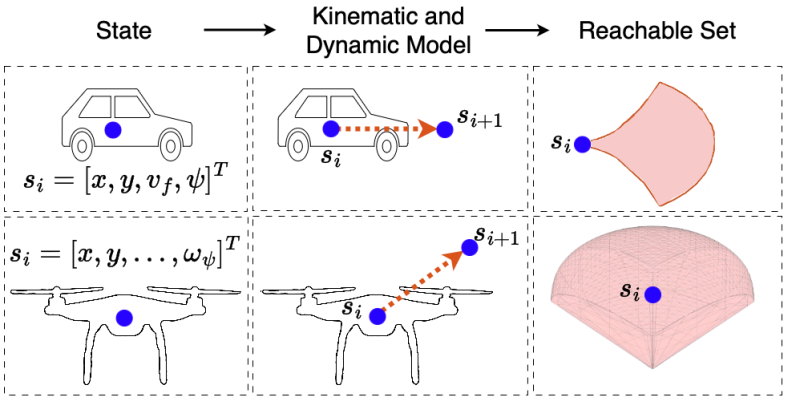


Figure 2.2: Visualization of a state s_i, future states s_{i+1} (dark dots), and all potential future states (shaded area or volume.)

2.3.1 Autonomous System State

An autonomous system AS operates in a scenario w and can only observe a portion of it, called a scene c. It uses its sensors, denoted as sen, to sense the scene c^{sen}, as well as its current state s^{sen}. This data is then consumed by the autonomous system to produce an action $a = AS(c^{sen}, s^{sen})$.

However, to understand this section, we need to grasp the difference between the software and

hardware of the system. The software of an autonomous system consumes (c^{sen}, s^{sen}) and produces an actuation command a', such that $a' = AS_{soft}(c^{sen}, s^{sen})$. In essence, it acts as the brain of the system, making decisions based on sensory input. The hardware then consumes the actuation command to produce the action in the real world, represented as $a = AS_{hard}(a', s^{sen})$. Essentially, this hardware involving electro-mechanical components, executes the decisions made by the software.

This raises a question: if the autonomous system is actually composed of two components, then what precisely is its "state"? The answer changes depending on whether we're approaching it from a software perspective, a robotics perspective, or a holistic perspective.

From a software perspective, discussing the state typically refers to AS_{soft}, the software component of the autonomous system. It consists of the information about the underlying software system, focusing on aspects such as variable values, the current execution point, and stored data. Essentially, this perspective views the state as an instantaneous abstraction of the software system, capturing all crucial information about the computational process at a given instant in time [147].

Conversely, within robotics, state typically refers to AS_{hard}, denoting the hardware aspect of the autonomous system. From this view the state is defined by its physical attributes, such as position, velocity, acceleration, alongside measurable properties like temperature, electrical currents, and mechanical stress [294, 355]. These attributes quantify the system's real-world conditions, reflecting both its own state and its interaction with the surrounding environment.

While these two definitions of state are often viewed as independent, it is crucial to recognize the existence of a holistic view, where there is interplay between them, culminating in the comprehensive AS state. The constraints and limitations inherent in the state of AS_{soft} directly impact the physical state of AS_{hard}, and vice versa. This dynamic interplay is fundamental to the development, testing, and operation of AS. The software state, with its own set of limitations and constraints, manifests as artificial limitations in the physical domain. For instance, if the software's thrust variable is a float constrained between 0 and 10 newtons, this range becomes an artificial limit on the physical system's produced thrust, regardless of the hardware's actual capabilities. Conversely, physical constraints and limitations of the hardware system, such as a maximum steering of ± 360 degrees, inherently limit the feasible current steering angle variable in the software state. This reciprocal shaping of

constraints between the software and hardware states underscores the intertwined nature of their limitations and the critical need for their integrated consideration in the design and operation of autonomous systems, as demonstrated in some of my previous work [137].

For the purposes of clarity in this dissertation, unless specified otherwise, the term state s will primarily denote the physical state of the system as governed by AS_{hard}. For instance, a simplified car state is represented by a 4th order state vector $s = [x, y, v_f, \psi]^T$, which represents the position (x, y) in 2-dimensional space, forward velocity (v_f), and heading (ψ) [294, 193, 287]. Conversely, more dynamic AS necessitate a more comprehensive description of their state. For example, a quadrotor's state can be described using a 12th order state vector $s = [x, y, z, \phi, \theta, \psi, v_x, v_y, v_z, \omega_\phi, \omega_\theta, \omega_\psi]^T$, encapsulating the position (x, y, z), attitude (roll ϕ, pitch θ, yaw ψ), linear velocities (v_x, v_y, v_z), and angular velocities ($\omega_\phi, \omega_\theta, \omega_\psi$) in three-dimensional space [355, 395].

2.3.2 Kinematic and Dynamic Models

Kinematic and dynamic (KD) models are sets of mathematical equations that predict the motion of an object given some initial state, and input. These models have proven particularly useful in robotics and autonomous systems [34, 332], as they enable predictions and insights into how the physical hardware component AS_{hard} of the autonomous system will behave.

Kinematic models are primarily concerned with the geometric aspects of motion, focusing on parameters such as position, velocity, and acceleration, without addressing the forces responsible for these states of motion [199]. This perspective allows for a simplified analysis of movement, essential for initial stages of design and understanding system capabilities in a controlled environment. Dynamic models, on the other hand, explore the causative forces behind motion, incorporating both internal mechanics and external factors into their calculations [110]. By analyzing these forces, dynamic models offer a comprehensive view of how and why an object moves, making them indispensable for predicting interactions with unpredictable environments.

The synergy between kinematic and dynamic models enables the accurate prediction of future states from an arbitrary current state s_i and input u_i at any given instant i. Similar to how the physical hardware of an autonomous system updates s using the actions a produced by $a =$

$AS_{hard}(a', s^{sen})$, KD models can predict how s would change, represented as $s_{i+1} = KD(u_i, s_i)$. Such a mechanism is vital for the understanding, and designing of these systems, allowing insight into how the physical system will behave in a wide range of s for a wide range of u.

As introduced in Section 2.3.1, we examined state representations using both car and quadrotor examples. Moving forward, both this section and the subsequent one will concentrate on the more complex quadrotor example. The simpler car kinematics, while foundational, are based on similar principles and can be independently explored for further understanding [294, 193, 287]. To compute the quadrotors s_{i+1}, we start with calculating control inputs u_{1-4} based on the the four motor speeds w_{1-4}, and physical attributes of the quadrotor, as shown in Equation 2.1. In this equation, u_1 represents the total upward thrust F generated by the rotors, u_2 and u_3 denote the differential thrust affecting roll M_x and pitch M_y, respectively, while u_4 captures the torque variance between clockwise and counterclockwise rotors, influencing yaw M_z. These calculations are based on the physical attributes of the autonomous system, specifically quadrotors arm length d and proportionality constants for thrust k_f and moments k_m.

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \begin{bmatrix} F \\ M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} k_f & k_f & k_f & k_f \\ 0 & dk_f & 0 & -dk_f \\ -dk_f & 0 & dk_f & 0 \\ k_m & -k_m & k_m & -k_m \end{bmatrix} \begin{bmatrix} w_1^2 \\ w_2^2 \\ w_3^2 \\ w_4^2 \end{bmatrix} \tag{2.1}$$

The values of u_2, u_3, and u_4 are used to compute the change in quadrotors angular velocity ω using Equations 2.2, where the I terms correspond to the inertial properties unique to each quadrotor.

$$\begin{bmatrix} \dot{\omega_\phi} \\ \dot{\omega_\theta} \\ \dot{\omega_\psi} \end{bmatrix} = \begin{bmatrix} \frac{I_{yy}-I_{zz}}{I_{xx}}\omega_\theta\omega_\psi \\ \frac{I_{zz}-I_{xx}}{I_{yy}}\omega_\phi\omega_\psi \\ \frac{I_{xx}-I_{yy}}{I_{zz}}\omega_\phi\omega_\theta \end{bmatrix} + \begin{bmatrix} \frac{1}{I_{xx}} & 0 & 0 \\ 0 & \frac{1}{I_{yy}} & 0 \\ 0 & 0 & \frac{1}{I_{zz}} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \\ u_4 \end{bmatrix} \tag{2.2}$$

The quadrotor's angular velocity ω is then used to compute the change in the attitude of the quadrotor using Equations 2.3.

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & sin(\phi)tan(\theta) & cos(\phi)tan(\theta) \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi)sec(\theta) & cos(\phi)sec(\theta) \end{bmatrix} \begin{bmatrix} \omega_\phi \\ \omega_\theta \\ \omega_\psi \end{bmatrix} \tag{2.3}$$

Finally, the change in velocity is computed using Equations 2.4. The new velocity is used to update the position of the quadrotor.

$$\begin{bmatrix} \dot{v_x} \\ \dot{v_y} \\ \dot{v_z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{1}{m} \begin{bmatrix} cos(\phi)cos(\psi)sin(\theta) + sin(\phi)sin(\psi) \\ cos(\phi)sin(\theta)sin(\psi) + cos(\psi)sin(\phi) \\ sin(\theta)sin(\phi) \end{bmatrix} u_1 \tag{2.4}$$

The versatility and efficacy of KD models have led to their adoption across various fields, beyond the immediate scope of robotics and autonomous systems. These models have proven instrumental in mechanical engineering [90], astrophysics [392], biomechanics [327], and even in creating realistic physics simulations within video games [242].

2.3.3 Reachability Analysis for Autonomous Systems

Reachability analysis is a method used to calculate the reachable set r for an AS. This set r represents the collection of all possible future states the AS can achieve within a specified time horizon h, given the system's physical capabilities [57, 13, 164]. We note that similar to the distinction between software states and physical states, we are focusing on physical reachability analysis rather than software reachability analysis, the latter of which centers around determining the portions of code, functions, or software states, which will be executed given a set of input [43]. Utilizing KD models, physical reachability analysis employs the current system state s_i, a defined time horizon h, and all possible inputs $\mathcal{U}$ to compute the reachable set. It can be intuitively understood as shown in Equation 2.5, where given s_i, it computes all possible future states over h using the KD and all inputs $\mathcal{U}$. Each of these future states is geometrically combined which we donate with $\oplus$, to produce r_i.

$$r_i = s_i \oplus \int_i^{i+h} KD(s_i, \mathcal{U})dt \tag{2.5}$$

The resultant r_i defines the spatial or volumetric limits the AS can explore within the given h at an instant i in time. Areas outside r_i remain unreachable for the given h, irrespective of the inputs $\mathcal{U}$ applied to the AS given its current state s_i. An illustrative example is provided in Figure 2.3, showcasing a quadrotor (dark blue dot) reachable set depicted (red shaded cone).

This visualizes all positions of s_{i+h} attainable by the quadrotor, contingent upon its initial state s_i and all inputs $u_i \in \boldsymbol{U}$. Specifically in this example we see that the quadrotor is stationary 20m above the origin. If all motors were switched off, it would fall precisely 9.81m from 20m to 10.2m over a 1-second time horizon due to gravity. This is represented by the lowest point of the red shaded cone in Figure 2.3. Conversely, if all motors were activated, based on the motor RPM and quadrotor dynamics, it would raise 8m from 20m to 28m. This is represented as the apex of the curve of the red shaded dome. The other areas represent all other possible positions the quadrotor could reach in the same 1-second time horizon, given $\mathcal{U}$.

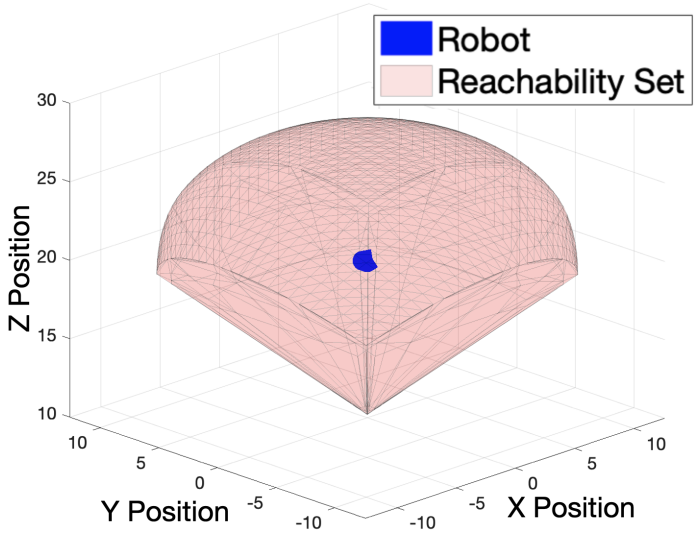


Figure 2.3: A quadrotor's (dark dot) reachable set (shaded area) illustrates the range of potential movements.

Despite its invaluable insights, the computation of reachable sets is recognized for its computational intensity, prompting significant research into optimizing this calculation process [54, 118, 155, 204, 249, 225]. Various methodologies have been proposed to address these challenges, generally trading accuracy for computational complexity.

Chapter 3

Test Generation

Test generation serves as the initial step in the validation process. Creating a test τ requires developers to account for three critical components, namely the *input*, *oracle* and *context*. This Chapter presents two approaches for creating the *input* and *oracle*, deferring the discussion of *context* to Chapter 4 which focuses on test execution.

Our first approach described in Section 3.1, builds upon stress testing in traditional software [39, 255], which aims to determine the robustness of software applications and systems under extreme conditions. Determining what constitutes extreme conditions for autonomous systems is particularly challenging, as the concept of "extreme" is intrinsically linked to the physical semantics of each autonomous vehicle. For instance, what is considered a stressful amount of acceleration or braking for an autonomous Formula 1 race car vastly differs from that for an autonomous passenger vehicle. By integrating the physical semantics directly into the development of a test's *input*, we can tailor it specifically for the autonomous vehicle being tested. However, while doing this ensures that an input is feasible for a given autonomous system, developers must also identify which inputs are likely to stress the system. To address this, we introduce a test *oracle*, termed a parameterizable scoring model, which leverages either domain knowledge or machine learning to predict stress. This *oracle* enables us to ascertain the success or failure of tests, given insights from developers or sufficient data, and can predict which tests are most likely to reveal stress in a system. The result is a set of

τ which are physically feasible and stressful for any given autonomous system.

Our second approach, in Section 3.2, explores *input* and *oracle* through the lens of differential testing from traditional software [244, 213]. Differential testing involves applying the same inputs across similar systems, or different implementations/versions of the same system to detect discrepancies in output behavior. The application of differential testing to autonomous systems encounters two challenges. First, this approach requires some *input*, which for an autonomous systems includes sensor data. Generating a vast and diverse amount of sensor data that accurately captures the physical environment an autonomous system is designed to handle is both costly and complex. We address this by leveraging the vast amount of existing real-world environmental sensor data that is readily available, transforming it as necessary to meet the requirements of each autonomous system. We then apply a filtering stage to remove *input* which the autonomous system was not designed to handle. Second, developing an *oracle* for autonomous systems presents a significant challenge because there are often multiple correct and incorrect behaviors in a given scenario, and determining the most desirable outcome is complex [162]. This complexity arises from the vast range of possible *input* under which these systems operate, making it difficult to define universally appropriate behaviors. To overcome this we employ a differential testing framework, which compares multiple versions of an autonomous systems behavior to identify discrepancies and potential failing behavior among the systems for any given *input*. To address this challenge, we utilize a differential testing framework that employs these systems as cross-referencing *oracles*. By comparing the behaviors of these autonomous systems when exposed to the same input, we can identify discrepancies. Any divergence in responses to the same input is flagged as a potential fault. The result is a set of potential failure-inducing test cases τ which can be used as future for improving autonomous systems.

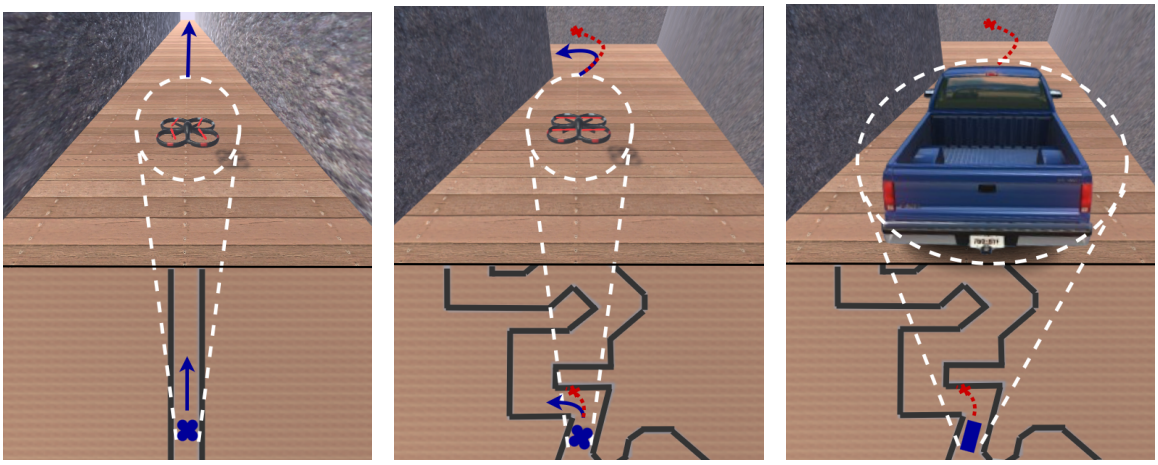
3.1 Feasible and Stressful Trajectory Generation

Many autonomous systems need to traverse their physical environments to achieve their goals. To do this, an autonomous system must compute and perform a combination of translation and rotation within the physical environment. The types of translations and rotations are dependent on the

physical semantics of the autonomous systems. A common method for developers to request such traversal is through waypoints. Waypoint traversal is a staple of autonomous system navigation and is found in almost all types of autonomous systems, including wheeled robots [316], advanced driver-assistance systems [125], autonomous fixed-wing aircraft [185], autonomous multirotor aircraft [306], and autonomous marine vehicles [22]. For more complex physical environments, or goals, waypoint navigation alone is sometimes not enough. For example, consider the case where an autonomous system needs to navigate around an object directly along the line of translation. A common solution in such cases is to follow a series of waypoints, known as a trajectory. This approach allows the autonomous agent to navigate around the obstacle by placing waypoints in a manner that avoids any need for translation through the obstacle.

To ensure that these types of systems behave correctly and to identify any potential failures, many test generation techniques aim to create trajectories that mirror potential real-world operations [87, 315]. For instance, in the context of advanced driver-assistance systems, trajectories can be devised over existing road maps with typical traffic loads [340], over synthetic maps that meet road-design and traffic constraints [182], or over scenarios developed following certain probability distributions [101]. A similar approach is adopted for autonomous multirotor aircraft testing, with adjustments made to account for the extra dimension of space [315].

While exploring typical trajectories is necessary to validate the behavior of autonomous vehicles, it may overlook faults that arise in the presence of "stressful" trajectories, trajectories that accentuate a particular behavior of the particular autonomous vehicle. In this context, stress is determined by a combination of the tests *input*, which in this case is a trajectory, and the vehicle's physical semantics. For example, the intricate maneuvers required for a micro-drone to navigate a narrow tunnel serve as a good illustration of this principle. The trajectory depicted in Figure 3.1a is relatively straightforward and presents a much less demanding test compared to that of Figure 3.1b. This latter trajectory introduces sharp turns that place significant stress on every aspect of the system, from its perception components to motor capabilities. However, applying the same trajectory from Figure 3.1c to a different kind of autonomous system, such as the advanced driver-assistance systems in an autonomous car, drastically alters the scenario. What constituted a stressful trajectory



(a) A quadrotor navigating in a straight corridor.

(b) A quadrotor navigating a more difficult trajectory through a winding corridor.

(c) A car unable to navigate the more difficult trajectory through a winding corridor.

Figure 3.1: Top half shows the autonomous vehicle from behind, while the lower half shows a birds eye view of the scenario. The dashed lines convey location across views, solid arrows show optimal behavior, while dotted arrows show unforeseen behavior leading to a collision.

for the autonomous drone becomes infeasible for the autonomous car, due to its inability to execute such tight maneuvers. This example underscores the importance of crafting *input* trajectories that not only push the boundaries of the autonomous system but also consider its *physical semantics*, ensuring that the *input* remains challenging yet achievable given the system's capabilities.

This work aims to develop a method for automatically generating such feasible and stressful trajectories for any given autonomous vehicle. This involves addressing three main challenges: 1) determining the physical feasibility of a trajectory given the *physical semantics* of the vehicle, 2) efficiently finding trajectories that induce stress, and 3) ensuring the method's applicability across various types of autonomous vehicles and stress measurements.

To tackle the first challenge, we leverage the understanding that the physical semantics of autonomous vehicles are often represented through KD models. Our method employs these KD models to calculate all potential states within the vehicle's reachable set. States outside this set are deemed physically unfeasible. For the second challenge, we utilize reachability analysis and KD models to effectively sample the valid physical space. This includes a scoring model that evaluates the stress

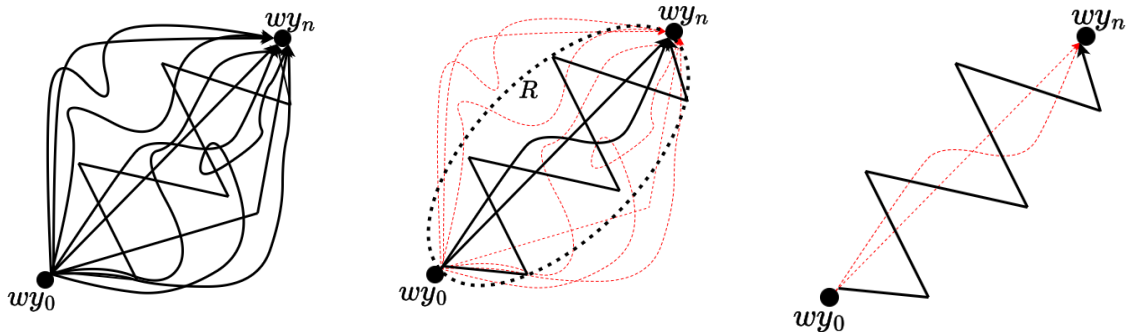
level of each trajectory during its generation. Additionally, a beam search strategy is implemented to methodically explore the trajectory space, aiming to identify and prioritize those that are most likely to impose significant stress on the system. Addressing the final challenge, our approach's generality is ensured by building on models that are accessible or can be easily estimated for most autonomous vehicles. This is complemented by a high level of abstraction and parameterization in the trajectory search process, along with employing the Robot Operating System (ROS) for standardizing the messaging formats [328] as part of our implementation framework.

3.1.1 Approach

The goal of the approach is the systematic and efficient generation of both feasible and stressful trajectories for autonomous systems. To define a trajectory, we first need to define a waypoint. We define a waypoint as a position pos and orientation ori pair; this is also known as a pose $wy_n = (pos_n, ori_n)$. Following this definition, a trajectory can then be defined as a sequence of waypoints, $traj = \langle wy_0, \ldots, wy_n \rangle$. We note that, while trajectories can also include timing information, ours do not; they only require that waypoints be visited in the defined order. In the following sections, we provide an overview of the approach, a detailed description of how our approach identifies both feasible and stressful trajectories, a running example of the approach, and a description of the implementation.

3.1.1.1 Overview

A high-level overview of the approach is presented in Figure 3.2. It is structured into three phases. Initially, a large set of potential trajectories is generated between start and end waypoints, denoted by wy_0 and wy_n respectively, as shown in Figure 3.2a. This is followed by a filtering phase in Figure 3.2b, where $traj$ not aligning with the autonomous system's reachable set R are removed, ensuring all remaining $traj$ are physically feasible. Finally, from these feasible trajectories, the most stressful ones are selected based on a scoring model, as shown in Figure 3.2c. This model either relies on expert input or machine learning techniques to identify the set of trajectories that maximize stress.



(a) Potential Trajectories. (b) Infeasible Trajectories. (c) Stressful Trajectories.

Figure 3.2: An overview of our approach, a) from the set of potential trajectories, b) remove all physically infeasible, and then c) select the stressful ones.

To achieve this, our approach consists of two main algorithms. Algorithm 1 manages the search for trajectories through the use of an exploration frontier function. The frontier consists of all the trajectories which are currently under consideration. Algorithm 1 expands the frontier through calls to "explore_frontier" which is described in Algorithm 2. Algorithm 2 controls how the frontier is explored by only checking trajectories that are both feasible and stressful.

3.1.1.2 Trajectory Generation

Algorithm 1 manages the search for feasible stressful $traj$ inside w. To keep the approach general, Algorithm 1 takes in ten parameters: (1) w: a scenario defining the physical volume in which trajectories will be executed, (2-3) wy_0 and wy_n: the start and ending waypoints for the returned trajectories, (4) n_{wy}: The number of waypoints to be explored in w, (5) n_{traj}: the required length of a trajectory, (6) t: the total computation time allowed, (7) res: resolution of samples used to compute the reachable set, (8) $width$: the number of trajectories explored and expanded during each loop of the algorithm, and (9) KD: the KD model of the autonomous system, (10) SM: a Scoring Model used to select the most promising trajectories to be further explored.

The goal of Algorithm 1 is to generate trajectories of length n_{traj}, that start and end at user-defined waypoints wy_0 and wy_n. It does this by representing w as a graph g_w, where the vertices are

Algorithm 1: Trajectory Generation Manager

Input : w, wy_0, wy_n, n_{wy}, n_{traj}, t, res, $width$, KD, SM
Output: $Traj_s$

1 $Traj_s = \emptyset$
2 **while** *Execution Time* $< t$ **do**
3 Wy = generate_random_waypoints(w, n_{wy})
4 g_w = create_graph(wy_0, wy_n, Wy)
5 $traj_0 = \{wy_0\}$
6 $Frontier = \{(traj_0; 0)\}$
7 $Traj_c = \emptyset$
8 **while** $Traj_c == \emptyset$ *and* $|Frontier| > 0$ **do**
9 $Frontier'$, $Traj_c$ = explore_frontier(g_w, wy_n, n_{traj}, $Frontier$, res, $width$, KD, SM)
10 $Frontier = Frontier \cup Frontier'$
11 **end**
12 $Traj_s = Traj_s \cup Traj_c$
13 **end**
14 return $Traj_s$

waypoints. Each vertex is connected to all other vertices by the shortest straight line between them, creating a complete graph. An edge represents the optimal path an autonomous system should follow to traverse between any two waypoints. A path is created by combining sequences of vertices and following the edges between them. All paths through the g_w represent all the possible trajectories in w.

It starts by initializing the set of stressful trajectories $Traj_s$ to an empty set in line 1. Algorithm 1 repeatedly generates trajectories until it exceeds a computation time of t and then returns the generated $Traj_s$ in line 14. The $Traj_s$ are computed in lines 2-12 as follows. First, in line 3, a set of random waypoints are generated Wy. Line 4 creates the graph g_w. The graph's vertices consist of n_{wy} randomly sampled waypoints as well as the user-defined wy_0 and wy_n waypoints. This approach, of creating g_w, consisting of n_{wy} random waypoints, is inspired by probabilistic roadmap planners (PRM) [179]. Lines 5-7, initialize a search $Frontier$. The $Frontier$ is a set of trajectories and trajectory score pairs. The $Frontier$ is used to track the explored trajectories and is incrementally expanded in the "explore_frontier" algorithm. It is initialized with $traj_0$, a trajectory containing the user defined wy_0, and a score of 0. A trajectory score represents an estimation of the trajectory induced stress on the autonomous system. The stress is estimated using a SM described later

in Section 3.1.1.6. The algorithm repeatedly invokes "explore_frontier" in line 9 to incrementally expand the $Frontier$ and search for complete trajectories $Traj_c$; trajectories which start at wy_0, end at wy_n, and are length n_{traj}. Algorithm 2 describes "explore_frontier", which returns the newly explored frontier $Frontier'$ and complete trajectories $Traj_c$ from each iteration.

3.1.1.3 Efficiently Exploring the Frontier

Algorithm 2 describes "explore_frontier" which can be broken down into four stages. First, Algorithm 2 selects trajectories from the $Frontier$ based on trajectory scores. Second, Algorithm 2 computes the physical space reachable by the autonomous system given the current autonomous system state. Third, Algorithm 2 expands the $Frontier$ by building a new set of trajectories by estimating the autonomous systems future state at each waypoint within the reachable space, and then using the estimated state to build new trajectories. Finally, the "assign_scores" algorithm gives a score to each of the new trajectories in the $Frontier$.

More precisely, Algorithm 2 starts by sorting the current $Frontier$ based on each trajectory score. The top $width$ trajectories are selected for further processing. The larger the $width$, the more trajectories are explored per invocation of the "explore_frontier" algorithm, and the more computationally expensive the operation is. However, the larger the $width$, the more likely the algorithm will find a trajectory that ends at wy_n, while also inducing large amounts of stress.

Selecting from the $Frontier$, in line 6-10, consists of removing the i^{th} most promising trajectory and checking if it meets the requirements to be a complete trajectory. If so it is added to the set of complete trajectories $Traj_c$ in line 9. In lines 11-19, if the selected trajectory is shorter than n_{traj}, the search continues by expanding the selected trajectory and adding it to $Frontier'$.

Before the selected trajectory is expanded and $Frontier'$ computed, a reachable set R needs to be computed. R defines the physical space the autonomous system can achieve in a time step given its current state. Thus all wy inside both g_w and R are feasible for the autonomous system. More specifically, the reachable set is computed using the autonomous systems last known state s_{last}, the autonomous systems KD model, and a sample resolution res. Computing R is described in Section 3.1.1.4.

Algorithm 2: Explore Frontier

```
1  Function explore_frontier(g_w, wy_n, n_traj, Frontier, res, width, KD, SM)
2      Traj_c = ∅
3      Frontier' = ∅
4      Sortedfrontier = sort(Frontier.scores)
5      for i = 0; i < width; i++ do
           // Select From Frontier
6          traj = Sortedfrontier[i].traj
7          Frontier = Frontier ∩ not traj
8          if |traj| == n_traj, and traj[n_traj].position == wy_n then
9              Traj_c = Traj_c ∪ traj
10         end
11         if |traj| < n_traj then
               // Calculate Reachable Set
12             s_last = traj[last].s
13             R = calculate_reach_set(s_last, res, KD)
14             for wy in (g_w ∩ R)) do
15                 s_new = estimate_state(s_last, wy.position)
16                 traj_new = traj ∪ {wy.position, s_new}
                   // Expand Frontier
17                 Frontier' = Frontier' ∪ (traj_new, ∅)
18             end
19         end
           // Assign Scores
20         Frontier' = assign_scores(Frontier', SM)
21     end
22     return Frontier', Traj_c
23
```

Once all the feasible waypoints for the autonomous system are known, the algorithm expands the *Frontier*. Each new trajectory is then added to the frontier and scores assigned to them before being returned.

3.1.1.4 Reachability Analysis to Explore the Feasible Frontier

The computed reachable set R allows the "explore_frontier" function, described in Algorithm 2, to precisely identify which $wy \in g_w$ are achievable given the autonomous systems KD model and s_{last}. Thus trajectories that the autonomous system could not physically achieve can be rejected during trajectory generation, as opposed to during trajectory execution.

In this work, we explore two techniques to compute reachable sets and later compare them to a baseline technique that sets the entire space as reachable. The first approach over-estimates R, by setting it to a sphere around the wy position, whose radius is equal to the maximum velocity the autonomous system can travel in 1 second.

The second approach leverages the full KD model to compute the reachable set. Computing such reachable sets is an active area of research and described in more detail in Section 2.3.3. For simplicity, we implement a brute force technique to compute it. Given s_{last}, we generate a set of input samples and apply the KD model to produce a set of potential reachable states. The convex hull of this state set is computed, offering a more precise approximation of R compared to our previous approach that utilized a simple sphere. This approach requires res^x evaluations of the forward KD model equations, where x is the number of input variables for the KD model equation, and res is the number of input samples taken [60]. For example, in the case of a quadrotor, which we later study, there are 4 input variables, one for each of the motors on each arm. If permutations of 5 linearly sampled inputs are taken, the approach would need to perform $5^4 = 625$ computations resulting in 625 achievable future states.

3.1.1.5 Estimating Autonomous System State for Trajectory Building

The autonomous systems state at a new waypoint, labeled s_{new}, is estimated, on line 15 in algorithm 2, based on the autonomous systems previous state s_{last} and the current waypoint wy. Approaches to state estimation can vary in cost and precision. At two extremes of this spectrum are estimators that 1) assume the autonomous system is at rest when reaching a waypoint, and 2) solve the inverse of the KD model equations. The first is inexpensive, but imprecise and the second is precise, but expensive.

We implement a hybrid approach that utilizes only specific segments of the KD model equations to estimate the state, while the remaining state variables are set to their resting values. This selective reset of state variables is configurable. For instance, in the case of the quadrotor we examine later, our method calculates the expected velocity at each waypoint by dividing the Euclidean distance between waypoints by the time allocated between them. Meanwhile, it resets the attitude and

angular velocity to 0 at each waypoint. This implies that we assume the quadrotor enters each waypoint in a level orientation.

3.1.1.6 Assigning Scores to Select Next Trajectory

Scores are assigned to each trajectory in the *Frontier* using a scoring model SM, as described in Algorithm 3. For each *traj* in the *Frontier* we start with an initial score of 0. The algorithm iterates through each pair of waypoints in the trajectory and assigns a score to the pair. The final trajectory score is then computed by accumulating the scores for that trajectory.

The scores are assigned by an SM and are calculated based on an estimate of the stress that the autonomous system will incur. A good SM will accurately estimate this stress, using a scalar metric, given two autonomous waypoints. Depending on the application of the autonomous systems, stress can be measured using different stress metrics. For example, three possible metrics are maximum deviation from the expected trajectory, maximum acceleration, or total time. The only requirement is for the selected stress metric to be measurable during autonomous systems execution. The main stress metric we use in our study is maximum deviation from the expected trajectory, which is illustrated in Figure 3.3. The maximum deviation, a standard measure associated with navigation safety, is a measure of the largest error between the expected position of an autonomous system and its actual position. In this work, we explored two classes of scoring models that we later compare to a baseline scoring model that randomly selects a score.

Algorithm 3: Assign Scores

```
1  Function assign_scores(Frontier, SM)
2      for traj in Frontier do
3          score = 0
4          for i = 0 to len(traj) − 1 do
5              wy_i = traj[i]
6              wy_{i+1} = traj[i + 1]
7              score += SM(wy_i, wy_{i+1})
8          end
9          traj.score = score
10     end
11     return Frontier
```

The first scoring model leverages a user's domain knowledge to create rules likely to maximize some goal, for instance in our case, maximizing deviation. In our evaluation, for example, we identified the trajectories velocity v_{in}, v_{out}, and the trajectory angle Θ, as shown in Figure 3.3, as attributes likely to be correlated to maximum deviation. For example, a large v_{in} and Θ correspond with the intuition that entering a waypoint with high velocity might result in a significant deviation if the autonomous system is also required to take a sharp turn. In general, the effectiveness of such a model will depend on a domain expert's ability to identify the attributes as well as how closely the attributes align with the autonomous systems behavior, which depends on the autonomous systems planner, controller, sensing, and actuation capabilities.

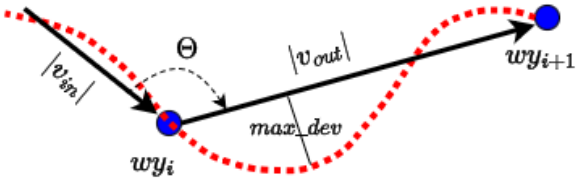


Figure 3.3: Trajectory attributes and the stress metric maximum deviation. The solid line is the expected trajectory while the dotted line is the true behavior.

The second scoring model learns from previous data. It consists of using a collection of trajectories generated using a random scoring model and subsequently identifying the factors that lead to particularly stressful trajectories. This knowledge can then be used to score future trajectories on their ability to cause stress. As an example, assume that there is a series of generated trajectories. The autonomous system could then execute the trajectories to render an actual maximum deviation. The traversed trajectories could then be broken down into pairs of waypoints (wy_i, wy_{i+1}) like that of Figure 3.3. The maximum deviation max_dev associated with each (wy_i, wy_{i+1}) and a set of attributes that may be associated with that deviation (e.g., v_{in}, v_{out}, Θ) could be used as training data. Then a learning technique can be used to produce a SM that, given a pair of waypoint attributes, can estimate the expected maximum deviation.

In our evaluation, we generated a SM using a polynomial regression model where the loss function is the linear least-squares function, and regularization is given by the ℓ_2-norm [113]. We determined the best polynomial degree using 10-fold cross-validation. If the resulting model provides a good fit (i.e., strong correlation and low cross-validation loss), then it can be used to assign predictive scores

to future trajectories without executing them. This approach incurs the cost of trajectory execution to generate the data to train the model. Thus its applicability depends in part on the cost of such execution. In many cases, such costs can be mitigated, for example through simulation, and overall it is beneficial in that it does not rely on the user's expertise.

3.1.1.7 Example Trajectory Generation

We provide a step-by-step illustration of our approach in Figure 3.4. In this example, we show 6 random waypoints, we explore 2 trajectories at a time, and we are looking for trajectories with 4 waypoints. More specifically, g_w is generated with n_{wy} set to 6, we set $width$ to 2, and n_{traj} to 4. After the g_w's construction using a modified version of PRM, we select from the frontier, which after the initialization in Algorithm 1, is a single trajectory that contains wy_0. We calculated the R for the last and only waypoint (wy_0) in the trajectory as described in Section 3.1.1.4. We then expanded the $Frontier$ using each of the waypoints inside g_w and R. Specifically we create 3 new trajectories $\langle wy_0, wy_1\rangle$, $\langle wy_0, wy_2\rangle$, and $\langle wy_0, wy_3\rangle$ by estimating the state s_{new} at each $wy.position$ and adding them to the $Frontier$ as described in Section 3.1.1.5. Scores are assigned to each of the new trajectories based on a SM as described in Section 3.1.1.6.

On the second iteration, due to the $width$ of 2, the two highest-scoring trajectories, $\langle wy_0, wy_2\rangle$ and $\langle wy_0, wy_3\rangle$, are selected from the frontier (filled circle). For each of the selected trajectories last waypoints, an R is calculated. The $Frontier$ is expanded using the waypoints in each R. This results in 4 new trajectories. It is important to note that a single waypoint may feature in multiple trajectories, demonstrating our approach's versatility. For example, the centrally positioned waypoint wy_2 is also referred to as wy_4 in a different trajectory. This naming variation emphasizes that each waypoint encapsulates a pose, or both position and state In particular, while wy_2 and wy_4 occupy the same position, they are differentiated by their states. Specifically, wy_2 is consistent in state across the trajectories $\langle wy_0, wy_2, wy_6\rangle$ and $\langle wy_0, wy_2, wy_7\rangle$. Conversely, in the trajectory $\langle wy_0, wy_3, wy_4\rangle$, the waypoint is represented as wy_4 to signify a different state, despite sharing its position with wy_2. This distinction highlights the flexibility of our approach to create many different trajectories through both variations in position and state. Following this, scores are assigned to each

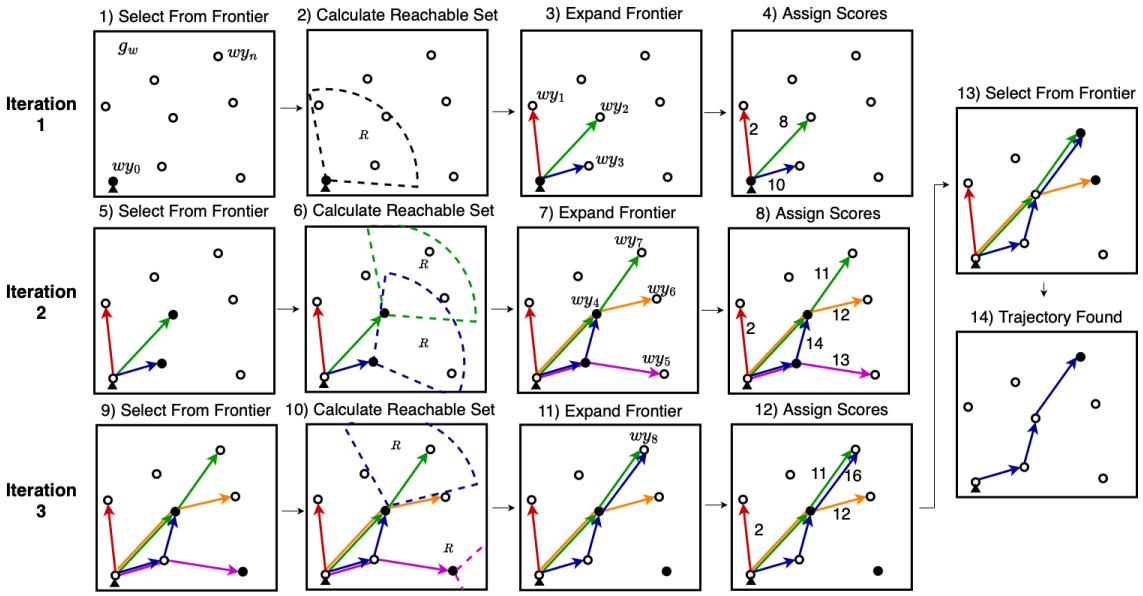


Figure 3.4: Our approach illustrated with waypoints (circles), trajectories (solid lines), and reachable sets (dotted lines). The example considers a 2D world with 6 random waypoints, a beam width of 2, and a trajectory length of 4. The autonomous system in this example, starts with 0 velocity and is facing directly upward (small triangle).

trajectory, and selections are made from the frontier based on these scores.

On the third iteration, we notice that the trajectory $\langle wy_0, wy_3, wy_5 \rangle$ produces an R with no further waypoints inside. Thus that particular trajectory is removed from the $frontier$. After this iteration two trajectories $\langle wy_0, wy_3, wy_4, wy_8 \rangle$ and $\langle wy_0, wy_2, wy_6 \rangle$, are chosen based on their scores of 16, and 12 respectively. The first trajectory meets the criteria of starting at wy_0, ending at wy_8 which shares a position with wy_n, while also being of length 4. This trajectory is added to the $traj_c$, and the algorithm repeated with a new g_w. Although this is a hypothetical example, the approach still selects a stressful trajectory. The final trajectory requests the autonomous system to take an immediate ≈70 degree right turn, followed by a ≈45 degree left turn before moving to wy_n.

3.1.2 Study

The goal of the study is to assess the proposed approach and determine what benefits the introduction of the KD model and scoring models has on stressful trajectory generation for autonomous

systems. More specifically, we aim to answer the following research questions for automated trajectory generation:

RQ1) Does the introduction of a KD model improve the ability to generate feasible and valid trajectories with respect to the physical semantics of the autonomous system?

RQ2) Does the introduction of a scoring model improve the ability to generate stressful trajectories?

3.1.2.1 Setup

The study w is set to a $30m \times 30m \times 30m$ area with 250 randomly placed waypoints. This selection matches the volume ($27000m^3$) and size of a typical outdoor aerial testing facility [203, 260, 360].

The systems we used are listed in Table 3.1. The first is an autonomous racing quadrotor executed in the publicly available FlightGoggles simulator [123]. The quadrotor has a weight of $1kg$, and a body length of $0.45m$ [309]. Its maximum velocity in simulation is $18m/s$ [239].

The FlightGoggles quadrotor comes with a built-in angular rate controller to manage roll, pitch, and yaw. To evaluate the wide variety of trajectory following techniques exhibited by today's quadrotors, we implement four commonly used quadrotor controllers [355] into the FlightGoggles simulator. Two controllers are of a waypoint control type, using a cascade of three PID controllers; the first controls the angle of the quadrotor, the second controls the velocity of the quadrotor using the angle controller, the third sets the velocity of the quadrotor based on the distance to a waypoint. The first implementation, "Unstable Waypoint Controller", replicates poorly written controllers that overshoot and oscillate around waypoints. The second implementation, "Stable Waypoint Controller", mimics tuned controllers that are stable and converge to the waypoint. The next instantiated controller was the "Fixed Velocity Controller". This controller assigns a shared

Table 3.1: Autonomous systems configurations

Hardware	**Software Controller**	**Execution**
Flightgoggles Quadrotor [123]	Unstable Waypoint [355]	Simulation
	Stable Waypoint [355]	Simulation
	Fixed Velocity	Simulation
	Minimum Snap [248]	Simulation
Parrot Anafi Quadrotor [275]	Waypoint [276]	Simulation and Real World

proportion of a fixed velocity over each the x, y, and z direction based on the location of the next waypoint. We set the controller to maintain a velocity of $2m/s$, allowing the quadrotor to maneuver easily. The final controller, "Minimum Snap Controller", computes a minimum snap trajectory and follows it using the waypoint PID controller. It was fundamentally different in that it builds a new trajectory through the waypoints that minimize snap, the 4th derivative of position [248], which means that it does not adhere to the assumption of the expected behavior being the shortest straight line between consecutive waypoints.

A second quadrotor, the Parrot Anafi [275], is studied later in Section 3.1.3. This is a commercial quadrotor, with a weight of $0.5kg$, maximum horizontal velocity of $15m/s$, and an arm length of $0.1m$. The Anafi has an autonomous flight mode, which can follow a series of waypoints. These waypoints are sent using Anafi's proprietary API [276], which changes the quadrotors pose using a controller that is not publicly available. However, Parrot has released the simulator, Parrot-Sphinx [277], used by their engineers during the development of this drone. This allowed us to run trajectories on the Anafi Parrot in simulation and the real-world.

3.1.2.2 Implementation

The implementation consists of 4 main software modules [139], as seen in Figure 3.5. The first module, trajectory generation, implements the approach as described in Section 3.1.1, while the next three modules run the experiments and are used for collecting the data used in the study.

The first module consists of both the trajectory generation and result processing toolchain. The trajectory generation uses the trajectory manager to explore the frontier using the reachable set and scoring model. The resultant trajectories are then processed and converted into data files that can be accessed by both the Anafi and FlightGoggles control software. The majority of this module is implemented using Python. The module consists of 36 python scripts with a total of approximately 7,000 SLOC. For certain functions, such as computing the convex hull, it was more convenient to use MATLAB, and so the approach calls these functions through the MATLAB API for Python [240].

The second module implements the integration with the Anafi quadrotor in both simulation [277] and the real-world through the Anafi API [276]. The module consists of software used to convert the trajectory into waypoints that are readable by the Anafi API. The Anafi API sends the waypoints to the Anafi quadrotor and records the returned GPS data through either a virtual network or real Wi-Fi connection.

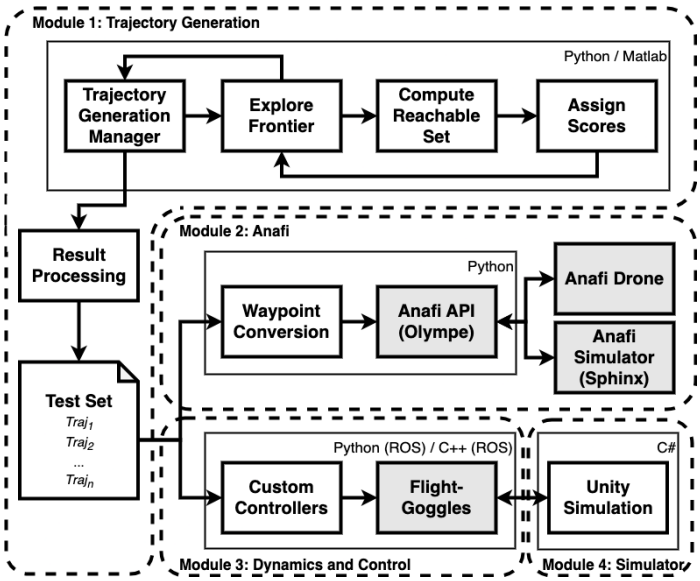


Figure 3.5: An overview of the implementation. Existing software is highlighted in a darker shade.

The final two modules contain the control and simulator code to fly the FlightGoggles quadrotor. The FlightGoggles simulator has two parts. The first part emulates the dynamics and control of the quadrotor, while the second part simulates the quadrotors sensor data and collision information. At the time of writing, the FlightGoggles simulation uses proprietary graphics assets. Thus we only use the part of FlightGoggles that emulates the quadrotor dynamics, and we re-engineer the FlightGoggles simulation tool in Unity based on the available documentation. The control code uses the Robotic Operating System (ROS) [328] and is written in C++. The implementation of the 4 custom quadrotors is integrated into the original code base using 11 Python classes consisting of approximately 2150 SLOC. The portions of the FlightGoggles simulator that were redeveloped in Unity are written in C#. The new simulator integrates with the base ROS code using the original TCP link in FlightGoggles. The new simulator uses assets that are freely available from the Unity Asset Store [351].

3.1.2.3 RQ1: Trajectory Generation with KD Models

To answer RQ1, we need to assess the cost and benefit of incorporating a KD model into the trajectory generation technique. At the time the work was published, there were no automated approaches or tools available for the automated generation of stressful target trajectories for autonomous systems. The state-of-the-practice consists of handcrafted stress tests built by experts, which tend to be effective but limited in the scale of exploration. Thus, to identify the benefits explicitly introduced by the KD models, we adapted how the reachable set in line 13 of Algorithm 2 is computed using 3 different techniques. The first approach, **No KD**, returns all waypoints in the world, without considering any form of a KD model. The second approach weakly approximates the reachable set, **Approx KD**, by computing a sphere whose radius is the distance the quadrotor could travel at maximum velocity in $\Delta t = 1s$. The final approach, **Full KD**, uses a full KD model as described in Section 3.1.1.4. While expensive [118], this guarantees that all explored trajectories are valid by construction. Each technique was given 2 hours to generate and execute trajectories. Algorithm 2 was set to have a beamwidth of 5 and trajectory length between 3 and 50.

Varying trajectory length allows us to assess the efficiency of techniques as the problem scales in complexity. For example, the number of possible trajectories of length 3 in a world with 250 possible waypoints is 1.5×10^7 and that increases to 4.1×10^{117} for trajectories of length 50.

For each technique, trajectories returned in line 22 of Algorithm 2 were checked for validity using the autonomous systems full KD model. For each valid trajectory, we model its execution time in

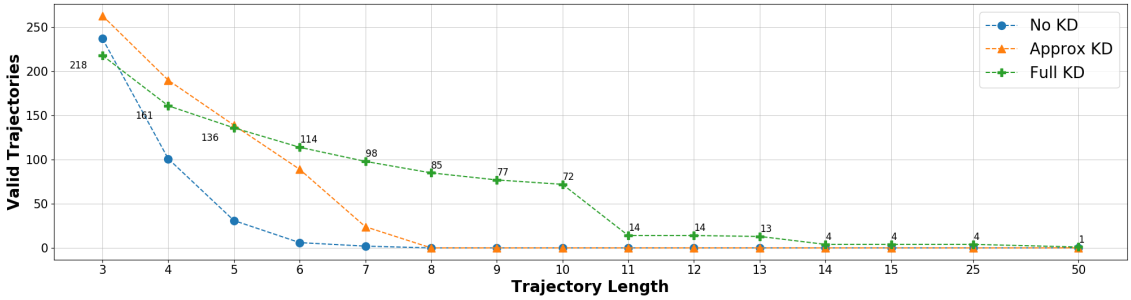


Figure 3.6: Valid trajectories generated of varying length.

proportion to its length and decrement the total experiment time. The number of physically valid trajectories returned by each technique within the 2 hour limit is shown in Figure 3.6.

Figure 3.6 shows that for simpler trajectories of length 3 for **No KD** and trajectories of length 5 for **Approx KD** the computationally cheaper approach produces more valid trajectories as opposed to our **Full KD** approach. This is because generating short, physically valid trajectories is easier, as only a few valid waypoints need to be selected. However, we can see that as the trajectories become longer and more complex, it becomes beneficial to use the computationally more expensive **Full KD** model. Figure 3.6 shows that **Full KD** start to outperform both **No KD** and **Approx KD** for trajectories of length 4 and 6 respectively, in terms of the number of physically valid trajectories produced. In fact, for trajectories of length 8 both **No KD** and **Approx KD** are unable to produce any valid trajectories in the given amount of time, while the **Full KD** can produce 85 valid trajectories. Even for trajectories of length 50, **Full KD** can still find 1 valid trajectory in the given time. This is because the **Full KD** approach provides information to Algorithm 2 on which waypoints in the world lead to invalid trajectories. This information, although expensive to generate, allows the search technique to reject invalid trajectories during trajectory generation as opposed to trajectory execution. This is especially important since the number of possible trajectories grows exponentially with their length – making pruning invalid paths cost-effective.

We then computed several performance metrics for valid trajectories of length 10. We ran each of the valid trajectories from the **Full KD** approach using the FlightGoggles simulator with the stable waypoint controller. Figure 3.7 shows the distribution of 3 performance metrics, namely: maximum deviation (m) from the optimal trajectory, the maximum acceleration (m/s^2) of the autonomous system, and total execution time (s). We chose these because they are diverse in that deviation captures the potential for the autonomous systems to operate unsafely, acceleration captures the stress placed on the autonomous system hardware, and total time reflects the autonomous systems ability to operate effectively.

Figure 3.7 shows that for each of the metrics, the quadrotor exhibits a broad range of possible

values and that each of the distributions is positively skewed, with longer tails to the right (more stress). The fact that the distribution has longer tails to the right shows that even though most trajectories produce little stress, there are trajectories which significantly stress the autonomous system and lie outside the normal operating profile. Figure 3.7 shows that not only do the valid trajectories with no scoring model result in a range of behavior but that we are also able to measure multiple performance metrics on them.

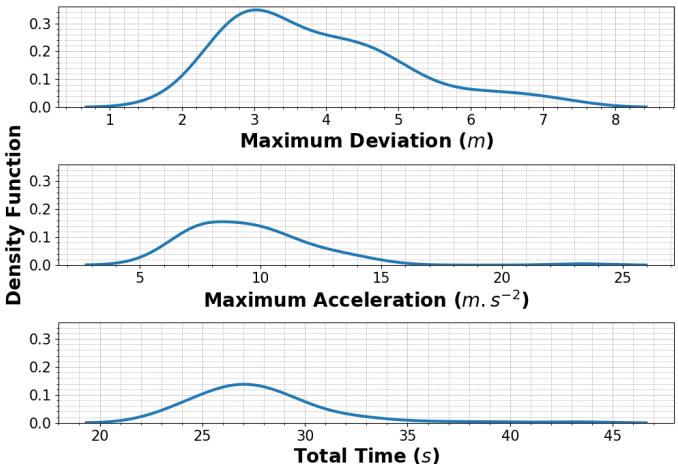


Figure 3.7: The distribution of performance metrics obtained by executing the FlightGoggles quadrotor in simulation.

RQ1 Findings: Although it is computationally more expensive to use a KD model, incorporating it into trajectory generation is critical for efficiently generating *valid* trajectories, especially as the trajectory length increases. We also found that, independent of the chosen measure, the stress induced by valid generated trajectories exhibited high variability. This highlights that, without a scoring metric, there is no guarantee that a valid trajectory will also be stressful.

3.1.2.4 RQ2: Incorporating a Scoring Model

To answer RQ2, we need to determine whether computing and including a scoring model, line 20 of Algorithm 2, leads to the generation of more stressful trajectories. We explore 4 different scoring models as described in Table 3.2. The first 3 scoring models are designed to represent scoring models designed by experts. Intuition tells us that for a quadrotor, the higher an autonomous system's velocity, the more deviation we can expect given a turn. Using this intuition, three handcrafted scoring metrics were created. The first assigned higher scores to **High Velocity** trajectories without consideration to turns. The second assigned higher scores to trajectories that had both high velocity and waypoints that resulted in 90 degree turns (**High Velocity + 90 Deg**). The last handcrafted

Table 3.2: The different scoring models and their descriptions.

High Velocity	Prefers high-velocity trajectories.
High Velocity + 90 Deg	Prefers high-velocity trajectories with 90-degree turns.
High Velocity + 180 Deg	Prefers high-velocity trajectories with 180-degree turns.
Learned	Learns based on past trajectory performance.

scoring model was similar to the second, except it placed a high score on 180 degree turns (**High Velocity + 180 Deg**).

These three approaches require domain knowledge, which is not always readily available. We thus tried a final scoring model, as suggested in Section 3.1.1.5, which **Learned** a scoring model based on the maximum deviation of each controller on the initial trajectories in RQ1. The learned scoring model uses 10-fold cross-validation to determine the polynomial degree used in a ridge regression model implemented using Python's Scikit-Learn library [280]. For each of the software controllers tested in RQ2, we extract attributes from their initial execution. The input and output velocity, the angle between the waypoints, and the actual maximum deviation is extracted, as shown in Figure 3.3. Using this as training data, we produced four independent scoring models that, given a pair of waypoints, predict the maximum deviation for the respective software controller.

For each new scoring model, we generated a new set of trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length 10. That is half of the time given in the RQ1 study to determine if the scoring model could produce more stressful resultant trajectories and do so in less time. For comparison, we also generated a baseline where each of the FlightGoggles software controllers was executed on the trajectory set generated using a **Full KD** model and no scoring model as per RQ1 with trajectories of length 10 and 2 hours of generation.

The resulting trajectories were run on each of the autonomous systems controllers, and the maximum deviation recorded. The choice of maximum deviation was made since it relates to safety – the further an autonomous system (quadrotor in our case) is away from the expected trajectory, the more significant the safety risk. To determine whether the introduction of a scoring model was beneficial, we divided each of the resultant maximum deviations with the mean maximum deviation from the baseline trajectory set. Thus any test that induced more stress and had a maximum

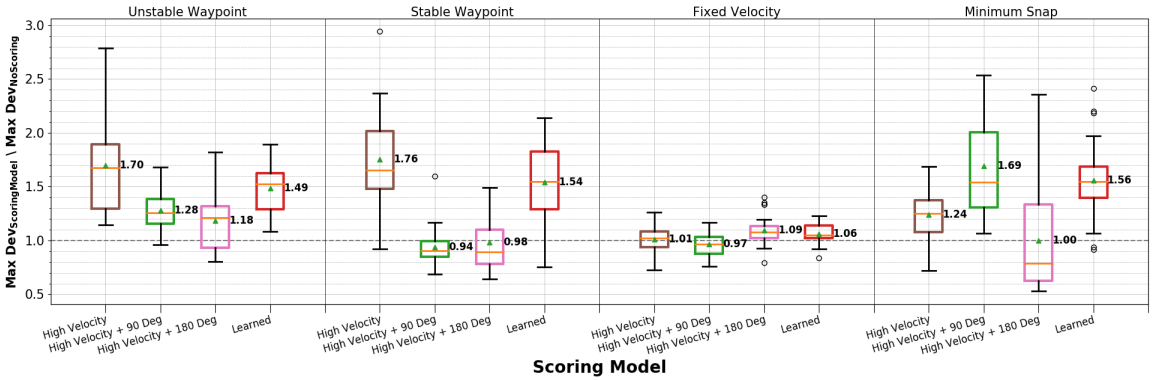


Figure 3.8: The ratio of maximum deviation with a scoring model to maximum deviation without one. Here the initial trajectory set with no scoring model would have a mean value of 1. Any trajectory set that produced more stress than the initial trajectory set would have values greater than 1. The medians (central line) and mean (triangle and number) are shown.

deviation greater than the initial test set with no scoring model from RQ1, would result in a value greater than 1. Similarly, a test with a value of less than 1 means that it induced less stress than the average test in RQ1.

The results are shown in Figure 3.8. When considering only the handcrafted scoring models, Figure 3.8 shows that for each of the controllers, at least 1 of the 3 handcrafted scoring models results in a more stressful test set. For both waypoint controllers, including a scoring model that favors trajectories of high velocity results in test sets that are 70% and 76% more stressful. For the fixed velocity controller, a scoring model that favors 180 degree turns resulted in a test set that is 10% more stressful. The low increase in stress is attributed to the controller's slow constant speed, however, we note that our approach still finds test cases that are ≈40% more stressful than the given random test set. For the minimum snap controller a scoring model that favored 90-degree turns induces on average 69% more stress. These findings are consistent with the operation of these controllers. Moreover, taking the mean of the best scoring models shows that, on average, having a handcrafted scoring model results in a **55.9% increase in maximum deviation** on the stressful trajectories. These findings show that handcrafted scoring models are beneficial when domain knowledge is available.

Figure 3.8 also shows that for all controllers, it is possible to learn scoring models that can

generate stressful trajectories for a specific quadrotor. This is useful, especially when there is no domain knowledge available, for instance, when testing a new autonomous system. Moreover, the quality of learned models is high, since for each controller we found the learned model produced a distribution of performance metrics similar to the best handcrafted scoring model. Taking the mean of all scoring models showed that on average a learned scoring model **increased the maximum deviation by 41.3**%.

Recall that the experimental setup for RQ2 used half of the time compared to RQ1, so the observed improvements in the performance metrics were also significantly less costly to produce.

RQ2 Findings: Introducing both handcrafted and learned scoring model into trajectory generation produces test that on average are **55.9**% **and 41.3**% **more stressful** than trajectories without a scoring model respectively. Moreover, learned scoring models can be generated without any prior domain knowledge.

3.1.3 Real-World Field Study

We performed a preliminary study to explore the application of the proposed approach to a commercial drone operating in an outdoor flying cage of $30m \times 30m \times 30m$, and analyzed the differences between executing the trajectories in simulation versus the real-world. As described in Section 3.1.2.1 we selected the popular Parrot's Anafi quadrotor [275].

As we are not certain about the particular controller used by the Anafi, we learned a scoring model from an initial set of trajectories that we executed sending waypoints to Anafi's API. To reduce the cost of collecting the training set of trajectories, we executed those initial set of random trajectories in the Parrot-Sphinx [277] simulator. We bound the initial generation to 2 hours, a beamwidth of 5, and a trajectory length of 10, and the learning was meant to generate a model that increases the maximum deviation in a trajectory. We then used the learned scoring model to generate stress-inducing trajectories using a total time of 1 hour, a beamwidth of 5, and a trajectory length of 10.

Figure 3.9 shows the findings in the form of a boxplot. The first pair of boxes show the results from the execution of trajectories in simulation, while the second pair of boxes show the results from executing the drone in the real world. Each pair represents the deviation of the initial trajectory set and the stress-inducing trajectory set, respectively. Each box is normalized by the mean maximum deviation of the corresponding initial trajectory set. As mentioned earlier, the initial trajectory set was generated without a scoring model, and it shows similar means and variation in simulation and in the real world.

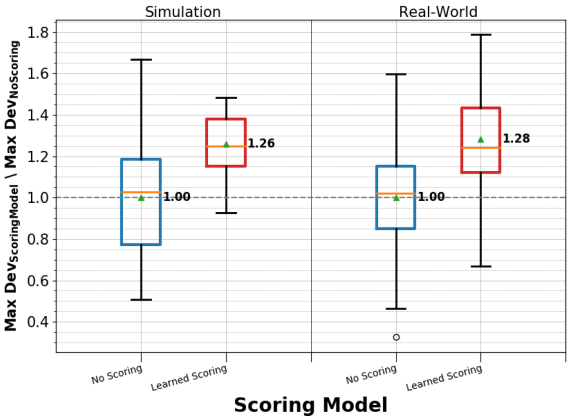


Figure 3.9: Maximum deviation for simulation and outdoors trajectories normalized by the mean of the trajectory set with no scoring model.

As shown by the second box, the scoring model learned in simulation allows our approach to generate trajectories that, when executed in simulation, cause on average a 26% increase maximum deviation in a trajectory. More interesting, however, is that when the same generated trajectories are executed in the real-world, they also cause a similar degree of additional deviations, albeit with greater variation (whiskers of the fourth box) introduced by external environmental factors such as GPS-localization noise and wind. This confirms that it is possible to mitigate the cost of learning a scoring model through simulation and apply trajectories generated with that model in real-world contexts.

From a testing point of view, one might be interested in trajectories (test inputs) that violated certain specifications. For example, might specify that the maximum deviation from the expected trajectory cannot exceed some threshold. Figure 3.10 shows the percentage of automatically generated tests that violate a given maximum distance specification. The results indicate that, regardless of the specified maximum deviation, using a scoring model produces a larger percentage of tests that violate the specification. For example, given a specified maximum deviation of $4m$, Figure 3.10 shows that with no scoring model, only 30% of tests generated would violate that constraint. However, our

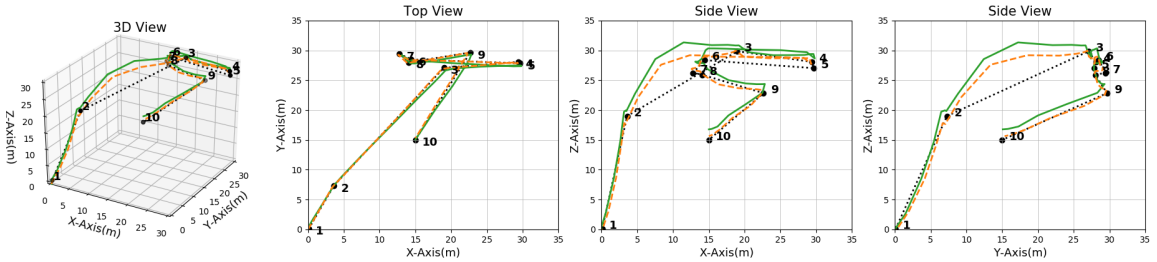


Figure 3.11: Anafi's position in the real-world and simulation as it traverses one of the stress-inducing trajectories. The expected behavior is marked as dots. The simulated data is marked with dashes. The real-world data is a solid line.

approach using a scoring model would generate a test set with approximately 70% of tests that violate the same specification. Additionally, these results show that using a scoring model not only generates a higher percentage of tests that violate the constraints, but also generates the test with the largest maximum deviation. The maximum deviation we observed outdoors with the generated trajectories was $6.2m$, with the average being $4.5m$. This highlights how the approach can generate stressful trajectories that push the drone to deviations that go way beyond the expected deviation for this kind of drone.

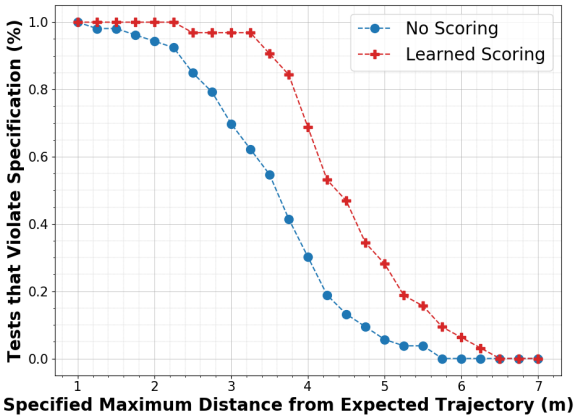


Figure 3.10: The percentage of outdoor tests which violated the specified maximum deviation

Developers can also use these trajectories to further investigate the behaviors which led to these violations. For example, using the Anafi quadrotor, we plotted the test that produced the largest maximum deviation. Figure 3.11 shows the generated test trajectory (dotted line) of the drone in both simulation (dashed line) and real-world (solid line). From the top view, it appears that the Anafi follows the expected trajectory precisely. However, from a side view, it seems like the Anafi follows the expected trajectory in all cases except when large changes in all x,y, and z-directions are requested, for instance, when flying from waypoint 2 to 3. The 3rd waypoint is the position $(19.0, 27.0, 29.9)$. In simulation, although it did not follow the expected short line trajectory, it flew

to a height of $29.9m$ as expected. In the real world, the Anafi similarly did not follow the expected trajectory, however it flew to a height of $31.34m$ high, $1.34m$ over the designated flying altitude of $30m$, even though all waypoints are within the flying volume. A pilot flying this quadrotor who was not aware of the distinct behavior shown through this trajectory would at best be surprised and, at worst, experience a collision.

3.1.4 Summary

We have introduced a novel approach for the automatic generation of feasible and stressful trajectories for autonomous systems. This approach is unique in that it combines kinematics and dynamics to generate trajectories to incorporate an autonomous system's physical semantics. It leverages algorithms from robotics planning and graph exploration for more efficient input space search and incorporates a highly parameterizable scoring model to guide the generation of trajectories that induce high stress on the system. The approach successfully generated valid trajectories, resulting in a mean increase of maximum deviations by 55.9% and 41.3% in the two systems we studied.

However, this approach requires executing many trajectories each time, which can be costly. Additionally, many trajectories might already be executed daily by, for example, a fleet of autonomous vehicles deployed worldwide. Therefore, our next work will focus on selecting tests from a pool of scenarios that have already been executed in the physical environment of the autonomous system.

3.2 Differential Testing on Existing Real Data

As of 2021, 80 companies had registered to test autonomous vehicles on public roads [188]. Each of these companies is producing vehicles that operate on our roads today, collecting real-world sensor data. As described in Section 2.2.2.2, these vehicles produce massive amounts of data (millions of miles worth). However, the physical world is vast, with an extremely long tail of rare scenarios that remain unexplored by even the largest fleets of autonomous systems [195]. This is evident as there are still accidents and reports of strange or incorrect behaviors by users when these vehicles operate in the real world [243, 160, 198, 285, 47, 156].

Current techniques generally consider using sensor data from one source, such as from the specific autonomous vehicle they are testing, or try to create sensor data from scratch through simulation or real-world execution. However, this limits the range of these long tail scenarios one can explore, constrained by the limitations of their own fleet of vehicles or the expense of running simulations and isolated real-world tests. This presents an opportunity: why not consider leveraging data from multiple sources of already existing data, including all the sensor data which has already been collected by all other existing autonomous systems operating in the real-world today? A technique capable of using any sensor data effectively could explore a much wider range of the long tail scenarios, much more cheaply, as it has already been collected. However, using all this data to test arbitrary autonomous systems raises some challenges: How do we determine if the autonomous system's behavior is correct in response to any given input, and how do we ascertain if a scenario is valid with respect to an autonomous system, especially if it was potentially collected by another source?

The challenge of effectively using any arbitrary sensor data presents two main difficulties. First, precisely defining and efficiently determining what behavior should be exhibited remains an open question [162]. This difficulty arises because a range of behaviors might be acceptable in any given scenario. Distinguishing between those that are indicative of normal functioning and those that point to potential issues is challenging due to the lack of a precise *oracle*, Therefore, we first need to devise an *oracle*, a method or function, capable of both efficiently and effectively distinguishing between correct and incorrect behaviors.

The second challenge relates to the specific conditions under which these vehicles are designed to operate, known as the Operational Design Domain (ODD). The ODD, as defined by SAE J3016, includes "operating conditions under which a given driving automation system or feature thereof is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, and/or the requisite presence or absence of certain traffic or roadway characteristics" [1]. Consequently, although massive volumes of real-world sensor data can be collected, an unknown portion may fall outside the ODD of any specific vehicle, where behavior is undefined. Therefore, we also need a method for filtering out data outside the ODD to concentrate on scenarios

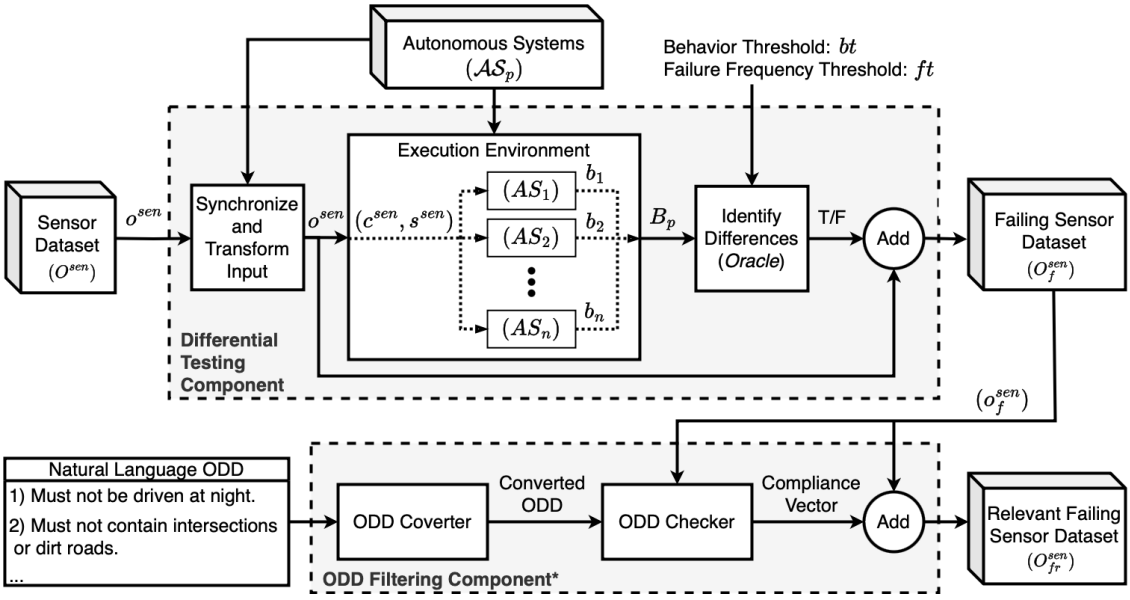


Figure 3.12: Overview diagram of our approach, which contains both a differential testing component and ODD filtering component.
*ODD filtering component was equal contributions from *Carl Hildebrandt* and *Trey Woodlief*.

where autonomous vehicle behavior is both defined and relevant.

3.2.1 Approach

An autonomous system navigates scenarios using onboard sensors sen, which generate sensed scene-state pairs (c^{sen}, s^{sen}). Each pair (c^{sen}, s^{sen}) is processed by the autonomous system AS, resulting in an action $a = AS(c^{sen}, s^{sen})$. This action is capable of altering the system's current state s and the scene c in which it stands. Over time, as the autonomous system operates within a scenario, it observes a sequence of sensed scene-state pairs $o^{sen} = \langle (c_1^{sen}, s_1^{sen}), (c_2^{sen}, s_2^{sen}), \ldots, (c_m^{sen}, s_m^{sen}) \rangle$. Applying this sequence o^{sen} to an autonomous system results in a sequence of actions, known as a behavior $b = \langle a_1, a_2, \ldots, a_m \rangle$.

All autonomous systems operating globally today produce massive amounts of arbitrary observed sensed data, denoted as $\mathcal{O}^{sen} = \{o_1^{sen}, o_2^{sen}, \ldots\}$. Testers can access portions of this sensor data, $O^{sen} \subseteq \mathcal{O}^{sen}$, whether from their own datasets, by combining datasets, or using external

data collected by an unknown autonomous system. However, much of the data in O^{sen} is uninteresting to testers because it generates behaviors $B = \{b_1, b_2, \dots\}$ that are either correct, outside the autonomous system's ODD, or both. Therefore, the goal of this work is to identify a subset of **failure-inducing** sensor data that is also **relevant** to a specific autonomous system's ODD, denoted $O^{sen}_{fr} \subseteq O^{sen}$.

To achieve this, we propose the approach depicted in Figure 3.12. This approach consists of two components. The first component is the differential testing component, which aims to identify the **failure-inducing** sensor data, $O^{sen}_{f} \subseteq O^{sen}$. Here, each sequence of sensor data $o^{sen}_{f} \in O^{sen}_{f}$ is known to result in a failure. The second component is the ODD filtering component.[1] This component further refines O^{sen}_{f} to include only data **relevant** to the autonomous systems' ODD, resulting in a subset of failure-inducing and relevant sensor data, $O^{sen}_{fr} \subseteq O^{sen}_{f}$. Consequently, given a large pool of arbitrary sensor data, our approach identifies a subset of this data that is useful to testers as it both produces failures in, and is relevant to, the autonomous system. Specifically, this process narrows the sensor dataset such that $O^{sen}_{fr} \subseteq O^{sen}_{f} \subseteq O^{sen}$.

3.2.1.1 Differential Testing Component

The first component of the framework is differential testing. This component processes a sensor dataset O^{sen}, where each sequence of sensed scene-state pairs $o^{sen} \in O^{sen}$ has a length of m. It also takes a set of n autonomous systems $\mathcal{AS}_{provided} = \{AS_1, AS_2, \dots, AS_n\}$ that share the same functionality, a behavioral threshold bt, and a failure frequency threshold ft. The component then outputs $O^{sen}_{f} \subseteq O^{sen}$, a subset of the data known to induce failures in at least one autonomous system $AS_i \in \mathcal{AS}_{provided}$.

As the main objective of this component is to identify $O^{sen}_{f} \subseteq O^{sen}$, we need to start by understanding exactly what constitutes a failure in autonomous systems. One way to determine if a system's output is incorrect is through the use of *oracle*s. A test oracle is a mechanism used to determine whether a system's output is correct with respect to some specification, given some input [27, 162]. Defining test oracles for autonomous systems is challenging for two reasons. First,

[1] Developed in collaboration with *Trey Woodlief*

specifications for autonomous systems are often not provided, or they stem from sources intended for human consumption. This means that when they are present, they may be either too high-level or imprecisely defined for actionable implementation. Examples include vague specifications such as "must comply with road rule" [38] or "should be comfortable for passengers" [24]. Second, there is no general way for an oracle to have access to ground truth. For example, consider the challenge of ensuring that an autonomous car always stops at a red traffic light. One could define an oracle that checks whether the car's velocity is zero when a red light is detected. However, if the oracle's definition of the traffic light's state is tied to the autonomous vehicle's perception, and the vehicle misclassified the color of the light from red to green, the oracle will incorrectly flag this as correct behavior. Alternatively, one can provide the oracle with some ground truth by specifically setting up a test in which one knows the light will be red, or by creating a separate device capable of determining the traffic light's state. However, this approach would either require significantly more effort to establish controlled tests and environments where the state of each entity, such as the traffic lights, is known prior to testing, or it would necessitate substantial investment in installing dedicated systems capable of detecting the state of each entity (which may still yield incorrect state readings). While both formalizing specifications and monitoring an autonomous system with respect to them is ongoing work, current approaches are still limited and costly [400, 335, 378].

To overcome these challenges, we propose using a concept known as a pseudo-oracle [79]. Pseudo-oracles work on the assumption that even if a system under test lacks explicit specifications, it must have been designed and implemented to meet some implicit specifications. A good example of this is autonomous vehicles, whose implicit specifications are to comply with road rules while being comfortable for passengers. In such cases, if we have access to two or more instances of such systems, they can be compared to each other, and when differences occur, we have potentially found a case where one violated the implicit specifications. This idea can be linked with differential testing from traditional software engineering [244]. In our differential testing component, each autonomous system in $\mathcal{AS}_{provided}$ acts as a pseudo-oracle for the others. When discrepancies in behaviors accross those systems occur, they suggest potential failures, assuming at least one of the implementations is correct. This method does not rely on a single authoritative source for the correct output but

rather uses the agreement among multiple sources as an approximation for correctness.

However, creating this differential testing component for autonomous systems is non-trivial. First, we need a way to apply any arbitrary sensor data $o^{sen} \in O^{sen}$ to any given autonomous system. For example, autonomous systems, including different versions of the same system, often require inputs in diverse formats, with variations in data rates, resolutions, and data types. Additionally, many of these systems receive inputs from multiple sensors of differing numbers and types, such as cameras, LiDAR, radar, GPS, and odometry. Consequently, the first stage of our differential testing component is dedicated to synchronizing and transforming the sensor data to meet these varied input requirements.

Second, we need a way to run multiple autonomous systems concurrently. Given their complexity, these systems demand access to specialized hardware, such as high-speed networking, GPUs, and extensive memory. This complexity generally necessitates isolated operation. For instance, these systems often use hardcoded network ports for communication, leading to conflicts when multiple versions operate concurrently on the same platform. Moreover, autonomous systems have stringent GPU latency requirements to process real-time data efficiently, which can cause performance bottlenecks when several systems compete for the same GPU resources. Additionally, the high throughput of sensor data on internal buses may exceed the data handling capacities of a single platform when multiple systems are running. These challenges highlight the difficulties of enabling multiple versions of an autonomous system to run concurrently without interfering with each other. This process is managed by the execution environment stage of the approach.

Finally, we need a way to create the *oracle* capable of detecting behavioral differences in autonomous systems. This task is challenging for two reasons. First, unlike traditional software, which typically produces discrete outputs, autonomous systems generate results that have continuity constraints. This characteristic can make subtle behavioral discrepancies over time more significant, while also potentially making massive instantaneous differences less impactful. For instance, consider an autonomous vehicle that produces both steering angle and velocity commands. A minor, momentary change in steering may be inconsequential for a slow-moving autonomous truck but could precipitate a disastrous collision in the context of a high-speed autonomous racing car. Conversely,

continuous minor steering adjustments that might destabilize a truck driving in a straight line for extended periods could be considered normal, even necessary, for a racing car adeptly maneuvering around a bending track. Second, like traditional software, autonomous systems can produce multiple outputs. However, unlike traditional software, these outputs have physical units and real-world meanings associated with them. This integration with the physical world means that while each output may appear isolated, they collectively influence the system's behavior in significant ways. For example, a car turning right at a set steering angle and low velocity will behave very differently from another car turning at the same angle but with high velocity. This complexity underscores the challenge of validating outputs in autonomous systems, as they cannot be assessed in isolation or in small groups without considering their combined effects on real-world behavior. This serves to highlight that our approach needs ways to parameterize how differences are detected. Not only must the approach account for varied numbers and potentially vastly different outputs, but it must also recognize that what constitutes a failure for one set of autonomous systems may not for another.

The subsequent sections describe each of these stages in more detail. We note while each stage is discussed in the approach and implementation, the first two stages, "synchronize and transform input" and the "Execution Environment", are more engineering-intensive and therefore are primarily discussed in the implementation section, while the final stage, being more of a conceptual contribution, is discussed primarily in the approach section.

3.2.1.2 Synchronize and Transform Input

Differential testing relies on providing n systems with the same input. The goal of this stage is precisely that: to provide each of the n autonomous systems, despite their different input requirements, with identical input. Specifically it aims to supply each autonomous system, denoted as $AS_i \in \mathcal{AS}_{provided}$, with the same o^{sen}.

To fully understand how this process works, let's first start by considering how this would work if we had access to each of the physical autonomous systems in a physical scenario. In such a case, we could place each of the n autonomous systems AS_i into the same real scene c within the scenario along with setting the AS_i's state s. This scene-state pair (c, s) would be processed by the sensors

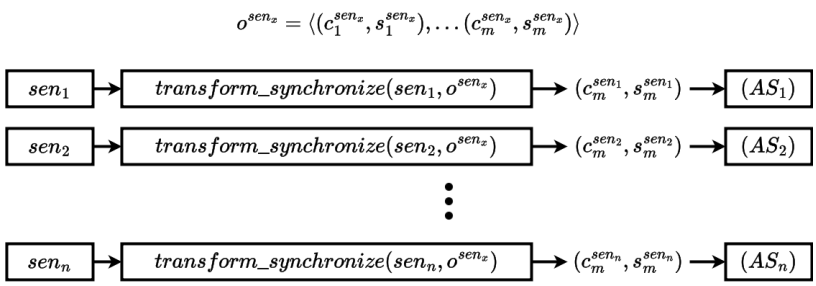


Figure 3.13: Synchronization and transforming each scene-state pair in o^{sen_x} to provide the autonomous systems with the same input.

sen_i of each autonomous system. As each AS_i may be equipped with different sensors sen_i, they would therefore produce slightly different representations (c^{sen_i}, s^{sen_i}) that all stem from the same original input (c, s).

In this work, the approach does not have access to the original (c, s), as shown in Figure 3.13. Instead, it is given $o^{sen_x} = \langle (c_1^{sen_x}, s_1^{sen_x}), (c_2^{sen_x}, s_2^{sen_x}), \ldots,$
$(c_m^{sen_x}, s_m^{sen_x}) \rangle$, which is sensor data collected from some arbitrary autonomous system sensors sen_x. Here, we need a way to convert it so that it can be consumed by each of the $AS_i \in \mathcal{AS}_{provided}$. To achieve this, our approach defines a function $transform\_synchronize$ that takes in the sensor input requirements and (c^{sen_x}, s^{sen_x}) to produce (c^{sen_y}, s^{sen_y}), such that $(c^{sen_y}, s^{sen_y}) = sen_y(c, s)$.

The $transform\_synchronize$ function performs two main operations. First, it modifies $(c^{sen_x}$, $s^{sen_x})$ such that the transformed (c^{sen_y}, s^{sen_y}) meet the specific sensor modalities and types of sen_y. Second, it ensures that (c^{sen_y}, s^{sen_y}) is synchronized with the expected functioning of AS_y if it were operating in (c, s). We will describe how this transformation function can be implemented in more detail later in Section 3.2.2.1.2. As the goal was to provide each $AS_i \in \mathcal{AS}_{provided}$ with the same input, we will simply refer to the input for all autonomous systems as o^{sen}.

3.2.1.3 Execution Environment

The goal of the execution environment is to run n autonomous systems $\mathcal{AS}_{provided}$ concurrently on the same input (after the proper transformation) and to collect the output $B_{provided}$. We showcase this process in Figure 3.14. It starts by taking in a single input o^{sen}, which consists of m sensed

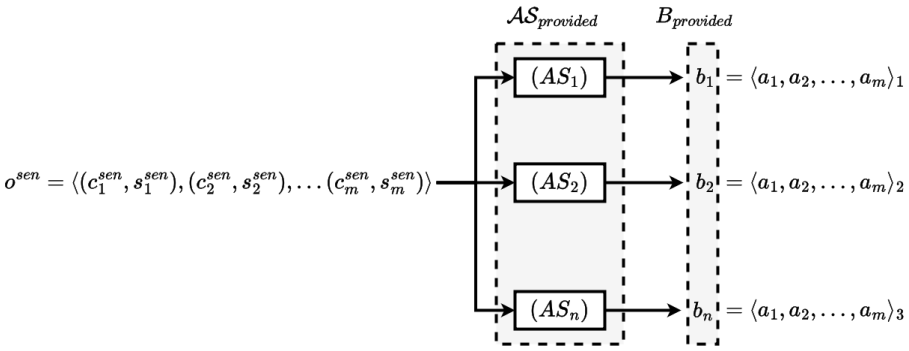


Figure 3.14: Synchronization and transforming each scene-state pair in o^{sen_x} to provide the autonomous systems with the same input.

scene-state pairs. This is then fed into each of the n autonomous systems, which subsequently produce n behaviors $B_{provided} = \{b_1, b_2, \ldots, b_n\}$. Recall that, each of these behaviors b is actually a sequence of actions $b = \langle a_1, a_2, \ldots, a_m \rangle$, corresponding to each of the sensed scene-state pairs from o^{sen}.

3.2.1.3.1 Identifying Behavioral Differences

Now that we can provide all autonomous systems $AS_{provided}$ with the same input o^{sen}, and collect the output $B_{provided}$, we need a way to determine if any of the behaviors $b_i \in B_{provided}$ are failures. To do this, we define an *oracle*. As described in Section 2.2.1.2, is a test oracle is a function or procedure that distinguishes between the correct and incorrect behaviors of a system under test [162, 28, 27]. Using the terminology from Barr et al. [28], an oracle is defined as using Equation 3.1. Here the oracle is a partial function D that maps from a test activity sequence T_a to true or false, representing whether it accepts the test activity or not. Under their definition, a test activity sequence T_a is a sequence of stimulus-response pairs produced by a system under test.

$$D : T_a \mapsto \mathbb{B} \tag{3.1}$$

Building on Equation 3.1, we can define an oracle for a single autonomous system using Equation 3.2.

$$oracle : (o^{sen}, b) \mapsto \mathbb{B} \tag{3.2}$$

In Equation 3.2, our "stimulus" is the sequence of sensed scene-state pairs o^{sen}, and the "response" is a behavior b. The question then becomes how to define the mapping function? In traditional software testing, the mapping function can take the form of a predicate, which is a specific condition or set of conditions that can be evaluated as true or false. The predicate can be derived from several sources of information. For example, from specifications, including state-based specifications [353, 352, 206, 161], which define the predicates as preconditions and postconditions, or model-based specifications [325, 128, 44, 99], which define predicates close to the implementation languages programmers use. However, as described earlier these approaches cannot be used due to the lack of explicit specifications for autonomous systems.

Therefore, as mentioned before we build upon a pseudo-oracle [79]. Pseudo-oracles address the so-called non-testable programs problem and build upon the fact that if specifications cannot be derived for a program, we can simply replace them with alternative programs implemented to share the same implicit specifications. More specifically, if we use the fact that all $AS \in \mathcal{AS}_{provided}$ share the same implicit specifications, such as safely navigating real-world road conditions while adhering to road rules, we can use each as oracles for the others. To do this, we provide each of the $\mathcal{AS}_{provided} = \{AS_1, AS_2, \ldots, AS_n\}$ with the same single input o^{sen} and then compare the collection of outputs $B_{provided} = \{b_1, b_2, \ldots, b_n\}$ to determine if any differ. Therefore, our oracle for a single autonomous system in Equation 3.2, is simply updated to take in all behaviors $B_{provided}$, rather than a single behavior as shown in Equation 3.3.

$$oracle : (o^{sen}, B_{provided}) \mapsto \mathbb{B} \tag{3.3}$$

Now that we have all the behaviors $B_{provided}$, the mapping function can be defined as one that compares all the behaviors to each other and looks for differences, as described in Equation 3.4.

$$mapping(o^{sen}, B_{provided}) : [\forall i, j \in [1..n], i \neq j | \Delta(b_i, b_j, bt) \geq ft] \tag{3.4}$$

This mapping function takes in the input o^{sen} and the set of outputs $B_{provided}$, and compares them to the user-defined behavioral threshold bt and failure frequency threshold ft. It does this by passing all combinations of $b_i \in B_{provided}$ and $b_j \in B_{provided}$, where $i \neq j$, to a function Δ, along with the behavioral threshold bt. The Δ function computes the differences between the behaviors and compares these to bt. Specifically, in our instantiation of this function in the study, we perform a pairwise comparison between each of the actions in the two behaviors. Each comparison is evaluated against bt, and the function returns a count representing how many actions exceeded bt. For example, if two behaviors consisted of five actions each (i.e. $m = 5$), and three of these actions differed by more than bt, Δ would output 3. Finally, this count is compared to the failure count threshold ft, which determines if the two behaviors differ enough for the oracle to accept that the behaviors are sufficiently similar or not. This double threshold allows us to overcome the challenge introduced in Section 3.2.1.1, which highlighted the need for an *oracle* capable of detecting both magnitude differences and variations over different frequency counts.

For example, consider $n = 3$ autonomous cars, such that $B_{provided} = \{b_1, b_2, b_3\}$, where each $b \in B_{provided}$ represents the throttle percentage used to control the cars acceleration. If your execution environment was set to process o^{sen} of length $m = 4$, you could get the following output shown in Equation 3.5:

$$B_{provided} = \{b_1, b_2, b_3\} = \{\langle 25, 30, 35, 40\rangle, \langle 25, 25, 70, 70\rangle, \langle 10, 35, 75, 100\rangle\} \tag{3.5}$$

The mapping function in Equation 3.4 would then be instantiated, with each $b \in B_{provided}$ being passed into the Δ function along with bt. Let's assume for this example that $bt = 10$, i.e., a developer wanted to identify throttle differences of more than 10 percent. If the Δ function returns a count representing how many items in the sequence differed by more than bt, you would get what is shown in Equation 3.6.

$$\begin{aligned} mapping(o^{sen}, B_{provided}) &: [\Delta(b_1, b_2, 10) \geq ft, \Delta(b_1, b_3, 10) \geq ft, \Delta(b_2, b_3, 10) \geq ft] \\ &: [2 \geq ft, 3 \geq ft, 3 \geq ft] \end{aligned} \tag{3.6}$$

In such a case, unless $ft = 4$, one of these would evaluate to True, indicating a difference, of greater than 10 percentage occurred more than ft times, between at least one pair of autonomous systems.

However, in order to get this result, this mapping function needs to compare all combinations of n behaviors, it will result in $\binom{n}{2}$ true or false values, describing which combinations of behaviors violated and did not violate the thresholds. This could be an issue, as it requires $\binom{n}{2}$ calls to the Δ function, which is expensive. This is especially a concern as our approach aims to process large amounts of O^{sen}. For example, in the study, we process 4.6 million scene-state pairs $(c^{sen}, s^{sen}) \in o^{sen}$ over $n = 3$ autonomous systems, which would require 13.8 million comparisons. We, therefore, propose two alternative mapping functions, both aimed at reducing the cost of this process.

The first alternative mapping function assumes that a developer only cares if one of the autonomous systems fails. This could be because the developers obtained $\mathcal{AS}_{provided}$ by including multiple versions of the same underlying autonomous system, and therefore only cares about the latest, AS_nth version. Alternatively, it could be because the developer has $n-1$ competitors' autonomous systems and wants to improve only their own autonomous system, AS_n. In such cases, the mapping function could be simplified to require only $n-1$ calls to the Δ function, and is shown in Equation 3.7.

$$mapping(o^{sen}, B_{provided}) : [i \in [1..(n-1)] | \Delta(b_i, b_n, bt) \geq ft] \tag{3.7}$$

This mapping function is very similar to the ones used in regression testing, which would only compare the latest version to a single previous version. However, in our mapping function, we compare the latest version to all other $n-1$ versions. This is done to keep the mapping function broad and applicable to both of the scenarios described above.

The second alternative mapping function does not make this assumption that developers only care about one of the autonomous systems and instead focuses on detecting failures in any of the autonomous systems, albeit at a lower computational cost. In such a case, at the expense of not knowing which $AS \in \mathcal{AS}_{provided}$ causes the failure, the mapping function can be defined as:

$$mapping(o^{sen}, B_{provided}) : \Delta(max(B_{provided}), min(B_{provided}), bt) \geq ft \tag{3.8}$$

This mapping function computes the extremes of all behaviors and compares them. By doing this, the function can guarantee that if a difference occurs, it will be detected while reducing the number of calls to the Δ function to just one. However, this does require that the extremes are calculated using the *max* and *min* functions, but this can be reduced to a time complexity of just m operations.

Once the developer has decided on the parameters of a mapping function, the final step is to use that mapping function in the oracle. Assuming that the mapping outputs true values when a failure occurs, and the approach wants the oracle to output true when a failure occurs, the oracle can then be defined as shown in Equation 3.9.

$$oracle : \bigvee mapping \tag{3.9}$$

This process takes all the true and false values from the mapping function and ORs each of them together. More specifically, if there was a failure during any of the behavioral comparisons, that will cause the oracle to output that a failure was detected.

The final consideration to make is that behaviors b_i may contain different outputs, such as steering or throttle values. In such cases, developers might want to define unique bt and ft for each behavior's output. To accommodate this, a developer could instantiate multiple oracles, one for each behavior's output. Then, they could detect failures across all outputs using Equation 3.10 shown below:

$$oracle : \bigvee_{i \in outputs} oracle_i \tag{3.10}$$

In this case, the oracles only accept the behavior if each $oracle_i$ for all potential behavior outputs accepts it.

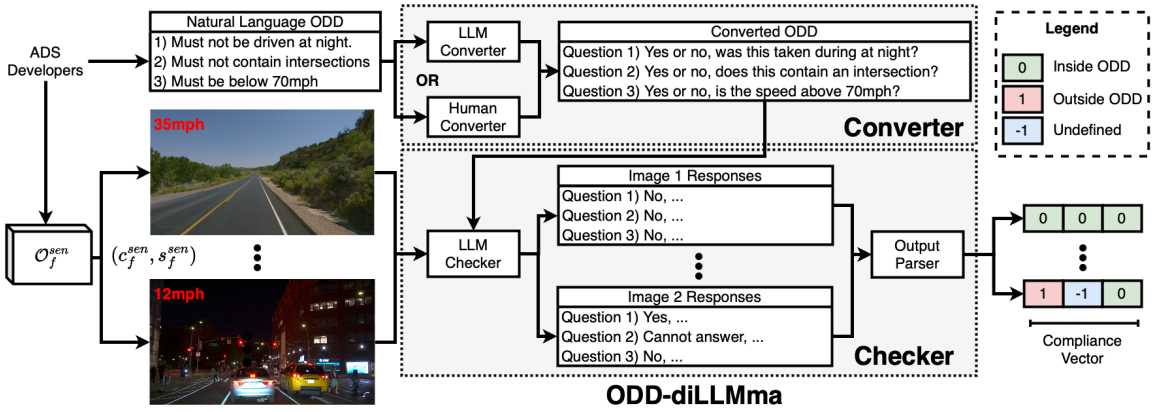


Figure 3.15: Visual depiction of ODD-diLLMma's filtering process: inputting sensor data and natural language ODD specifications, and outputting a compliance vector indicating adherence to ODD.

3.2.1.4 ODD Filtering

The second major component in the framework is the ODD filtering component. The goal of the ODD filtering component is to process each $(c_f^{sen}, s_f^{sen}) \in O_f^{sen}$ produced by differential testing component and exclude all data that fails to comply with the vehicle's ODD criteria, producing a set of relevant in-ODD failures $O_{fr}^{sen} \subseteq O_f^{sen}$. As presented in this dissertation, this is the first automated ODD filtering mechanism designed for verifying sensor data with arbitrary ODD's.

Figure 3.15 illustrates an instance of the ODD filtering system, which is called ODD-diLLMma. This system processes failure inducing sensor data (c_f^{sen}, s_f^{sen}), in conjunction with a set of ODD specifications expressed in natural language. The output is a compliance vector that evaluates the congruence of (c_f^{sen}, s_f^{sen}) with the ODD specifications for the autonomous system.

ODD-diLLMma's encompasses converting each semantic dimension outlined in the ODD specifications into a series of boolean queries via the ODD converter component. These queries, together with (c_f^{sen}, s_f^{sen}), are then fed into a multimodal LLM within the ODD checker component. The output from the LLM is analyzed to formulate the compliance vector, which indicates whether the data adheres to each specified semantic dimension of the ODD. Below we first give some background on LLM, and then describe each component of the ODD-filtering section in more detail.

3.2.1.4.1 Background on LLM

LLMs are advanced artificial neural networks designed for processing, understanding, and generating human-like natural language [402]. LLM's take as input a *context* and a *prompt* and then provide as output a *response* to the prompt. Recent advancements have extended LLMs capabilities to multimodal forms [169]. Multimodal LLMs can process a wide array of data types, including text, images, and videos, enabling them to understand and generate *responses* based on a richer set of *context*. Prominent examples include OpenAI's GPT series [268, 269, 270], Google's Bard [284], and Meta's Llama language models [347, 348, 405].

LLMs and their multimodal variants stand out from traditional neural networks because they use the transformer architecture. Unlike traditional models that use serial processing, transformers use parallel processing [357]. This design not only reduces training time but also addresses performance issues associated with long dependencies that earlier models like recurrent neural networks [247] and long short-term memory models [146] faced.

LLMs have shown remarkable versatility across various sectors. For instance, in the service industry, they are increasingly used in sectors like law [205], education [393, 2], finance [382], healthcare [201, 398], content generation [6, 303] and language translation [391, 7]. The field of software engineering has also benefited from LLMs in areas like debugging [170, 218], security analysis [223, 83], testing [363, 216], and documentation [95, 150].

In autonomous robotics, LLMs are starting to play a crucial role in enhancing robots' abilities for interacting with humans using natural language. Projects like SayCan demonstrate LLMs guiding mobile manipulator robots in performing a variety of tasks inspired by everyday activities, such as in kitchen settings [8]. TidyBot, another LLM-powered agent, personalized cleaning processes by learning user preferences through textual interactions [380]. This is not limited to single robot systems; Smart-LLM introduced a framework using LLMs for controlling multiple robots [175]. The growing availability of open-source libraries for LLM-based solutions further emphasizes their expanding role in practical applications [381, 291].

Another interesting area LLM are starting to be used in is autonomous driving. Traditional

autonomous driving systems have typically adopted a modular approach, which segments tasks into distinct units responsible for perception [230, 224], prediction [322, 166], and planning [51, 371]. While this approach facilitates a clearer understanding of decision-making processes, it also has inherent limitations, such as loss of key information and redundant computations during the transition between modules [52, 390]. Multimodal LLMs offer a promising avenue developing end-to-end solutions which may overcome these inherent limitations of the more traditional modular approach [210, 339, 191, 63, 41, 385].

3.2.1.4.2 ODD Converter

The ODD converter takes a set of ODD specifications written in natural language. Publicly available ODDs are written in natural language [342, 119, 69], often in the form of lists describing the ODD semantic dimensions. For our approach, we must convert the specifications into a structured format so the LLM Checker's responses can be unambiguously matched with the semantic dimensions. We transform the list of semantic dimensions into a series of yes-no questions due to prior demonstrated success in LLMs responding to this paradigm [295, 407]. This conversion is a one-time task that can be accomplished either manually or automatically, e.g. by an LLM-based Converter.

For example, a typical ODD specification might state: "Many factors can impact the performance of openpilot ALC and openpilot LDW, causing them to be unable to function as intended. These include, but are not limited to: Poor visibility (heavy rain, snow, fog, etc.) or weather conditions that may interfere with sensor operation..." [66]. This statement would be reformulated into a question like: "Yes or no, does the image exhibit poor visibility conditions such as heavy rain, snow, fog, or other weather conditions that may interfere with sensor operation?" By translating ODD specifications into this question format, we streamline the process for the LLM Checker to analyze sensor data in the context of these specifications while providing a specific and consistent interface for collecting data, enhancing the efficiency and effectiveness of our approach.

3.2.1.4.3 ODD Checker

The ODD checker takes a set of ODD converted ODD specifications formatted as as series of "yes" and "no" questions, and a failure inducing sensor dataset of sensor readings (e.g., images, point clouds) that serve as context for the LLM Checker. Our approach uses these inputs to compute a single compliance vector describing how the sensor input complies with the ODD. Each of the n ODD semantic dimension questions are passed to the LLM Checker with the sensor input as context as shown in Figure 3.15.

The LLM Checker is prompted to output either "yes" or "no", which is then converted to "Inside ODD" (0), "Outside ODD" (1), or "Undefined" (-1). Given the inherent unpredictability of LLM outputs [120], we cannot guarantee that the LLM Checker will output explicitly and solely "yes" or "no". As such, we use a multi-method parsing strategy to interpret and validate the response. Strategies include looking directly for "yes" or "no", applying regular expressions to identify numbering patterns, and filtering out parts of the response based on context clues. Each compliance vector can then be checked to see if the data is compliant and included in as part of the relevant failure dataset, or removed.

3.2.1.5 Limitations

The approach has several limitations. Before looking at the limitations of each component, we first note a primary limitation of the entire approach. Our approach first passes sensor data to several autonomous systems before assessing ODD compliance. This means that there is a chance an autonomous system will be asked to operate outside its ODD. However, this is necessary because autonomous vehicles require a continuous stream of sensor data to function properly, and by removing all non-ODD compliant data, we would effectively create a discretized set of sensor data. This imposes an overhead a limitation as it requires all autonomous systems to process all sensor data, which is resource-intensive. Secondly, this means that the autonomous systems may be exposed to non-ODD compliant data just before entering a scenario within the ODD. Therefore, failures identified in ODD may be attributed to a sequence of inputs outside the ODD just prior to the

observed failure. As autonomous systems become more advanced and their ODD expands, this limitation will become less relevant. Next, we will examine the limitations of each component in isolation.

Differential Testing: Our approach is capable of detecting differences in multiple autonomous systems that share the same specifications. However, it does have a few limitations. First, our approach assumes that sensor data is readily available and that it is a superset of the data consumed by the autonomous systems. Second, it assumes the availability of multiple autonomous systems that share the same specifications. Both of these assumptions are becoming less of a limitation as more vehicles, from various companies capable of generating these sensor datasets, begin to operate on our roads.

Finally, we recognize several limitations of the pseudo *oracle*. The first limitation occurs when all autonomous systems violate the same specification; our oracle would mistakenly classify this as correct behavior. This is a core limitation of differential testing, and can be minimized through the use of more and varied autonomous systems. The second limitation involves improperly setting of the threshold values. If set too high, the oracle might overlook differences indicative of a failure, whereas a threshold set too low could lead to numerous false positives. The third limitation arises when differences in behaviors, though within specifications, are incorrectly identified as failures. An example is a fork in the road where both choices, turning left or right, could be equally valid according to the specifications. This limitation could be overcome through our *oracle* taking into account the input o^{sen}. Through this, over time, the oracle could learn to recognize and potentially disregard such scenarios.

ODD Filtering: While our approach is the first to provide an automated way for checking arbitrary sensor data with respect to an ODD written in natural language, there are several limitations. First, our approach is expensive. Generally, the inference of deep neural networks is a relatively cheap operation. However, with LLMs, even inference can be expensive due to the size of the models being used [53, 96]. This is particularly challenging for the amount of data we need to process. For example, a 10 minute video at 60FPS would require $10 \times 60 \times 60 = 36,000$ inference calls to process the entire data-stream.

The next issue is that these LLMs are trained to be general, and while this is beneficial in the sense that it allows for a wide range of ODD specifications to be checked, it means we potentially lose accuracy on certain semantic dimensions which are important to us. While we could fine-tune these models to improve accuracy on the semantic dimensions we care about, this runs into the same issue as above, where it is an expensive operation, but also requires large amounts of labeled data which does not yet exist for ODD checking.

The final limitation is with the ODD itself. Publicly available ODDs often lack detail and therefore can be interpreted in many different ways. For example, consider the example above about poor visibility. What exactly constitutes poor visibility for openpilot is not defined well enough to validate if our ODD Checker is correct or not. One LLM, or human for that matter, may find something to have poor visibility, while another may not. Both may be correct and have arguments as to why they are correct, and this will remain true until more information is provided in the ODD. This is discussed later in the study in Section 3.2.2.3.

3.2.2 Study

This study evaluates the effectiveness of our approach for detecting failures in autonomous vehicle behavior. To do this, we address three key questions over three datasets executed through three versions of a commercial-grade autonomous system, filtered using two LLMs with two distinct prompting strategies. Each component of our methodology is detailed below. Specifically, we explore the following questions:

RQ1) How effective is the differential testing component at finding failures?

RQ2) How accurate is the filtering component at classifying data within or outside the ODD?

RQ3) How effective is the entire framework at selecting failure inducing tests within the ODD from large sets of existing sensor data?

3.2.2.1 Setup

We explore our approach using 3 versions of comma.ai's openpilot as our autonomous system. Openpilot is a commercial, open-source, road-deployed autonomous system that is capable of Automatic

Lane Centering (ALC). We selected openpilot as it uses camera-based inputs to determine its behavior, allowing us to leverage existing camera-based sensor datasets. Additionally, openpilot is compatible with over 250 vehicle models [73] and has driven over 50 million miles while deployed [68] indicating the maturity of the system.

3.2.2.1.1 Datasets

We selected two datasets provided by comma.ai to illustrate the existence of viable test cases within datasets currently in use by the same company that produced the autonomous vehicle. The chosen datasets are comma.ai's 2016 dataset [305], which consists of 11 videos totaling 7 hours, and the comma.ai 2019 dataset [310], containing 2035 videos and extending to 34 hours. These selections were based on the premise that, coming from the same source as the autonomous driving system, the data would likely adhere with the system's ODD and exhibit few, yet hopefully some, failures.

Furthermore, to demonstrate our method's ability to uncover test cases from alternative sources of existing real-world sensor data, we examined the most recent 50 videos, totalling to 43 hours, from the External JUtah dashcam video collection [172], which is unaffiliated with comma.ai. This collection has thousands of dash cam recordings, showcasing the extensive range of sensor data already gathered that is publicly accessible. This choice allowed us to explore our approach's ability in identifying real-world test cases from existing data pools. Due to its lack of affiliation with comma.ai, we anticipated this dataset would yield a higher incidence of failures and a wider variety of data that deviates from the specified ODD.

Each of these videos represents an $o^{sen} \in O^{sen}$, however, as these are just videos, they lack state information. More specifically, for an autonomous system to compute behavior, it requires the input pair $(c^{sen}, s^{sen}) \in o^{sen}$. The datasets provided by comma.ai, as well as the alternative sources, only contain sequences of c^{sen}. To overcome this, we pair each c^{sen} with a hardcoded s^{sen}. Specifically, openpilot requires a velocity reading, which we set to 30 mph. This speed was chosen as it is feasible, though slightly slow, for highway driving and reasonably, though slightly fast, for main roads in cities and suburbs. We leave the exploration of alternative approaches to setting state, as well as how state affects our approach for future work.

3.2.2.1.2 Synchronizing and Transforming Data

An arbitrary sensor dataset $\mathcal{O}^{sen}$ may use various combinations of sensor modalities. The goal of this initial stage is to define a $transform\_synchronize$ function that can convert sensor readings into formats that meet the input specifications of an autonomous system's sensors. To achieve this, the transformation phase must identify the necessary sensor specifications. Although this process could theoretically be automated by examining metadata from sensor messages or through static analysis of the code, we implemented it manually by analyzing the code, its execution, and associated documentation.

Once the specifications were identified, the next step is to remove any sensor data not utilized by the autonomous systems. For example, some sensor data may include LiDAR information, whereas the autonomous system under test may not consume LiDAR. In our case, one of the datasets included the approximate geographic location where the sensor data was recorded, which was removed as it was not consumed by the autonomous systems' sensors.

The next step is to transform and synchronize the remaining sensor readings to match the identified input specifications. Several different transformations may occur, such as reducing or increasing sensor fidelity, changing the frequency of the data, or altering the reference frame of the sensor data. In our study, the resolution of both the comma.ai 2k19 and 2016 datasets was increased to meet the 1928×1208 resolution required by openpilot, whereas the resolution of the External Utah dataset was reduced. Additionally, each video was synchronized by setting each to a standardized 15 FPS, aligning with the frequency at which openpilot outputs steering angle data. These modifications were performed using FFmpeg [346], an open-source and commonly utilized multimedia framework for processing video files in various formats. The post-transformed data was then manually inspected to ensure that it yielded reasonable data.

The final stage is to approximate or generate any missing sensor data. This stage is the most challenging and, in some cases, cannot be achieved. However, with recent advancements in generative AI [212, 15, 293], this task should become much more achievable in the future. In our case, the velocity information for the system's state was not included. As mentioned earlier, we approximated

this to be 30 mph. This speed was chosen because it is feasible, albeit slightly slow, for highway driving and reasonably, albeit slightly fast, for main roads in cities and suburbs.

This transformation and synchronization process generated approximately 4.6 million frames in total, highlighting the number of potential tests that can be found from 3 datasets alone. To further facilitate subsequent synchronization during the next stage, each frame was assigned a unique frame ID. This ID would be used by the autonomous systems to record the steering angle associated with each specific frame, allowing each of the steering angles to be precisely matched up post execution.

3.2.2.1.3 Executing Multiple Autonomous Systems

An autonomous system takes in sensor readings c^{sen} and a sensed system state s^{sen} as input and produces a behavior b, such that $b = AS(c^{sen}, s^{sen})$. This process is known as executing the autonomous system. The objective of this step is to execute multiple autonomous systems concurrently. Specifically, we aim to execute n autonomous systems, where $n \geq 2$. Therefore, given the set of autonomous systems $\mathcal{AS}_{provided} = \{AS_1, AS_2, \ldots, AS_n\}$, our goal is to produce a set of behaviors $B_{provided} = \{b_1, b_2, \ldots, b_n\}$, for each input.

The process of generating $B_{provided}$ requires considering two dimensions. First the ratio of computation required by $\mathcal{AS}_{provided}$ to the computation available in the execution environment. Second the implementation of the $\mathcal{AS}_{provided}$, specifically concerning mutual interference.

In our study, we used three versions of openpilot's ALC from April 2022 [67], March 2023 [72], and June 2023 [71]. Each of these systems required independent access to a GPU and had hard-coded network interfaces. Therefore, to ensure each autonomous system had access to a GPU, each was executed independently on separate PCs, each with a clean installation of Ubuntu 20.04 and NVIDIA's latest graphics drivers 535. This setup also overcame the challenge of hard-coded network interfaces by providing physical separation of the autonomous systems.

3.2.2.1.4 Identifying Differences

The output of each autonomous system, $B_{provided}$, contained only steering information. Since we were interested in identifying failures in any autonomous system, we used the mapping function

described in Equation 3.8, replacing $B_{provided}$ with the sequences of steering angle data from each of the three autonomous systems, $\Theta = \{\theta_1, \theta_2, \theta_3\}$. The implementation of this equation is shown below in Equation 3.11.

$$mapping(o^{sen}, \Theta) : \Delta(max(\Theta), min(\Theta), bt) \geq ft \tag{3.11}$$

Here, the mapping function computed the extreme behaviors over Θ, and then compared them to the behavior threshold bt. Finally, in our approach, we described the failure frequency threshold ft as a threshold determining how many actions in a behavior need to be violated before the oracle outputs a failure. In our study we were interested in stricter sequential sequences of failures, as opposed to just failure frequency. Therefore we modified our implementation to only look for sequential failures. Specifically, it would only output a failure if a **sequential** sequence of actions in a behavior violated the behavior threshold bt for a duration meeting the failure frequency threshold ft. The values for both bt and ft were varied in the study.

To ensure that the oracle aired on the conservative side when flagging failures, we implemented a preprocessing phase that clipped each of the steering angles in Θ. Specifically, we clipped each of the steering angles in $\theta \in \Theta$ to ± 90 degrees. This effectively removed any failures that were solely due to the magnitude of the steering angle. For instance, without this preprocessing, two steering angles indicating a sharp right turn might be flagged as inconsistent simply because of a difference in magnitude, even though both were logically performing the same action. While some developers might want to detect these differences, our goal was not to flag differences based solely on the magnitude. Rather, we aimed to identify failures that were clearly impactful, as they were caused by differences in logical actions. By limiting $\theta \in \Theta$ to ± 90 degrees, we essentially restrict the range of behaviors, making it more likely to identify failures that are caused by significant differences in logical behaviors. For example, one AS turning left $+45$ degrees and the other turning right -45 degrees represent a significant difference of 90 degrees. This contrasts with both angles turning in the same direction, such as one at 180 degrees and another at 270 degrees, which also differ by 90 degrees but are less likely to reflect impactful differences in driving logic.

3.2.2.1.5 ODD Converter

The initial phase of this task required manually converting the ODD of comma.ai's openpilot ALC system from its natural language description, presented as a bullet list on their website [70], into a structured format. Each of the points from the website are presented in the first column of Table 3.3. The researchers identified various semantic dimensions from each of these bullet points. For instance, a bullet point mentioning "When in sharp curves, like on-off ramps, intersections, etc, ..." was interpreted to encompass three distinct semantic dimensions: sharp curves, on-off ramps, and intersections, all of which are detailed in the second column of Table 3.3. Subsequently, each of these 11 semantic dimensions was translated into a series of binary "yes" or "no" questions, maintaining as much of the original ODD wording as possible. These questions are displayed in the third column of Table 3.3 and were used as part of the prompts for ChatGPT-4V(ision) [270] and Vicuna [59] in the subsequent ODD Checker phase.

3.2.2.1.6 ODD Checker

In the development of the ODD checker, we explored two readily available LLMs. The first utilizes the MiniGPT-4 framework, which integrates the open-source Vicuna V0, a 13 billion parameter model [59], through a projection layer. This integration allows for the processing of visual inputs. The second LLM we assessed was OpenAI's proprietary ChatGPT-4V(ision) [270], which we accessed via their API. This model offers potentially more advanced, industry-grade capabilities but incurs a cost of approximately $0.02 per image-prompt pair.

3.2.2.1.7 Prompt Fine-tuning

Prompt fine-tuning has been shown to potentially enhance the performance of LLMs, leading to the generation of a second set of prompts. Utilizing 10 recognized prompting strategies [375] and ChatGPT-4, we developed 10 alternative question sets. By combining these strategies and questions, we created 100 unique prompts, which were then evaluated against 150 sample images whose ground truth compliance vectors had been determined by three independent researchers. The eval-

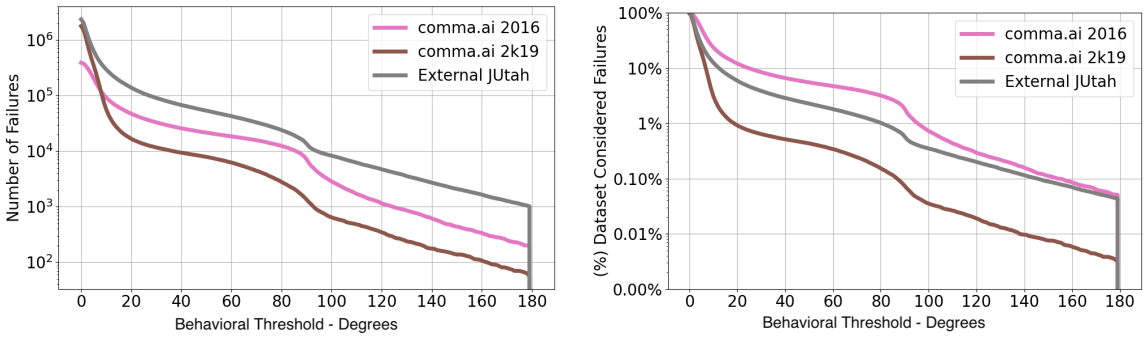
Table 3.3: The natural language ODD, the identified semantic dimensions, and the converted questions

Natural Language ODD	**Semantic Dimension**	**Converted Question**
Poor visibility (heavy rain, snow, fog, etc.) or weather conditions that may interfere with sensor operation.	Poor Visibility	Does this image have poor visibility (heavy rain, snow, fog, etc.) or weather conditions that may interfere with sensor operation? - [YES/NO]
The road facing camera is obstructed, covered or damaged by mud, ice, snow, etc.	Image Obstructed	Was the camera that took this image obstructed including by excessive paint or adhesive products (such as wraps, stickers, rubber coating, etc.), covered or damaged by mud, ice, snow, etc? - [YES/NO]
Obstruction caused by applying excessive paint or adhesive products (such as wraps, stickers, rubber coating, etc.) onto the vehicle.		
The device is mounted incorrectly.	NA	NA
When in sharp curves, like on-off ramps, intersections etc; openpilot is designed to be limited in the amount of steering torque it can produce.	Sharp Curve	Is the road we are driving on a sharp curve? - [YES/NO]
	On-Off Ramp	Is the road we are driving on an on-off ramp? - [YES/NO]
	Intersection	Is the road we are driving on an intersection? - [YES/NO]
In the presence of restricted lanes or construction zones.	Restricted Lane	Does the road in this image have restricted lanes? - [YES/NO]
	Construction	Does the road in this image have construction zones? - [YES/NO]
When driving on highly banked roads or in presence of strong cross-wind.	Banked Road	Is the road we are driving on highly banked?- [YES/NO]
Extremely hot or cold temperatures.	NA	NA
Bright light (due to oncoming headlights, direct sunlight, etc.).	Bright Light	Does this image have bright light (due to oncoming headlights, direct sunlight, etc.)? - [YES/NO]
Driving on hills, narrow, or winding roads.	Narrow Road	Is the road we are driving on narrow or winding? - [YES/NO]
	Hilly Road	Is the road we are driving on a hill? - [YES/NO]

uation aimed to find the prompt that resulted in LLM responses most closely aligning with human annotations, as determined by the F1-score [365]. This refined prompting method, named Vicuna+, was not applicable to ChatGPT-4V due to its daily usage constraints [271], which at the time limited the number of requests to 100 images prompt pairs a day. Although prompt fine-tuning can further harness the potential of LLMs, it necessitates extra LLM queries and the availability of human-annotated dataset segments to ascertain the most effective prompt.

3.2.2.2 RQ1: Identifying failure effectiveness

To evaluate the efficacy of our differential testing component in identifying failures within real-world sensor data, we processed three datasets using this component. We began by analysing o^{sen} with



(a) Number of failures found.

(b) Percentage of failures found.

Figure 3.16: Single frame input which produce varying steering differences.

$ft = 1$. In this scenario we varied bt from 0 to 180 degrees. This is the equivalent of looking for single frame failures, whose difference in steering were between 0 and 180 depending on bt. The results are presented in Figure 3.16 as raw counts in Figure 3.16a and as a percentage of the total dataset in Figure 3.16b.

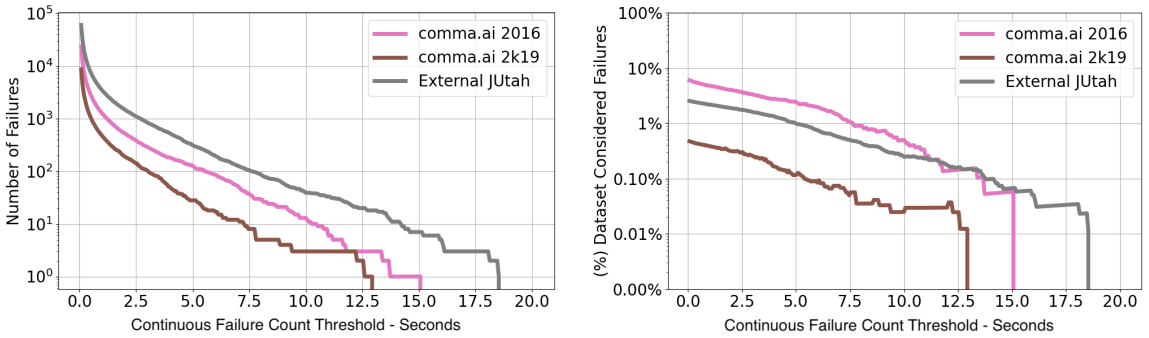
In these figures, a behavioral threshold of bt degrees represents the number of sensor readings in O^{sen}, which resulted in behaviors whose maximum and minimum steering angle difference was greater than bt degrees. When $bt = 0$ degrees, our approach, as expected, classified all 4.6 million sensor readings from the datasets as input which results in failures. Increasing the threshold, results in reduction in the number of identified failures. For example, at a $bt = 10$ degree steering difference, we detected 296,513 failures in the External Jutah dataset, 92,332 in the comma.ai 2016 dataset, and 56,750 in the comma.ai 2k19 dataset, amounting to 445,595 sensor readings or $\frac{445595}{4.6\times10^6} = 9.8\%$ of all the data. This pattern of diminishing failure rates continued with higher steering angle thresholds. However, notably, a significant number of behavioral differences were still discovered along this distribution's tail. This means there was a significant number of sensor readings in O^{sen} which produced large behavioral differences. Specifically, at a 100 degree steering difference ($bt = 100$), a threshold indicative of severe failures, our analysis revealed 8,279 inputs that result in such a behavioral differences in the External Jutah dataset, 2,846 in the comma.ai 2016 dataset, and 644 in the comma.ai 2k19 dataset. These findings not only represent a substantial number of potential

interesting inputs which can be used as test cases but also underscore the utility of leveraging large volumes of existing real-world data for test generation. If comma.ai were to only use their dataset for testing, they would identify 3,490 test cases which produce steering differences greater than 100 degrees in 3 versions of their autonomous system. Incorporating other datasets, for example the publicly available sensor data from the Jutah dataset, increases this number to 11,769 failures, a $\frac{11769-3490}{3490} = 237\%$ enhancement in real-world test cases for comma.ai.

When examining the percentage of each dataset causing a failure, as shown in Figure 3.16b, we find that failure rates in External Jutah are intermediate between those of the comma.ai datasets. This suggests that although the most failures were found in External Jutah, the comma.ai 2016 dataset still contains a relatively higher number of identifiable failures. This indicates a greater need for data exploration in External Jutah compared to comma.ai 2016 to uncover these failures. Nonetheless, since this process is less costly than creating new test cases from scratch and the data is readily available, it proves to be an effective method for testing autonomous vehicles.

Next, we investigate the effects of varying the length of ft. Remember, in our implementation, we have a stricter definition of ft: the behavior must contain more failures than ft and must also be sequential. Intuitively, this can be thought of as looking for failures of varying duration. As such, we present these findings in seconds rather than the number of frames, using a simple conversion based on the frame rate of openpilot, which is 15 frames per second. Using this, we can convert a length of $ft = 1$ to $\frac{1}{15} = 0.066$ seconds. By examining failures over longer periods, we aim to enhance the relevance and practicality of our findings, identifying test cases with a longer impact on vehicle behavior. Figure 3.17a presents these findings, emphasizing test cases of varying lengths with a difference threshold of $bt = 45$ degrees. This threshold was chosen for its significance in marking a clear distinction between maintaining a straight path and initiating a turn.

We find that the already existing, real-world data in External Jutah not only captures the highest number of failures but also produces failures that last for the longest period of time. For example, developers seeking to find test cases that would result in the autonomous vehicles producing a steering error over a span of 10 seconds ($ft = 150$) would find limited instances within their own datasets: 13 in comma.ai 2016 and 3 in comma.ai 2k19. However, leveraging the publicly available



(a) Number of failures found.

(b) Percentage of failures found.

Figure 3.17: Varying duration of input which produced a continuous steering difference of 45 degrees.

External Jutah dataset reveals a substantially higher count of 107 test cases, illustrating an increase in potential test scenarios of $\frac{107-16}{16} = 569\%$.

Furthermore, Figure 3.17a highlights that the External Jutah dataset not only finds the largest number of test cases but also the longest test cases. Within the comma.ai datasets, the longest recorded failure durations were 15.1 seconds for comma.ai 2016 and 12.9 seconds for comma.ai 2k19. In contrast, the Jutah dataset presents a failure lasting 18.6 seconds, marking a $\frac{18.6-15.1}{15.1} = 23\%$ increase over the longest duration found in their own datasets. This comparison not only highlights the ability of publicly available real-world data to enrich the pool of significant test cases but also demonstrates the differential testing components' capacity to uncover them.

RQ1 Finding: Our study demonstrates the effectiveness of the differential testing component in uncovering a significant number of potential failures within autonomous vehicle systems using real-world sensor data. Specifically, we find that 9.8% of all data tested produces a steering angle difference of 10 degrees across three versions of the same autonomous system. Given our dataset, this would provide developers with $445,595$ inputs to enhance testing and training. Furthermore, we find a long tail of inputs that result in significant behavioral differences. More specifically, we find $11,769$ inputs that result in behavioral differences greater than 100 degrees. We also find that these failures can occur continuously for long durations. Specifically, we find 123 failures that produce a steering behavior difference of greater than 45 degrees for over 10

seconds, or 150 sensed scene-state pairs. Finally, we show how this component allows testers to use external data to find significantly more failures. We observed a substantial increase in test case volume, with 237% more single-frame input which results in a 100 degree steering difference and 569% more input that produce continuous failures of at least 10 seconds. Additionally, our approach can identify longer-duration failures, up to 23% longer, using publicly available data, underscoring the value of both our approach and diverse already-existing data sources in enhancing autonomous vehicle safety and testing comprehensiveness.

3.2.2.3 RQ2: Filtering based on ODD effectiveness

Next, we examine the accuracy of the ODD filtering component in isolation. This study requires a dataset with ground truth compliance vectors for comparison. As none of the selected datasets, nor any known dataset, had corresponding ODD compliance vectors, we needed to create one from scratch. Additionally, since none of the datasets used included state information, we only considered sensed scenes c^{sen}, which took the form of camera images. To create the ground truth compliance vector image dataset, we needed to perform three steps. First, we required a method to fairly and accurately annotate the images with corresponding compliance vectors. Second, we needed to decide how many images to annotate, as annotating all 4.6 million images with ground truth compliance vectors was not feasible, regardless of the annotation method selected. Finally, following on from above, we would need to determine an appropriate sampling method to create the subset of data to be annotated.

To reduce bias and ensure accuracy, we decided to have three independent researchers, Trey Woodlief, Sebastian Elbaum, and Carl Hildebrandt, review each of the selected images and annotate them with a compliance vector. This vector consists of 11 semantic dimensions from openpilot's ODD, as detailed in Table 3.3. Following their individual assessments, the researchers convened to resolve any discrepancies in their evaluations, reaching consensus on images where compliance was challenging to ascertain.

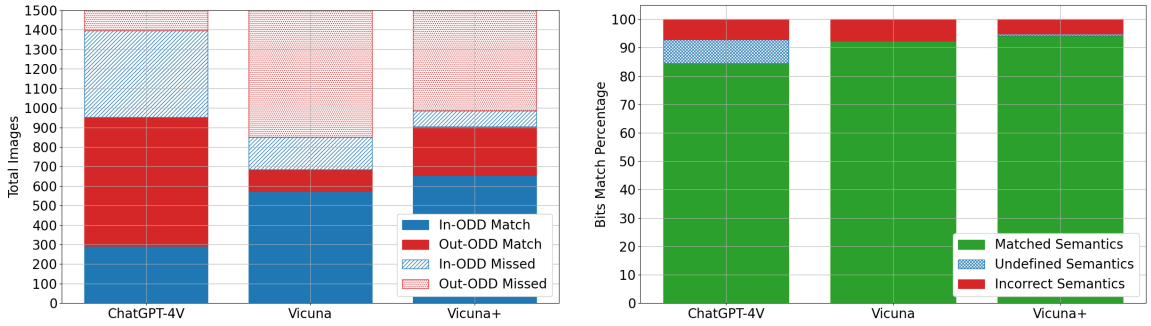
Our selected method for annotation attempted to reduce the chance of bias; however, it came at the cost of requiring three times the number of annotations, as each of the three reviewers had to

annotate the same image. This resulted in the researchers only being able to annotate 500 images from each of the datasets, leading to a total of 1500 images being annotated by each researcher. As each image required answers to 11 different questions, each researcher independently had to answer 16,500 Boolean questions. This serves to highlight that while 1500 images may not seem like a large number, significant effort went into creating this dataset, and it is the first of its kind with no other datasets annotated with ODD compliance vectors.

Finally, to select the 1,500 images, we created two *oracle*s. The first *oracle* looked for single-frame failures using $ft = 1$, with a behavioral threshold difference set to $bt = 45$. This threshold, similar to the second half of RQ1, was chosen for its significance in marking a clear distinction between maintaining a straight path and initiating a turn. Using this oracle, we created a subset of $92,139$ images, or roughly 2% of all the data, which we knew to be failing-inducing sensor data. To mitigate any bias stemming from an imbalanced ground truth dataset, potentially skewed by an over-representation of failures, which may disproportionately fall outside the ODD, we also identified a set of passing images. To accomplish this, we created the second *oracle* to look for single-frame failures using $ft = 1$, with a behavioral threshold difference set to $bt = 1$. Then, instead of using it to find failure-inducing input, we simply examined the inverse, which identified input that passed or had a behavioral difference of less than 1. This produced a subset of passing sensor data with a total of $599,230$ images, or roughly 13% of all the sensor data, whose steering difference was less than 1 degree. From both of these sets, we randomly selected 250 passing and 250 failing images, equally distributing random choices across all videos, to create three subsets of 500 images.

Using this subset of data, we then analyzed the ODD-filtering accuracy using different LLM Checkers across several dimensions with two metrics. The first metric, in-ODD accuracy, measures whether the LLM Checker correctly identified the in/out of ODD status per image, with results shown in Figure 3.18a. This evaluates the true and false positive rates for in/out-ODD labeling. The second metric, semantic accuracy, assesses accuracy over the compliance vector's semantic dimensions, illustrating the LLM Checker's overall accuracy; results are presented in Figure 3.18b.

In Figure 3.18a, "In-ODD Match" indicates a true positive in-ODD, and "Out-ODD Missed" indicates a false positive in-ODD, i.e., the LLM Checker mislabeled an out-of-ODD image as in-



(a) ODD-Filtering in-ODD Accuracy

(b) ODD-Filtering Semantic Accuracy

Figure 3.18: ODD filtering accuracy per LLM Checker

ODD. ChatGPT-4V appears conservative in its labeling of items as in-ODD, achieving by far the lowest false positive in-ODD count and rate, while Vicuna and Vicuna+ are much less conservative, labeling many images as in-ODD.

These results can be thought of in terms of in-ODD precision, i.e looking at the in-ODD true positive to false positive ratio. We find that ChatGPT-4V has a high in-ODD precision of $\frac{294}{(294+104)} = 73.9\%$, meaning that in 73.9% of cases where ChatGPT-4V says an image is in-ODD, it is correct. By comparison, Vicuna has a precision of $\frac{573}{(573+650)} = 46.9\%$, and Vicuna+ has a precision of $\frac{656}{(656+514)} = 56.1\%$. Despite ChatGPT-4V achieving the highest precision, Figure 3.18b shows that Vicuna and Vicuna+ achieve slightly higher aggregate semantic accuracy with 84.4%, 92.6%, and 94.2% respectively. This performance difference is largely due to ChatGPT-4V's high number of "Undefined" answers, due to cases where the model would say it was unsure; prompt refinements to encourage the model to take a stance may render fur-

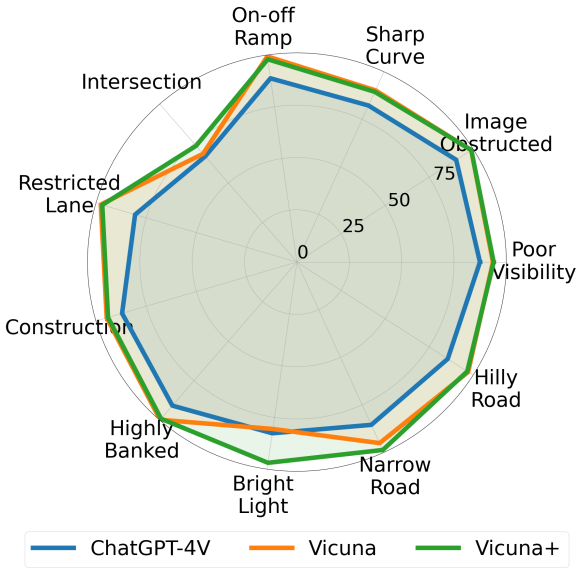


Figure 3.19: ODD-Filtering Accuracy per Semantic Dimension

ther improvements. Overall, the performance of ODD-filtering rendered encouraging results at the semantic level.

Figure 3.19 further drills into this accuracy, showing the performance of each LLM Checker per semantic dimension. Across all configurations, ODD-filtering achieves an average accuracy of 90.3%, with a maximum of 99.7% accuracy for Vicuna on the "Highly Banked" dimension. From this, it appears that Vicuna and Vicuna+ are more accurate than ChatGPT-4V; this is again largely due to the high number of "Undefined" answers, which are marked as inaccuracies here. We also note that prompt fine-tuning improved Vicuna's accuracy, moving from an average of 92.3% to 94.2%, with particularly strong gains in the "Bright Light" and, to a lesser degree, "Intersection" dimensions. This may point to possible future improvements for the LLM Checkers in the ODD-filtering component.

RQ2 Findings: ODD-filtering achieves high aggregate semantic accuracy of up to 94.2%. Further analysis shows accuracy is high across all semantic dimensions, achieving a maximum accuracy of 99.7%. Additionally, ODD-filtering demonstrates high precision for determining if an input is in-ODD, with a maximum precision of 73.9%.

3.2.2.4 RQ3: Full framework effectiveness

To answer the final research question about the effectiveness of the proposed approach, we examine the pipeline from both a quantitative and qualitative viewpoint. Firstly, we consider it from a qualitative perspective. We showcase some of the failure-inducing sensor images that produced steering angle differences greater than 45 degrees for one frame, across each of the datasets, sampled from a set of 1500 images. These examples demonstrate where both the differential testing identified a failure, and the ODD checker correctly highlighted that they were in-ODD. Figure 3.20 showcases 9 of the sampled images, with the horizontal axis representing each of the 3 different datasets, and the vertical axis representing each of the LLMs used in the ODD Checker. Each image also has the steering angle of the three different openpilot versions overlaid onto the image to aid in post-analysis.

While each of the images shows concerning failures, there are a few that we want to highlight as particularly troubling. For example, consider the image highlighted by Vicuna+ in the comma.ai

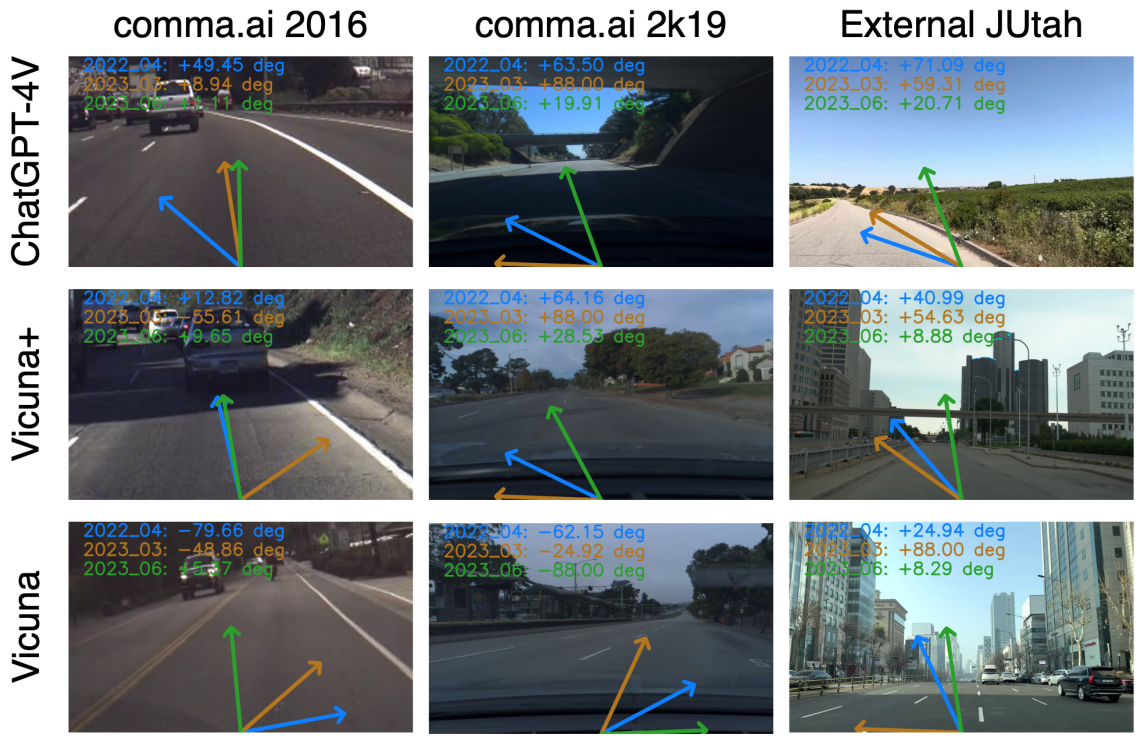
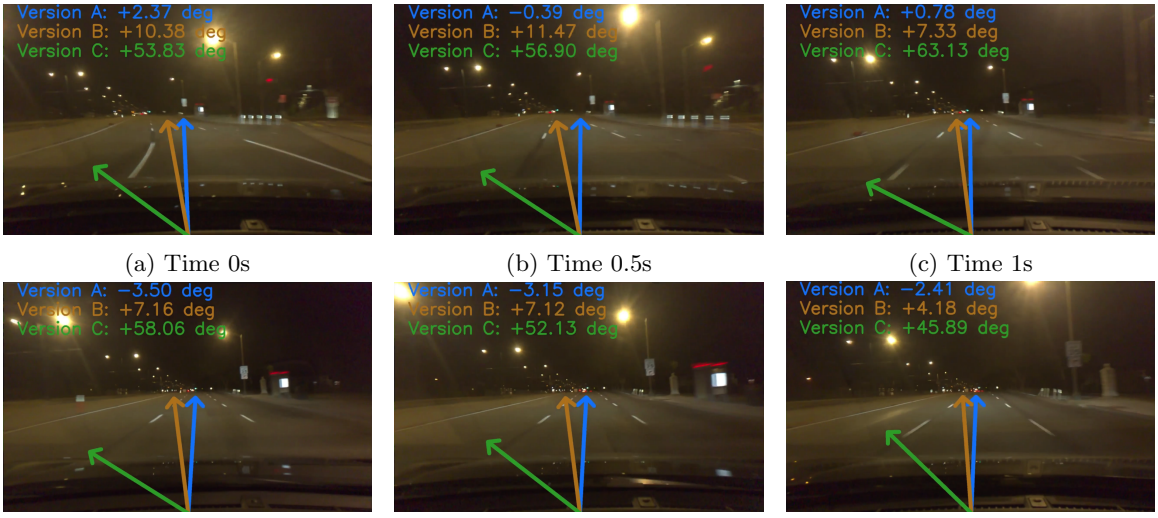


Figure 3.20: Camera images that produced a steering difference of more than 45 degrees, for at least 1 frame, that were in-ODD, identified by our approach.

2016 dataset (left middle row), where a car navigates a road with a slight curve to the left. Here, while the oldest and latest versions of openpilot are following the road, the intermediate version makes a hard right turn off the road, clearly heading into the embankment. A similar scenario is observed in the image found by Vicuna in the External Utah dataset (right bottom row). The car is navigating a straight multi-lane road, and the intermediate openpilot version is making a hard left turn off the road, potentially entering another lane or, in the worst case, colliding with the road railings. Another interesting finding is that the intermediate version is not always the culprit. In the image identified by Vicuna in the comma.ai 2016 dataset (left bottom row), the older version appears to be at fault, while in the image found by Vicuna in the comma.ai 2019 dataset (center bottom row), it appears to be the latest version. This highlights that failures can occur with each version, in each dataset, and with each LLM tested.



(a) Time 0s (b) Time 0.5s (c) Time 1s

(d) Time 1.5s (e) Time 2s (f) Time 2.5s

Figure 3.21: A sequence of camera images that produced a steering difference of more than 45 degrees, for at least 38 frames (approximately 2.5 seconds), that were in-ODD, identified by our approach.

Each of the failure-inducing inputs in Figure 3.20 was found when the *oracle*'s was looking for single frame failure $ft = 1$, i.e. only one frame needed to exhibit a steering difference of 45 degrees or more. While we argue that these are significant, as this is a safety-critical system, there is room to argue that this might not cause an issue, as there is ample time for the autonomous system to correct itself. To account for this, and to determine if such failures only occur briefly, we examined all failures that lasted 2.5 seconds, a duration during which, at speeds typical on traditional roads, a steering difference of 45 degrees could cause significant behavioral changes. Figure 3.21 showcases six frames, each 0.5 seconds apart, for one such case found within ODD. In this sequence, the car is seen driving down a straight, well-lit road at night, in the center lane. Interestingly, throughout this sequence, the latest version of openpilot attempts to make a sharp left turn out of the center lane, while the previous two versions continue to drive straight to varying degrees. There is a much higher chance that this sequence of images would result in significant consequences, potentially even a crash, due to the severity and total duration of this difference.

Next, we examined the pipeline quantitatively, focusing specifically on its efficiency in automating

the process. Consider a developer tasked with generating test cases. Our approach is the first to automate this process by selecting failure-inducing images that are most likely in-ODD for the developer to review. Our approach provides the developer with a list of images judged to be in-ODD, which the developer then reviews to confirm. Therefore, to determine efficiency, we compare the percentage of true in-ODD inputs found versus the percentage of inputs the developer reviewed.

Figure 3.22 illustrates the efficiency gains achieved by automating the review process. The red dashed line represents the human annotation approach baseline, where each input image is manually reviewed. As the human examines a greater portion of the input set, it is assumed that they would discover a proportionate amount of failure-inducing inputs that are within the ODD, reaching 100% when all images are inspected. Techniques that yield scores above this line are more efficient as they enable humans to identify more in-ODD failure-inducing inputs in the same amount of time. Techniques below this line are less efficient, requiring humans to spend more time analyzing failure-inducing inputs to find the same number within ODD.

We observe several interesting patterns. First, ChatGPT-4V, across all datasets, is more efficient than the human annotation approach. At its peak on the comma.ai 2016 dataset, ChatGPT-4V correctly identifies 24.7% of all in-ODD failure-inducing inputs while requiring the human to review only 10.0% of images: $\frac{24.7\%}{10.0\%} = 2.47\times$ or 147% improvement in efficiency. By contrast, Vicuna consistently labels almost all inputs as in-ODD, leading to efficiency on par or slightly below the baseline. However, Vicuna+ is able to improve upon this performance, showing efficiency over

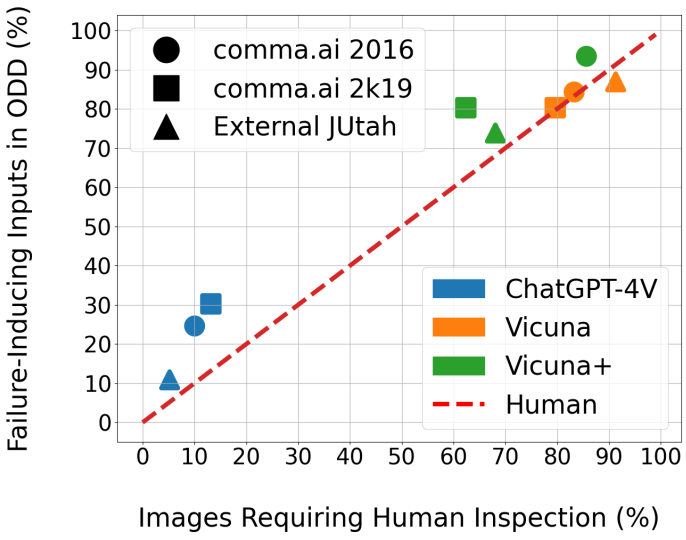


Figure 3.22: The efficiency improvements of our approach compared to a human baseline.

the baseline on all datasets. Vicuna+ achieves peak efficiency on comma.ai 2019 by finding 80.3% of failure-inducing inputs while only requiring the developer to review 62.4% of the dataset—an improvement of $\frac{80.3}{62.4} = 28.7\%$. While a lesser gain in efficiency compared with ChatGPT-4V, Vicuna+ is able to identify a much larger quantity of in-ODD inputs.

The out-of-the-box success of the commercial ChatGPT-4V and the improvements shown by simple prompt fine-tuning on open-source models demonstrate the potential for our approach to provide utility in automating ODD compliance checking. Furthermore, while our approach is the first capable of identifying such failure-inducing inputs at a rate more efficient than humans, it is quite conservative by design. In order for a failure-inducing input to be marked in-ODD, every semantic dimension must be compliant; this constraint could be relaxed to consider only partial ODD compliance. Additionally, we note that the LLMs currently used have had no additional training for this problem domain, and we assume that further improvements could be made through these methods.

RQ3 Finding: Our approach is capable of identifying safety-critical failures, which can occur for up to 2.5 seconds in length. It is also automatically capable of identifying when failures are in-ODD, achieving up to 147% improvement in efficiency when compared to purely manual analysis. This could, for example, enable a developer to find 24.7% of failures while only analyzing 10.0% of the dataset.

3.2.3 Summary

This work highlights the inconsistencies in one of today's commercial autonomous cars. It demonstrates how our approach enables us to find a large number of potential failures. It also shows the potential to identify failure-inducing sensor input within existing datasets, drastically reducing the cost of test generation. Furthermore, we illustrate the untapped potential of compliant data that remains unused or wasted due to the lack of automated compliance checks. We then showcase our entire approach and find several compelling cases. These cases come from a variety of scenarios and span a range of durations. Overall, we show how our approach is capable of identifying inputs that could potentially lead to fatal accidents without the need for expensive data collection or precise

definitions of what the output of any given input should be.

3.3 Conclusion

This chapter highlights two approaches for generating tests, specifically focusing on a test's *input*s and a test's *oracle*. The first approach looks at how to generate *input*s using insights from an autonomous vehicle's *physical semantics*. Specifically it presents a way to construct test *input*s that are both physically feasible and stressful. We found that incorporating physical semantics, through kinematic and dynamic models, although computationally more expensive, is crucial for efficiently generating valid tests. Furthermore, our approach was able to generate tests that were, on average, between 41.3% and 55.9% more stressful compared to tests constructed without our approach. Additionally, we demonstrated this approach using a commercially available quadrotor, which successfully produced tests that resulted in deviations from the anticipated flight trajectory by up to 6.2 meters—nearly 20% of the testing environment's total length. This underscores the approach's capability to maximize stress even in commercial systems.

The second part of this chapter argues that a vast amount of *input*s, in the form of sensor data from the real-world *physical environment*, already exists. Therefore, this work focuses on how to identify *input*s that result in inconsistent and potentially incorrect behaviors in an arbitrary autonomous system. Specifically, we describe how this can be achieved using an *oracle* capable of efficiently identifying behavioral differences across multiple autonomous systems given the same input. Our approach also introduces a novel technique for filtering arbitrary sensor data with respect to an autonomous system's ODD. We then present a study showcasing our approach on three real-world commercial autonomous vehicles. We highlight our approach's ability to detect a large number of inputs in already existing sensor data that result in inconsistent behaviors in three autonomous systems. Specifically, we find that 445,595 inputs, or 9.8% of the data tested, produce steering differences of more than 10 degrees in the three commercial systems. We further highlight, 11,769 instances where the three systems differ by more than 100 degrees and showcase how existing external sensor data can be used to significantly increase the number of failures detected in these

systems, with our study showing an increase in failures by up to 569%. The study then examines our novel filtering technique, demonstrating the approach's ability to correctly identify semantic dimensions of arbitrary sensor input with accuracies of up to 94.2%. To achieve this, we also created the first dataset annotated with ground truth ODD semantic compliance data. Finally, we conclude this work by showcasing several examples of cases in which the autonomous system failed, both on diverse single-instance sensor data and on continuous input sequences lasting up to 2.5 seconds.

Generating *input*s, however, is not enough. Tests need to be executed or have already been executed in either simulation or the real world in order to judge their outcome. In the next chapter, we will explore ways in which these tests can be executed more efficiently with respect to their physical environments and physical semantics.

Chapter 4

Test Execution

In the previous chapters, we discussed the first step of the testing pipeline, test generation, and focused on the first two components of a test: the *input* and *oracle*. In this chapter, we discuss the second step in the testing pipeline, test execution. Additionally, we introduce the final component of a test, which has not yet been discussed: the test's *context*. The *context* is essential as it defines the backdrop against which all tests are executed. Specifically, it refers to how closely the scenario w and the autonomous system's AS state s are mocked. Two *common* forms of context, are simulation and the real world. If it is in simulation, what type of simulation is used? A simple point simulator or a highly complex simulator capable of replicating both the scenarios and physics of the real world? If it is in the real world, is it conducted in a controlled lab environment or in the uncontrolled real world?

To understand how to answer these questions, let's first examine what constitutes an effective test execution strategy. An effective test execution strategy should strive for a balance between cost and realism. By "cost," we mean that the testing environment should aim to minimize the expenses associated with executing tests, thereby enabling a larger volume of tests to be conducted. The lower the cost, the more extensive the testing that can be performed, and a broader range of system behaviors can be observed, providing developers with increased data points. These data points do not conclusively confirm that the system operates as expected; rather, they enhance the likelihood

of detecting any potential failures. By "realism," we mean that the testing scenario should aim to replicate the physical environment and the physical semantics of the system's future deployment scenarios and the system itself as closely as possible. This ensures that the results of the testing process closely match what would be observed during actual operation.

Using these metrics, we can now answer the above questions about simulation versus real world execution. Simulation provides a cost-effective way to execute tests, allowing developers to run many tests quickly. However, simulation can never fully replicate the real physical environment or its physical semantics. This discrepancy between simulation and reality is known as the simulation-reality gap [163]. This gap means that simulation only provides an approximation of the physical environment and physical semantics for the autonomous system. So while developers can use simulation to explore a wide range of possible scenarios cheaply, the resulting behaviors may not fully represent what will be observed when the system operates in the real world, potentially overlooking critical failures.

On the other hand, real-world execution is generally expensive and limited in the scope of possible tests. Running experiments in controlled conditions requires personnel, safety measures, physical hardware, a testing space, and substantial preparation. Furthermore, fully exploring both the physical environment and the system's physical semantics is not always possible. Consider the substantial effort required to determine if an autonomous system can safely stop for a pedestrian running into the street. Testers would need to vary the physical environment so that the pedestrian appears from behind several different obstacles, on various road types, under many different lighting conditions, and with diverse body types. They would also need to alter the physical semantics by using different tires, road conditions, and varying speeds and system states. This is before even considering how to safely and realistically position a pedestrian in front of a moving vehicle. In this case, the number of tests that can be run is limited due to the costly and hazardous nature of such testing. In contrast, conducting experiments in uncontrolled real-world settings, such as in a city, might seem less expensive initially due to the absence of setup costs and could potentially offer a wider range of scenarios. However, the potential cost of failure is exponentially higher due to the risks posed to innocent bystanders. Moreover, this approach shifts the goal from a systematic exploration of

physical environments and semantics to one that follows the probabilities of the real world. Consider wanting to determine how an autonomous package delivery drone handles different packages under varying wind conditions to validate the vehicle's physical semantics. A package delivery company would need to start delivering packages, merely hoping that the wind conditions and the packages of that day match those intended for testing. In this case, exploring all corner cases, or even identifying which corner cases to anticipate, become much harder to do.

In this chapter, we examine the shortcomings, and provide alternatives to both execution strategies. First, in Section 4.1, we explore ways to reduce the simulation-reality gap and how it manifests in an autonomous systems physical environment. Specifically, we introduce a mixed-reality test environment that is constructed by combining real physical environments with simulation. This enables the mixed-reality testing environment to maintain the cost-effectiveness of simulation, while providing the ability to vary the realism of the test.

The second approach, presented in Section 4.2, focuses on the physical semantics and the complexities of executing tests in real-world scenarios. In particular, this section focuses on reducing the cost of testing autonomous systems, such as drones, under various real-world conditions. Our approach involves developing a haptic suit designed to replicate and apply various external forces to an autonomous system in a cost-effective way. This enables us to observe a wide range of potential behaviors based on the system's physical semantics in different real-world conditions.

4.1 World-In-the-Loop Simulation

A fundamental limitation for simulations today is the inability to create perfect replicas of the real *physical environment* or *physical semantics* of an autonomous system. This discrepancy leads to what is known as the simulation-reality gap. The discrepancy, due to the *physical environment*, results in differences in sensed scene readings of an AS operating in the same scenario in simulation and reality; it can be defined as $gap_c = diff(c_r^{sen}, c_v^{sen})$. Similarly, the discrepancy due to the *physical semantics* results in differences in sensed state readings in that same scenario in simulation and reality; it can be defined as $gap_s = diff(s_r^{sen}, s_v^{sen})$. The simulation-reality gap is a result of

both of these discrepancies, defined as $gap = gap_c + gap_s$. The objective of this work is to reduce this gap and facilitate a more seamless transition for developers from simulated to real-world scenarios.

To achieve this, we introduce the concept of World-In-the-Loop simulation. This approach reduces the simulation-reality gap by running two autonomous systems simultaneously: one in the simulation and one in the real world. It starts by overwriting the sensed state of the simulated autonomous system, s_v^{sen}, with the real-world's sensed state, s_r^{sen}. This effectively eliminates gap_s, the gap introduced by the *physical semantics* Thus, for the remainder of this chapter, we will primarily focus on this approach's ability to tackle gap_c, the gap introduced by the *physical environment*. World-In-the-Loop simulation tackles the issues introduced by gap_c, by merging sensed scene data from both the virtual c_v^{sen} and the real-world c_r^{sen} to form a mixed-reality scene c_m^{sen}. Since c_m^{sen} is created from a combination of both c_v^{sen} and c_r^{sen}, our approach allows us to vary the proportion of simulated to real-world data used in the sensed scene. This in effect allows the approach to vary gap_c, the gap introduced by the *physical environment*. Once both the state is overwritten, and the mixed reality sense scene is created, both are fed back into the autonomous systems, allowing the *AS*'s to execute effectively.

(a) Simulation (c_v^{sen}, s_v^{sen}).

(b) Real-world (c_r^{sen}, s_r^{sen}).

(c) World-In-the-Loop (c_m^{sen}, s_r^{sen}).

Figure 4.1: The camera image sensed from a grounded drone operating in simulation (a), the real-world (b), and the mixed-reality created by our approach World-In-the-Loop by integrating both (c).

An illustrative example of World-In-the-Loop simulation is shown in Figure 4.1. Here, the gate sensed by the autonomous system during the simulation (Figure 4.1a) and the autonomous system sensed reality (Figure 4.1b), are integrated to form a sensed mixed-reality (Figure 4.1c).

The mixed-reality created through World-In-the-Loop simulation aligns more closely with the real-world environment than a purely simulated one. This alignment is demonstrated in two ways. First, by replacing the simulated state with the real one, we completely remove any discrepancies due to the simulated state. This is evidenced by the fact that both the simulated and real cameras assume the same pose in the scenario. Second, by comparing the mixed-reality sensed scene data with that of the simulation, for example, by comparing the textures, shapes, and lighting, we can see that the mixed-reality scene more closely resembles the real world.

Another benefit of World-In-the-Loop simulation is that it reduces the cost during execution. For instance, it eliminates the need for a physical gate during testing, thereby reducing setup and execution costs. Furthermore, in scenarios where a drone collides with the virtual gate in the mixed-reality, there is no actual damage to the drone, thus reducing the risks and costs associated with failure.

4.1.1 Approach

Given a scenario w and a goal g, the autonomous system builds an understanding of its current scene in the scenario $c \in w$ and state $s \in \mathcal{S}$ through sensors sen, and acts on the sensor data (c^{sen}, s^{sen}) through a sequence of actions a to create a behavior $b = \langle a_0, a_1, a_2, \dots \rangle$. Due to the simulation-reality gap, defined as $gap = gap_c + gap_s = diff(c_r^{sen}, c_v^{sen}) + diff(s_r^{sen}, s_v^{sen})$, for some input, tests executed in simulation can result in the generation of simulated behaviors $b_v \notin \mathcal{B}$, which may not belong to the set of all possible behaviors found in the real world.

The goal of World-In-the-Loop is to narrow the simulation-reality gap. It achieves this by creating a new mixed reality that combines elements from c_r^{sen} and c_v^{sen} to generate c_m^{sen}. By creating c_m^{sen}, the approach is essentially capable of varying gap_c by adjusting the proportions of real and simulated c^{sen} used. Additionally, it replaces s_v^{sen} with s_r^{sen}, completely overcoming gap_s. Thus, by varying gap_c and removing gap_s, our approach should produce behaviors such that $diff(b_r, b_m) < (b_r, b_v)$.

4.1.1.1 Overview

An overview of World-In-the-Loop simulation is depicted in Figure 4.2. Given goal g, the approach simultaneously runs two instances of the autonomous system, one in simulation w_v and one in the real-world w_r. To ensure that both autonomous systems' are synchronized, the real-world sensed state s_r^{sen} from AS_r overwrites the virtual sensed state s_v^{sen} in AS_v. After each synchronization step, sensor readings c_r^{sen} from the real world and c_v^{sen} from the simulation are collected by sensor callback functions.

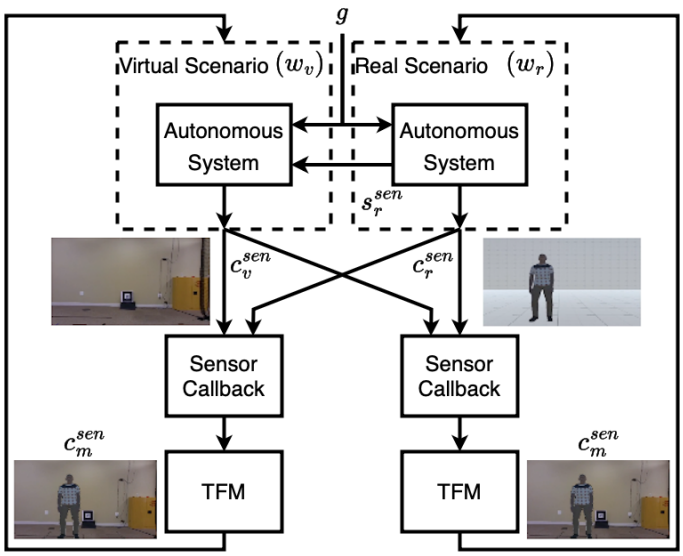


Figure 4.2: An overview of our approach.

These readings are combined using a Transform, Filter, and Merge (TFM) module to generate a mixed sensor value set c_m^{sen}, which is then fed back to both instances of the autonomous systems.

Algorithm 4 summarizes this process. The inputs include both a simulated autonomous system AS_v and a real autonomous system AS_r, a simulated scenario w_v and a real scenario w_r, a goal g, and a *recipe* file that describes how the mixing of c^{sen} is done. First, in line 2, World-In-the-Loop simulation subscribes to all sensor data from both autonomous sys-

Algorithm 4: World-In-the-Loop Overview

1 **Given** AS, AS_v, w_r, w_v, g, *recipe*
2 subscribe(AS, AS_v)
3 start(AS_v, w_v, g)
4 start(AS, w_r, g)
5 **Function** *sensor_callback*(c_r^{sen}, c_v^{sen}, s_r^{sen})
6 c_t^{sen} = **transform**(c_r^{sen}, c_v^{sen}, *recipe*)
7 c_f^{sen} = **filter**(c_r^{sen}, c_v^{sen}, c_t^{sen}, *recipe*)
8 c_m^{sen} = **merge**(c_r^{sen}, c_v^{sen}, c_f^{sen}, *recipe*)
9 publish(c_m^{sen}, s_r^{sen})

tems. The subscriber directs all sensor readings to the *sensor_callback* function, as described in line 5. In lines 3 and 4, the simulated and real autonomous systems are given the goal. The *sensor_callback* function receives sensor data c_r^{sen} and s_r^{sen} from AS_r and c_v^{sen} from AS_v. It then calls a transform, a filter, and a merge function to produce c_m^{sen}, the mixed-reality sensor data. Finally, in line 9, c_m^{sen} and the real state s_r^{sen} are published and used as inputs to both AS_v and AS_r. We now describe each of the transform, filter, and merge functions used to generate c_m^{sen}, along with the definition of a *recipe* file, in more detail below.

4.1.1.2 Transforming Sensor Readings

The transformation function removes structural discrepancies between the sensor readings obtained in c_v^{sen} and c_r^{sen}. Our built-in support focuses on dimensions-units, shape, and frame-of-reference, which we have identified as common sources of dissonance among execution environments. In terms of units and dimensions, we found that it is common to describe the same quantity in different ways. For example, when working with GPS, some systems sensors will use the full GPS data while others return positions in terms of a local frame using X, Y, and Z. When referring to rotations, we encounter quaternions, radians, and Euler angles [272]. Even when the units are comparable, they might not use the same frame-of-reference [302]. For example, one might work using a North East Down (NED) frame and another in an East North Up (ENU) frame. In terms of shape dissonance, it is common to find simulation environments that use a different resolution to build on an existing dated component or use a lower resolution to improve performance. The transformation function enables us to overcome such discrepancies in sensor data.

For example, to produce Figure 4.1, the sensor data was transformed in multiple ways. First, the quadrotor's simulated position was transformed from a local frame to a global frame that used GPS coordinates. Second, the quadrotor orientation in simulation was transformed using an offset so that the quadrotors heading in the real-world matched those in simulation. Third, the simulated camera image needed to be reshaped to match the real-world camera's resolution. Finally, both cameras' sensor publishing rates needed to be matched, which we achieved by storing the latest sensor data

from both simulation and reality, allowing the merge function to access the latest data regardless of rate.

4.1.1.3 Filtering Sensor Readings

Filtering aims to retain the sensor readings or parts of readings that will be integrated by the merge function. It can include diverse filters, from dropping a range of values or noise from the LiDAR, to removing sets of colors from an image, similar to the techniques used when using a greenscreen. Other examples use DNN's that perform object detection and isolation [296], or background subtraction techniques to remove camera images' backgrounds. Other sensors, such as microphones, could have bandpass filters applied to isolate a range of frequencies, for example, those typical to human speech. Thus, filtering functions for the sensed values enables World-In-the-Loop to isolate parts of the sensor data required for merging while discarding data to reduce excess throughput and later speed up merging.

As an example, to produce Figure 4.1c, the camera data from the simulation was passed through a color isolation function that identified the color orange. This removed all parts of the image except for the gate which was orange.

4.1.1.4 Merging Sensor Readings

The merge function creates a mixed-reality sensor reading c_m^R. For example, given simulated and real-world camera data, a simulated obstacle can be overlayed over the real camera data, giving the AS the impression that an obstacle exists in the real-world (as per Figure 4.1c). The function starts by creating an empty mixed-reality sensor reading c_m^{sen} that is then populated with the mixed-reality readings by applying the corresponding combination function to each sensor. The sensor data can be combined using a number of general mechanisms, including: 1) sensor prioritization, where sensor data from c_v^{sen} and c_r^{sen} is layered according to some predefined priority, 2) sensor replacement, where sensor data is replaced according to some rule, for example, the source world, and 3) sensor aggregation, where the sensor data is combined by performing some operation like average, minimum, or maximum sensor reading.

Going back to the example in Figure 4.1, we note how some of the real-world camera pixels are replaced by the virtual gate pixels, allowing the *AS* to perceive both virtual and real obstacles.

4.1.1.5 Recipe Files

A *recipe* file offers a mechanism for users to define and link various transform, filter, and merge functions to each of the autonomous systems' sensors. There are several ways this could be instantiated; however, we provide a brief example of how one such instantiation might work. Consider Listing 4.1, which resembles a traditional XML file which includes distinct tags. The first tag, *AS sensors*, lists all the sensors of the autonomous system. Each sensor listed could then be assigned a unique ID. This ID would enable World-In-the-Loop to differentiate between two sensors of the same type, for example, two cameras. Another significant tag could be the *combine* tag, which links a sensor reading to a specific function using the unique ID from the *AS sensors* tag. The combine tag would specify which function should be applied at each stage of the approach for a given sensor. For example, Listing 4.1 shows that the 'filter_definition_1' function performs the filtering for camera1. This general approach would allow developers to implement and quickly switch between different functions, creating various mixed-reality scenarios for the autonomous system.

Listing 4.1: An example recipe file

```
<recipe file>
    <AS sensors>
        <Camera id="camera1"/>
        <Camera id="camera2"/>
        <LiDAR id="LiDAR1"/>
        <.../>
    </AS sensors>
    <combine id="camera1">
        <transform> transform_definition_1 </transform>
        <filter> filter_definition_1 </filter>
        <merge> merge_defintion_1</merge>
    </combine>
    ...
    <combine id="LiDAR1">
        ...
    </combine>
</recipe file>
```

4.1.1.6 Implementation

Our implementation provides support for the collection and distribution of sensor values, which is built on top of ROS publish and subscribe architecture [328]. To be integrated with existing simulators and systems, World-In-the-Loop only requires the completion of a plugin to set the pipelines to distribute AS sensor values through specific message types underlying the pub-sub model. By building on ROS we also leverage its standard message types that already support a wide range of sensors such as cameras, LiDARs, or IMU's [301]. Our approach also took advantage of ROS launch files to quickly activate or deactivate the appropriate subsystems to easily switch between simulation, mixed-reality, and reality.

The final piece of the implementation worth mentioning is the support for processing *recipe* files. Instead of creating an XML parsing and function linking system from scratch, we prototyped each function as a ROS node, setting each subscriber and publisher based on the sensors they were meant to process. These were then instantiated using standard ROS launch files. Each node was defined in Python files, making them easily replaceable or modifiable for various use cases. The implementation has been made publicly available [135].

4.1.1.7 Limitations

World-In-the-Loop makes a few assumptions that may limit its applicability and efficacy. First, it assumes that it is possible to match sensor specifications between simulation and reality. For example, a camera has a resolution, field of view, dynamic range, color depth, and readout speeds to name a few. In recent years, simulation technology has become more sophisticated and robust, making this a reasonable assumption for many scenarios. In our implementation, we used the Sphinx simulator [277] created by the drone developers, so the sensors were closely matched. Second, World-In-the-Loop assumes that the latency introduced through sensor data manipulation does not affect the test results. Our implementation mitigates this risk by using low-latency communication channels combined with optimized data manipulation libraries. For example, our implementation generates mixed-reality sensor readings at 17Hz. Although this is slower than the drone's 60Hz

camera [275], it is faster than the perception layer's control loop, which operates between 5-15Hz. Third, it assumes access to precise real-world autonomous system state information. Inaccurate state information could lead to a misalignment of the autonomous system in simulation and reality, resulting in mismatched sensor information. Our implementation addresses this risk through the use of Vicon [359], an industry-grade motion capture system with millimeter accuracy. The final limitation is that the implementation is a prototype closely designed to support the subsequent study. Its generalization for other systems would benefit from its reconstruction as a more generic API that allows, for example, determining when and where simulation and reality are mixed.

4.1.2 Study

The goal of the study is to assess our approach World-In-the-Loop's ability to reduce the simulation-reality gap. More specifically, we aim to answer the following two research questions:

RQ1) Does World-In-the-Loop reduce the simulation reality gap?

RQ2) Does World-In-the-Loop reduce the cost of execution and failures in the real-world?

4.1.2.1 Setup

For the evaluation, we are using the Parrot Anafi quadrotor [275], which weighs $0.5kg$, has a width of $0.3m$, and is equipped with a stabilized front-facing camera.

We use two simulation platforms. The first one is Sphinx [277], a simulator developed and maintained by Parrot and built upon Gazebo [111]. Since it was developed by Parrot's engineers, it serves as a good baseline to characterize the simulation-reality gap. However, Sphinx does not enable read/write access to the whole sensor space, a requirement to integrate it with World-In-the-Loop. So as a second platform, we use a simulator built on the Unity framework [92], which has already been used for drones in the past [123], focusing mainly on the graphical aspects such as the camera readings given a drone's pose. This second simulator's data is mixed with real sensor data.

We designed three distinct scenarios and goals for the Parrot to achieve. The real-world tests were run in an indoor flight cage of $6m \times 6m \times 2.5m$ instrumented with a Vicon infrared motion capture system [359] that allowed precise tracking of the drone to obtain further data for some of

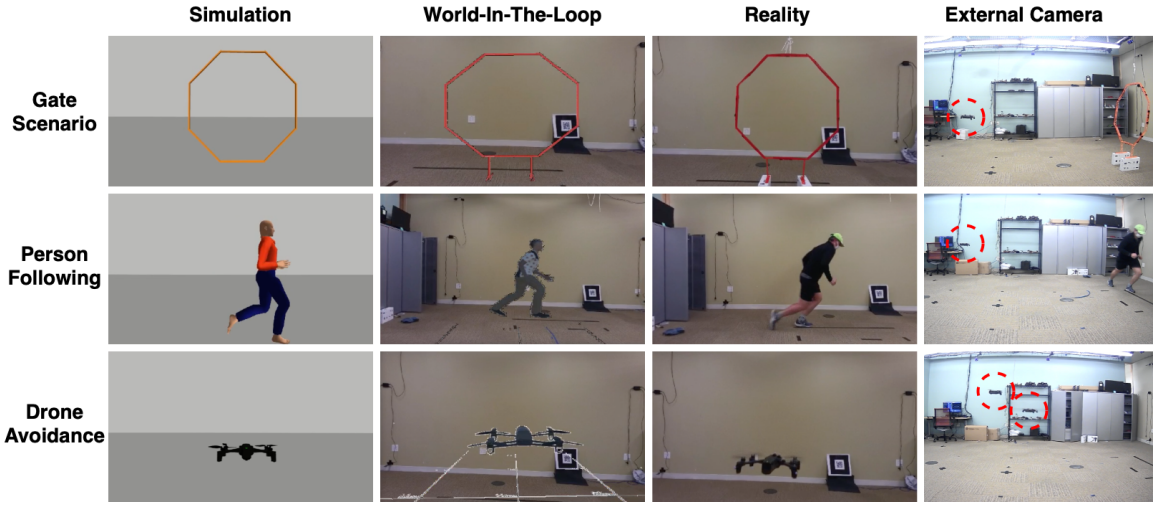


Figure 4.3: The camera sensor data that is fed into the AS software during simulation, mixed-reality, and reality for all three test scenarios. The final column shows an external camera with the drone highlighted in dashed lines.

the scenarios. A description of each scenario is given below.

Scenario 1: The first scenario had the drone fly through a gate [171], as seen in the top row of Figure 4.3. To do this, we built a subsystem that uses visual cues from the camera to navigate through the gate before stopping [133]. The subsystem works by identifying the gate and navigates the quadrotor towards the center of the gate's mass using a PID controller. Orange gates with a diameter of $1m$, and $0.5m$ were selected to allow for both an easy and challenging scenario. In the challenging scenario, the quadrotor has only 10cm between the propellers and the gate's sides if it is centered correctly. The gate is placed in front of the quadrotor $3m$ away. A failure occurs if the quadrotor touches any part of the gate at any time.

Scenario 2: The second scenario was person following [320], as seen in the second row of Figure 4.3. We built another subsystem that used object detection based on the camera to track and follow the person. An existing object tracking algorithm, YOLO [296], generated a bounding box around the person object. The subsystem uses the bounding box's center to align the quadrotor with the person while using the bounding box's area to keep the person a set distance away. In this scenario, a person started $3.5m$ away from the drone and either walked or ran between the starting

point and another point perpendicular to the quadrotor $1.5m$ away. When the drone moves outside a predefined area while attempting to follow the person, we assert a failure as it risks colliding with external obstacles or the person. The area was set such that the quadrotor was allowed to overshoot by at most $1m$ horizontally and needed to maintain between $2.25m$ and $4.25m$ away from the person at all times.

Scenario 3: The final scenario was obstacle avoidance [379], shown in the final row of Figure 4.3. We developed a third subsystem using camera based object detection to avoid incoming obstacles. We extended the object tracking implementation from the previous scenario and measured the bounding boxes of objects to judge whether an item was moving towards the quadrotor. If that is the case, the quadrotor will attempt to avoid it by moving upwards. In our scenario, we placed two drones $2.5m$ apart. The first drone's goal was to avoid the incoming drone. The second drone would take off after a set time, and once at the same height as the first drone, fly towards it. We developed two test cases. The first had the incoming drone reach a velocity of $0.5m/s$. The second had a velocity of $1m/s$. For this scenario, we considered the two drones colliding a failure.

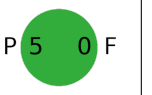
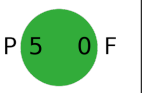
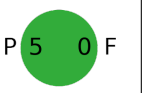
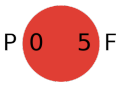
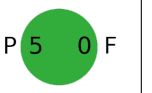
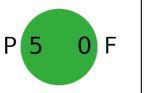
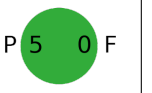
Each scenario was developed so that the quadrotor could reliably complete each of the tasks in simulation. This represents a typical development process where an autonomous system is first perfected in simulation before real-world tests begin. After the quadrotor passed each simulation scenario, it was run in mixed-reality and then in the real-world (when feasible).

4.1.2.2 RQ1: Reducing the simulation reality gap

Table 4.1 summarizes the results across the three scenarios. Overall, 5 tests were run for each of the 2 variants of the 3 scenarios, resulting in a total of 30 tests. The number of passing (P) test cases and failing (F) test cases were recorded for each scenario. Table 4.1 shows that although all tests pass in simulation, World-In-the-Loop found failing test cases. Moreover, if World-In-the-Loop found a failing test case, there was always a failed test case in reality. Similarly, if World-In-the-Loop found no failing test cases, there were no failed tests in reality.

When considering each scenario in isolation, for example, gate navigation, we notice that the drone can always navigate through the gates without any failures in simulation. Using World-In-the-Loop, we find that the drone can successfully navigate through the large gate. However, for the small gate, the drone crashes 80% of the time. To further assert that the mixed-reality results represented how the drone would behave in reality, the tests were repeated in the real-world. We can see that for the large gate, the drone can successfully navigate through it without failure. However, when running the drone through a small gate in reality, it successfully passed once and then failed on the next attempt. After the failed test, testing was stopped due to the cost of damaging the drone.

Table 4.1: Results from each scenario

Scenario	Test Case	Simulation	WIL	Reality
Gate Navigation	Large			
	Small			
Person Following	Walk			
	Run			
Obstacle Avoidance	Slow			
	Fast			Too Costly

The person following scenario produces results similar to that of the gate navigation. Moving from simulation to mixed-reality and then reality, we notice cases where the drone fails. We also observe that there are more failure cases in reality than in mixed-reality. We believe this is partly due to the person's movement's variation not being modeled accurately in simulation. Consider the velocity of the person while walking. The average standard deviation in the velocity of the walking person in simulation was $1.51m/s$, in mixed-reality it was $1.32m/s$, and in reality it was $2.27m/s$. This additional variation caused the quadrotor to move outside the stipulated area to track the

person in reality. Regardless of this variation, World-In-the-Loop still identified at least one failing instance in mixed-reality, reducing the simulation-reality gap.

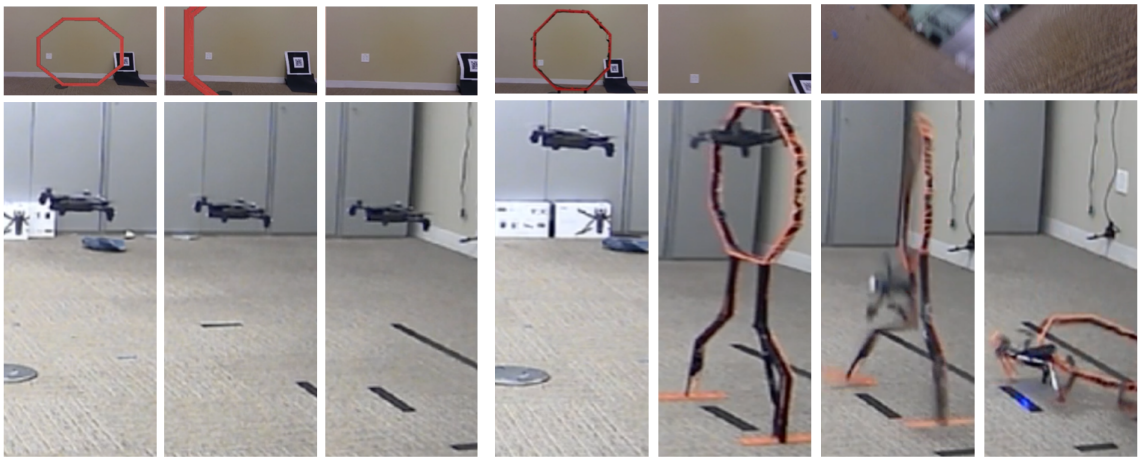
The obstacle avoidance scenario shows similar trends. Failure cases start to appear in the fast drone scenario during mixed-reality. However, due to the expense of failure in the real-world, where a failure would likely destroy two drones, the fast test case was never attempted in the real-world.

RQ1 Finding: We found that the quadrotor's simulated behavior does not always reflect its real behavior due to the simulation-reality gap, but World-In-the-Loop can reduce that gap producing results more closely aligned with that of tests performed in the real-world.

4.1.2.3 RQ2: Reducing the cost of executing in the real-world

The second research question explores the potential of World-In-the-Loop to reduce execution costs and field failures compared to traditional real-world testing. To answer this, we perform a qualitative analysis. Specifically, when considering the gate scenario, tasks such as adjusting the gate's position, altering its size, maintaining vertical alignment, and resetting it post-collision are significantly faster, cheaper, and more precise in a mixed-reality environment than in a real-world setting. For example, in mixed-reality, these tasks require only the modification of a set of variables in code. In the real world, however, they involve constructing a gate, taking precise measurements of its size and placement, securing the gate to ensure it remains stable against the quadrotor's downdraft produced by the propellers, and repairing both the drone and the gate after any incidents.

Similar observations were noted in other scenarios. For instance, the person-following scenario in a real-world setting necessitates an additional human participant, whereas in mixed-reality, this requirement is fulfilled through simulation. Implementing the drone avoidance scenario in the real world would require two drones equipped for communication and processing. The communication must be implemented in such a way as to ensure that there is no interference. Additionally, it requires twice the preparation effort, as twice as many batteries and preflight checks need to be performed. Finally, it also requires an additional backup pilot, a common practice when flying autonomous drones in confined spaces. In contrast, mixed-reality allows for one physical drone, with the other being virtually simulated.



(a) Mixed reality (b) Real-world.

Figure 4.4: Gate navigation failure in mixed-reality and reality.

The final consideration is the reduction of failure costs. Real-world failures in the tested scenarios could lead to damaged drones or harm to humans. In a mixed-reality context, such failures involve interactions with simulated entities. Consider the small gate navigation scenario: during World-In-the-Loop testing, collisions are recognized through data overlay from the Unity simulator, marking failures as Boolean flags. This avoids harm to the drone operating in an unoccupied real-world space, as illustrated in Figure 4.4a. This figure displays three frames showing views from both onboard and external cameras, where the drone navigates through the virtual gate and collides without real-world repercussions. Conversely, a failure in a real-world test, depicted in Figure 4.4b, results in physical contact with the gate, causing damage to both the drone and the gate.

Parallel conclusions are drawn from other scenarios. The person-following scenario, when conducted in reality, incorporates conservative measures to safeguard humans, a constraint that is relaxed in the World-In-the-Loop setting. Likewise, the incoming drone avoidance scenario in a real-world setup risks damage to two drones, a risk that is mitigated through World-In-the-Loop testing.

RQ2 Finding: The study showcases examples of cost reductions in both execution and failure across three tested scenarios within a mixed-reality environment. These reductions are evident in terms of execution efficiency and minimized risk of damage or harm to humans.

4.1.3 Summary

We have introduced a novel approach, World-In-the-Loop simulation, to narrow the simulation-reality gap by integrating sensor data from both the simulation and the real world. This method provides a framework for validation in a mixed-reality environment, where the balance between simulated and real elements can be adjusted according to the tester's needs. Our study demonstrates how this approach can reveal behaviors more closely aligned with those observed during real-world testing, while reducing both the cost of execution and failures.

Finally, although World-In-the-Loop facilitates the injection of simulated objects into physical environments, fully exploring the physical semantics of the autonomous system remains costly. Consider, for example, the challenge of replicating a drone flying through a gate under varying wind conditions. World-In-the-Loop could reduce the cost of using elements associated with the physical environment, such as the gate; however, the wind would still need to be simulated manually in the real world. In subsequent work, we aim to explore methods to reduce the costs associated with further examining the physical semantics of the system, specifically through the application of real forces to an autonomous system during real-world testing.

4.2 Mimicking Forces on a Quadrotor Through a Haptic Suit

Autonomous systems need to be tested under various scenarios and conditions to ensure their safe operation. In these scenarios, autonomous systems will be exposed to numerous forces, each with an impact that can vary depending on the physical semantics of the given autonomous system. Validating that an autonomous system can handle such external forces is crucial, as these forces can significantly alter the behavior of a system. For instance, consider how you have to counteract strong gusts while driving on a highway on a windy day, whereas on a calm day, the car travels

smoothly.

While validating external forces is important for all autonomous systems, it is particularly critical for systems whose physical semantics are inherently unstable. A stable system will naturally return to a safe and stable state after an external force is applied, with no additional input from a user or the autonomous agent. Examples of such systems include fixed-wing passenger aircraft, which are designed to level out when given no command. Another example is a car, whose wheels will always try to return to the straight position while driving. These systems, while needing to handle external forces, will naturally return to a safe state if the autonomous agent fails and provides no commands.

However, this is not the case with unstable systems. These systems must constantly make adjustments and perform some level of control to maintain a safe state. Specifically, it is the responsibility of the autonomous system to maneuver into a safe state after an external force is applied. There is no inherent safety designed into the system, without appropriate responses, the system will naturally tend toward instability and potentially crash. Legged robots [40], for example, need to take a step back to counteract someone pushing them from the front, or they risk failure and potential damage from falling over. Quadrotors [124] also need to make micro-adjustments to remain airborne; a strong wind or an impact from an object will result in a crash if the control mechanisms and motors cannot overcome the force.

In this section, we will take a closer look at drone systems, such as quadrotors, whose physical semantics need to be thoroughly validated before they can be used in real-world conditions. These multi-rotor aircraft exemplify advanced developments in robotics and automation. Their natural instability allows them to be both versatile and agile under a variety of conditions. This allows them to perform complex, precise, and lightning-quick maneuvers, making them ideal for a myriad of applications. These applications include carrying sensors and objects [273, 32]; balancing inverted pendulums [46, 130]; juggling balls [86, 253]; manipulating objects [326, 317]; and even launching from aquatic environments [16].

In these applications, given the system's physical semantics, external forces significantly impact the drone's behavior. For instance, carrying objects introduces a downward force; balancing pendulums induce a swinging motion; interacting with balls results in intermittent impacts; manipulating

objects leads to combined downward and rotational forces; and launching underwater involves overcoming resistance, which dissipates once airborne. For this reason, we need to be able to effectively validate them under a variety of external forces.

However, test execution for drones in the real world presents notable challenges. Consider a drone that must maintain its position while delivering cargo. First, precisely validating the effects of external forces, such as wind, would require specialized mechanisms like wind tunnels. Second, validating the range of behaviors the drone could exhibit would entail testing various cargo configurations (e.g., weight, shape); this process can be laborious, as testers often need to set up each scenario manually. Third, some forces are extremely difficult to replicate, such as those arising from the subtle interplay between gusts of wind and cargo configuration. These challenges make validating drone behavior in the real world resource-intensive, complex, and sometimes impractical.

4.2.1 Approach

This section introduces a framework designed to address these challenges. Drawing inspiration from haptic feedback systems in interactive technology that simulate force interactions for user immersion [132, 165], our framework simulates real-world forces on the drone during the test execution phase. The framework aims to mimic the forces a drone may experience during flight, enabling engineers to validate their drones physical semantics under various scenarios more efficiently, quickly, and easily.

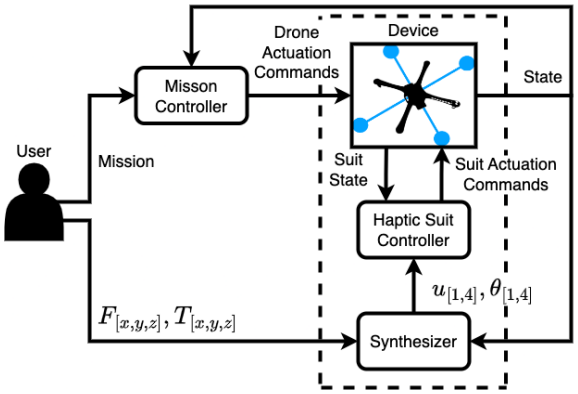


Figure 4.5: Overview of the Framework

4.2.1.1 Overview

Figure 4.5 presents the framework's structure, which includes three primary components: 1) an electro-mechanical attachment for the drone, equipped with directional propellers capable of pro-

ducing a range of forces and torques; 2) a synthesizer that converts specified force $F_{[x,y,z]}$ and torque $T_{[x,y,z]}$ into target motor velocities u_i and angles θ_i for each motor i, utilizing the inverse kinematic equations of the attachment; and 3) controllers that monitor the attachment and drone states, employing u_i and θ_i as setpoints to activate the attachment propellers to achieve these targets.

4.2.1.2 Haptic Suit Device

The following requirements guided the haptic suit design. First, the device needed to generate forces with a wide range of magnitudes and directions to support many scenarios. Second, the device needed to mount on the host drone without any point of contact with other external entities in order to reduce flying constraints or interference. Third, the device needed to minimize the disturbance to the drone's normal behavior to avoid failures that the introduction of the suit may cause.

The suit design follows from those requirements. Conceptually, the suit is elegant in its simplicity in that it adopts the host-drone structural design. Without loss of generality and to facilitate the explanation, we exemplify the integration of the haptic suit to a quadrotor as shown in Figure 4.6. For a quadrotor, shown in Figure 4.6-A, the suit device consists of four arms, each one with a rotor at its end, as shown in Figure 4.6-B, reflecting the design of the drone. In the suit, each of the i ($i = 4$ for quadrotor) rotor's thrust F_i and torque values T_i can be independently controlled by varying the motor speed u_i. However, unlike a typical drone where F_i is always perpendicular to the drone, the suit can independently rotate each arm by θ_i through additional motors at the base of each arm.

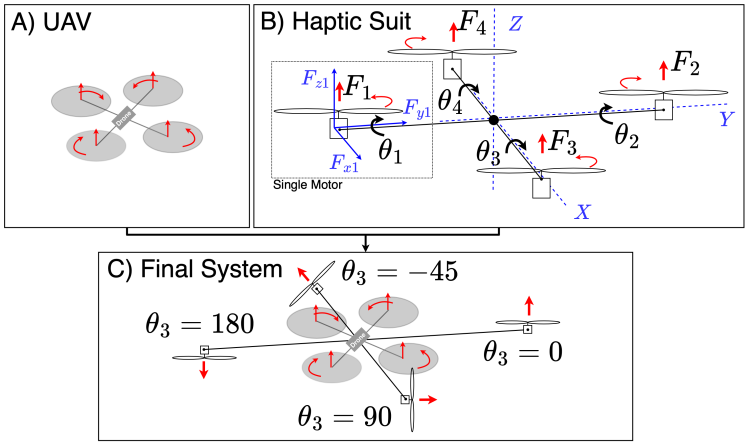


Figure 4.6: Quadrotor, Haptic-Suit, and Haptic-Suit integrated with Quadrotor (grey circles represent quadrotor propellers).

To allow for precise control, each arm is fitted with an IMU reporting its attitude. This allows us to vary the magnitude of F_i and T_i by varying u_i, as well as vary the direction by varying θ_i, to create a wide range of forces on the drone regardless of the drone's current pose.

The integration of the haptic suit and drone is shown in Figure 4.6-C. The haptic suit is mounted onto the drone with a shift of 45 degrees. This integration meets our requirements in the following ways. First, as each arm has a motor that can independently rotate, it can generate a wide range of forces, allowing for the simulation of a wide range of scenarios. Second, the suit is not attached to anything but the drone, thus not limiting the drone to any specific environment. Third, the suit is symmetrical, lightweight, and has arms that protrude out further than the drone's arms. This minimizes the changes in both the total and center of mass of the drone while also minimizing interference among airflow through propellers. These design choices reduce the chance that the suit will affect the normal behavior of the drone.

The conceptual design simplicity offers a rich space of trade-offs that will determine the magnitude of the forces that the suit can generate. For example, more powerful motors can generate greater forces. However, their weight, the weight of the arms required to support them, and the battery requirements may undermine some of the forces they can generate and the preciseness of the overall force manipulation. The arms' length and rotational speed can also affect the type and magnitude of forces that can be applied. This also indirectly affects the airflow of the other drone's rotors. The profile of all these elements can affect the dynamics of the drone, from adding weight to changing the drag coefficients. We will discuss the particular choices we made for one instantiation of the framework under Section 4.2.1.7.

4.2.1.3 Force to Control Set Points

Given a haptic suit device, the framework must convert a set of user-defined forces $F_{[x,y,z]}$ and torques $T_{[x,y,z]}$ into a set of motor speeds $u_{[1,4]}$ and rotations $\theta_{[1,4]}$.

For example, if a user wants to simulate carrying a package, which is equivalent to applying negative force in the Z direction ($-F_z$), we would need all motors to spin at equal speeds ($u_1 = u_2 = ...$) such that they generate a force of the correct magnitude (F_z). Furthermore, each of the

motors would also need to be pointing directly downwards ($\theta_{[1,4]} = 180$), so that the direction of the force is in the same direction as gravity ($-F_z$) to exert the same force as a package. Similarly, if a user wants to mimic a horizontal wind with a force F_y, whose magnitude is the same as that of the first example, then motors 2 and 3 would need to create forces in the F_y plane by rotating ($\theta_{[3,4]} = 90$), and each would need to operate at double the speed of the first example to compensate for the lack of motors 1 and 2.

To perform such conversion computations, we define a set of kinematic equations for the haptic suit. Consider motor 1 in Figure 4.6-B, with target speed u_1 and rotation θ_1. The force F_1 and torque T_1 generated by that motor are $F_1 = k_f \times u_1$ and $T_1 = k_m \times u_1$ where k_f and k_m are the proportionality constants for thrust and moments respectively [354]. Since the capability to rotate the motor affects the forces and torques direction, we must decompose F_1 and T_1 into (F_{x1}, F_{y1}, F_{z1}) and (T_{x1}, T_{y1}, T_{z1}). Equations 4.1 showcase the decomposition of F, with similar equations holding for T.

$$\begin{aligned} F_{x1} &= -F_1 \times sin(\theta_1) \\ F_{y1} &= 0 \\ F_{z1} &= F_1 \times cos(\theta_1) \end{aligned} \tag{4.1}$$

We can now generalize these equations to a full suit as per Equations 4.2, for the first two motors

$$\begin{aligned} F_{xi} &= -F_i \times sin(\theta_i) \\ F_{yi} &= 0 \\ F_{zi} &= F_i \times cos(\theta_i) \quad \text{for } i \in [1, 2] \end{aligned} \tag{4.2}$$

A similar set of equations can then be used for the next two motors as shown in Equations 4.3.

$$
\begin{aligned}
F_{xi} &= 0 \\
F_{yi} &= -F_i \times sin(\theta_i) \\
F_{zi} &= F_i \times cos(\theta_i) \quad \text{for } i \in [3,4]
\end{aligned}
\tag{4.3}
$$

These equations are derived from Equation 4.1 and thus $F_{yi} = 0$ for motors 1 and 2, while $F_{xi} = 0$ for motors 3 and 4, as these motors are incapable of rotating in the Y and X planes respectively. To compute torque we use a similar derivation except that since motors on opposite sides of an arm rotate opposite each other, their torque cancels out, so the terms for T_2 and T_4 are negated. The final step is to compute the total force F_x, F_y, F_z as per Equation 4.4.

$$
F_{[x,y,z]} = \sum_{i=1}^{4} F_{[x,y,z]i} \tag{4.4}
$$

Computing the torques T_x, T_y, T_z must take into consideration the additional torque placed on the system by any unbalanced forces in any given directions as shown in Equation 4.5. For example, a torque around the X axis can be created by an imbalance of the forces in the Z direction between motors 1 and 2.

$$
\begin{aligned}
T_x &= \sum_{i=1}^{4} T_{xi} + (F_{z1} - F_{z2}) \\
T_y &= \sum_{i=1}^{4} T_{yi} + (F_{z3} - F_{z4}) \\
T_z &= \sum_{i=1}^{4} T_{zi} + ((F_{x1} - F_{x2}) + (F_{y3} - F_{y4}))
\end{aligned}
\tag{4.5}
$$

To obtain the speed u_i and rotations θ_i needed to generate a given force F and torque T, we compute the inverse kinematics. Given the complexity of the equations and the fact that the number of degrees of freedom is different from the number of variables, we approximate the inverse using

numerical methods described in more detail in the implementation.

4.2.1.4 Controller

The final step is to actuate each of the suit's motors and arms into the correct configuration. The framework uses two traditional PID closed-loop controllers per arm. The first controls the motor speed, using the current motor speed as feedback, while the second uses the IMU attitude information as feedback. Note that having an IMU on each arm allows the framework to control the haptic suit independently of the drone's pose and behavior, which allows users to define forces in the world frame while ignoring the drone's pose. The set points are given by $u_{[1,4]}$ and $\theta_{[1,4]}$ produced by the previous framework component.

4.2.1.5 Generality

As defined, the framework can be directly instantiated to support multiple drone configurations under two conditions: 1) the device placement can coincide with the drone's center of mass, and 2) a symmetrical distribution of suit motors is a good fit for the drone structure and the intended forces and torques. As part of the framework presentation we introduced a 4-motor-suit configuration that fits, for example, common quadcopters and octocopters, in the following study we use a 2-motor-suit configuration that suffices to explore the target scenarios on a quadcopter, and extensions to more motors should be trivial to instantiate reusing a similar suit structure (with different motors and arms' length), body of kinematic equations, and controllers.

4.2.1.6 Limitations

The haptic framework's range of forces is constrained by three factors. The first results from the suit design, which mimics a typical drone. Drones control yaw by varying the speeds of motors that spin in opposite directions. A speed imbalance will produce a non-zero overall torque, resulting in a yaw. Three things occur when our suit rotates an arm, as seen in Equation 4.5. First, as the arm rotates, the force component in the F_z direction will decrease, resulting in a roll or pitch. Second, the arms rotation will induce a F_x, or F_y component, resulting in additional yaw. Third, the torque

generated by the motor's spin will also become imbalanced, adding additional roll, pitch, or yaw depending on the rotation. Therefore while we can generate a force in any direction, some forces may result in rolls, pitches, or yaws that can be modeled and should be monitored.

The second results from mounting the suit onto the drone. The drone's design and capabilities (e.g., weight, footprint, lift, controller) and its tolerance for the unintended perturbations caused by the suit (e.g., interference with airflow, additional weight) directly affect the design of the suit (e.g., the motors it can carry, the propellers dimension, power supply, the arms' length) and thus the magnitude of the forces our haptic suit can produce.

Finally the haptic suit is limited in that it requires a controller to function. Additionally, each time the haptic suit design changes, for example by adding or removing additional arms, the controller too needs to be redesigned. This could be minimized through learning the controller function, although that is not something explored in this work, and we leave that for future work.

4.2.1.7 Implementation

Our implementation, which is publicly available [144], uses a DJI F450 [85] with a Pixhawk controller [286] governed by the Freyja mission controller [318]. The experiments only required operation in the XY-plane and thus only required a suit with 2 arms. Since the drone's arms are 25cm, to minimize airflow interaction, we used 5mm-diameter and 29cm length carbon fiber tubes. Each arm connects to a DC motor for rotation driven by a dual-motor driver (Adafruit TB6612). Each arm also has an IMU (SparkFun BNO080) to determine its current rotation angle. 3D-printed components were used to firmly attach the suit to the drone.

Figure 4.7: Two-arm haptic suit prototype (marked using dashed lines) mounted on a DJI Flamewheel F450 drone [85].

The Flamewheel can carry up to 1kg of payload with limited noticeable behavioral changes,

which drove the selection of components. We used two tri-blade propellers (length: 5 inch, pitch: 4 inch) and two brushless motors (GARTT ML2204S 2300KV) that are driven by two electronic speed controllers (HGLRC 30A) to generate thrusts. The brushless motors are connected to two UXCELL GA12-N20 DC motors with reduction gears (6V, 70 RPM). Each motor was connected to the end of the carbon fiber tube. The DC and brushless motors were controlled using an Adafruit Feather M0, to which the user could send commands over WiFi. We equipped the suit with its own battery to avoid affecting the drones' behavior by sharing a power source. The suit, including battery, weighed under 800 grams to minimize the effects on the drone's behavior.

The remaining components are implemented on top of the ROS [328] as nodes developed in Python. The suit PID controllers are implemented on the Adafruit Feather M0. The numerical solver used to compute the inverse kinematics in Matlab [241] was the vpasolver.

4.2.2 Study

The goal of this study is to evaluate the potential of the haptic suit framework in facilitating the testing of physical semantics of a drone during the test execution phase. Specifically, the study seeks to address the following research question:

RQ1) How effective is the haptic suit framework in mimicking real-world forces on a drone?

RQ2) Does the haptic suit reduce the cost of executing tests in the real-world?

4.2.2.1 Setup

To answer this research question we build the suit, as well as developed 5 scenarios, each described below:

4.2.2.2 Suit

To assess the haptic suit framework capabilities, we used the prototype described in Section 4.2.1.7 with the suit shown in Figure 4.7. Through a set of empirical calibrations, we set the $k_f = 2$ and $k_m = 3$ of the kinematic model, as introduced in Section 4.2.1. The synthesizer that transforms

forces and torches to motor velocities and rotation angles was executed offline, while the controller was executed online as the scenarios were running.

4.2.2.3 Scenarios

We flew the drone under a series of scenarios, listed in Table 4.2, designed to expose different types of forces. The weight scenario produces a constant force that points in the Z direction. The wind scenarios produce either a constant force in the Y direction, or in the case of the gusting wind a force in the Y direction and torques in the X, Y, and Z direction sampled from a set interval.

Table 4.2: Each of the scenarios along with associated forces and torques.

Scenario	**Force and Torque (Newtons)**
Weight (Indoor)	$F_z = -3$
Steady Wind (Indoor)	$F_y = 1$
Gusting Wind (Indoor)	$F_y = f_1([0.5, 1]), T_{[x,y,z]} = f_2([0.1, 0.5])$
Pendulum (Indoor)	$F_y = f_3(p), F_z = f_4(p)$
Drop Weight (Outdoor)	$F_z = f(t) = \begin{cases} -5 & \text{if } 0 < t \leq 30 \\ 0 & \text{if } t > 30 \end{cases}$

The pendulum scenario produces a force in the Z and Y direction that varies based on the position of the drone p. The scenarios were performed indoors using a motion capture system, Vicon [359], that is capable of measuring the vehicle's pose at 200Hz with a 5mm accuracy. The outdoor drop-weight experiment follows a piecewise function based on the time t, and measurements were taken based on the commands generated by Ardupilot [20].

For each scenario, we compare the drone's behavior under the real external forces versus the forces induced by the haptic suit framework. The scenarios with real forces used physical weights, wind generated by fans, and attached pendulums to create a baseline.

For the weight scenario, we selected a 100g, 200g, and 300g cargo. For the wind experiments, we used an industrial fan capable of generating gusts of winds up to 4m/s. To create the pendulum scenario, we attached a hinge allowing a unidirectional motion to a carbon fiber rod that carried weights of 100g, 200g, and 300g. For the outdoor drop-weight scenarios, we used 500g cargo that

was dropped after a set time interval using a rope connected to a pin which, when pulled, released the weight.

4.2.2.4 RQ1: Effectiveness of replicating real-world forces

Below we look at how effective the haptic suit was at replicating real-world forces for each of the different scenarios.

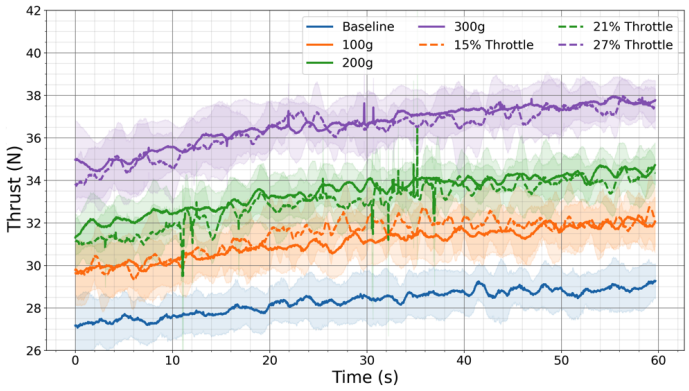


Figure 4.8: Average thrust at different real weights and weights induced by the haptic suit.

Weight Scenario: In this scenario, the drone was set to hover at 1m while carrying a set weight. Each of the experiments was run 5 times for 1 minute. We assess the drone's behavior in terms of the thrust commands sent by the controller Freyja [318] and the drone's altitude while carrying different weights.

Figure 4.8 shows the thrust commands sent to the drone by the controller in newtons (N). Solid lines represent thrust readings using real weights, while dashed lines represent thrust using the haptic suit. As expected, the more weight that is added, the higher the thrust command sent to the drone to maintain altitude. More interestingly, the thrust command's average and standard deviation are almost identical when using the haptic suit to replicate the real weights. This suggests that, from the

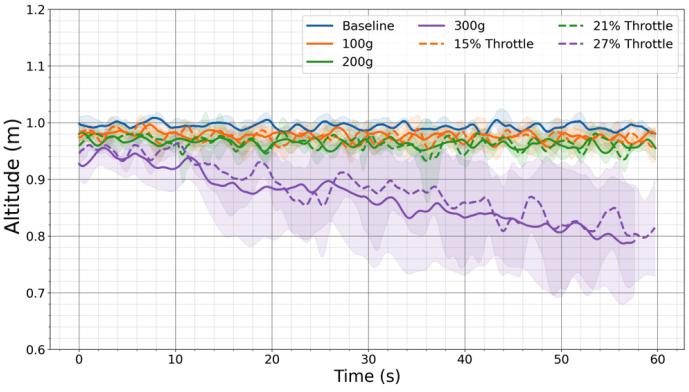


Figure 4.9: Average altitude induced using different real weights and weights induced by the haptic suit.

drone's perspective, the suit is exercising almost the same external forces.

Figure 4.9 shows the altitude of the drone as measured by Vicon. The average and standard deviations on altitude are almost identical when comparing the real and the replicated scenarios using the haptic suit. Even the slow decline in altitude at 300g observed in reality, likely caused by an erroneous drone configuration parameter, is replicated when using the haptic suit.

Wind Scenario: We first wanted to explore how the drone would behave under a constant wind force. Testing this often requires an elaborate environment to produce steady wind flows. It is not difficult to conjecture, however, how the drone should behave under a steady wind. We expect that the drone should lean against the wind to maintain its position. Thus, for this first wind scenario, rather than testing our drone against a real constant wind, we validated our hypothesis using the haptic suit to replicate the drone operating under steady winds.

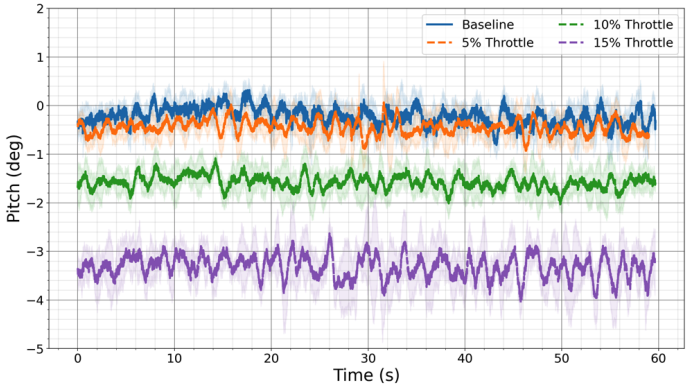


Figure 4.10: Average pitch reported at different constant wind velocities induced by the haptic suit

The test involved increasing the magnitude of the force in the F_y direction using 3 constant values. These correlated with suit throttle values (ω) of 5%, 10%, and 15%. Each of these configurations was run 5 times, and the average and standard deviation of the drone's pitch is shown in Figure 4.10. The pitch adjustments of the drone match our conjecture.

When the haptic suit's motors were set to 5%, the pitch was not significantly different from the baseline. However, as the horizontal force placed on the drone grew, the drone made more drastic pitch adjustments to counteract this force. At 15% throttle, which generates a wind equivalent of 1N of force, the drone must almost continuously perform corrections of up to five degrees.

For the varied wind scenario, we rely on industrial fans to mimic wind forces on drones. The airflow in the area where the drone is to hover was recorded between 2.9m/s and 4.1m/s, reflecting

the noisy airflow produced. To recreate this scenario using the haptic suit, we randomly varied both F_y and $T_{[x,y,z]}$ inside set intervals.

Each random sample was selected from a normal distribution to mimic the real wind gusts. The pitch of the drone was then recorded over 5 runs when exposed to the fans and 5 runs when using the haptic suit forces. As shown in Figure 4.11, the recorded pitch values show a greater amplitude than with the steady wind, as the drone must continuously adjust to different forces to maintain its position.

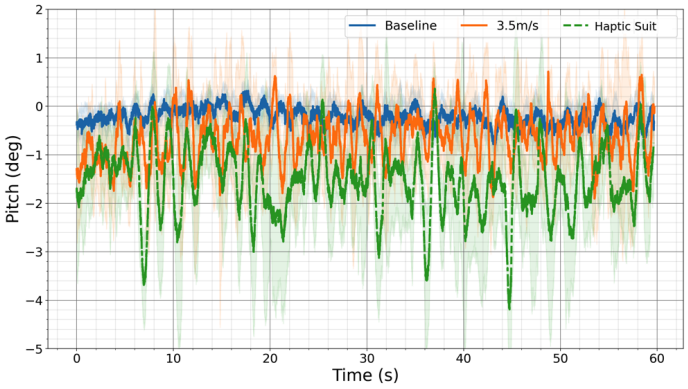


Figure 4.11: Average pitch induced by real fans and haptic suit.

We find that the behaviors are similar when comparing the real wind gusts and the haptic suit-induced ones. However, the haptic suit, on average, produces slightly more pitch than the wind gusts. This is possibly due to slight variations in the drone's altitude, which would take it in and out of the airflow produced by the fans. This difference points to the limitations of employing such current devices compared with the haptic suit.

Oscillating Pendulum Scenario: For this scenario, the drone flies at an altitude of 1m along the x-axis starting at -1m and traversing to 1m before aggressively turning back until it gets to the start point, where it would wait 20 seconds before repeating the process.

To define the forces induced by the pendulum that would serve as inputs to the haptic suit, we modified OpenAI's cartpole simulation [49] to match our scenario (our pendulum is facing down, has a range from 0-180 degrees). We modified the simulation to include friction and a PID controller that moved the base of the pendulum to a given setpoint p corresponding to the pose of the drone. Thus, the simulation pendulum angle would be updated as the drone moved. The magnitude and direction of the transformed force of the simulator were then sent to the force to control module, which forwarded the correct $\omega_{[1,2]}$ and $\theta_{[1,2]}$ to the haptic suit.

Figure 4.12 shows the drone's pitch. We were specifically interested in the 6-second mark when the drone tries to become leveled but is affected by the swinging pendulum. We notice how well the haptic suit mimics the real-world forces, with a maximum average difference of 3 degrees at 6s. We conjecture the slight differences observed can be attributed to the simplistic simulation model we used (oversimplified friction approximation) or communication delays (the suit was controlled from a wireless station). Overall, we find it promising that this rather complex scenario can be modeled so accurately using the haptic suit.

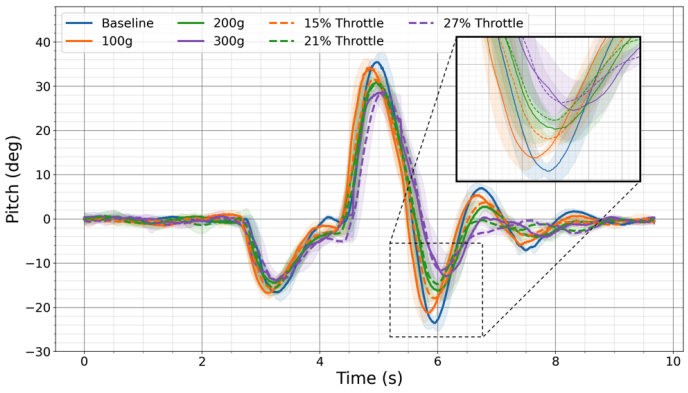
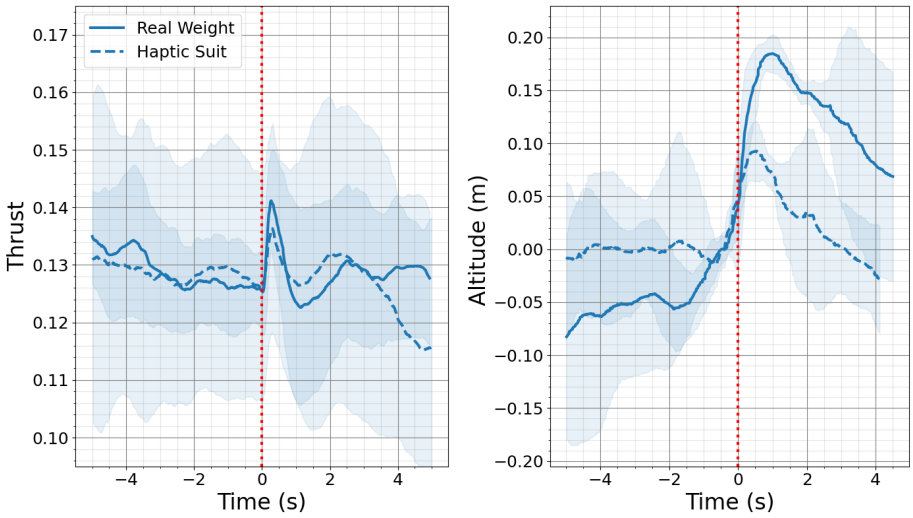


Figure 4.12: Average pitch induced by a pendulum with different weights and haptic suit.

Drop-Weight Scenario: In this outdoor scenario we manually flew the drone to roughly 2m above the ground while using Ardupilot's altitude hold mode to maintain the drone's current altitude without any pilot feedback. For the scenario we took off with a payload of 500g attached. After roughly 30 seconds, the payload is released. We expect that as the weight is released there is a sudden change in altitude accompanied by a change in thrust to recover the altitude.

Figure 4.13a showcases the average thrust (a unitless value reported by Ardupilot) when using the real weight and the haptic suit approximation, with time 0 corresponding to the weight being dropped.

In both cases, when the weight is dropped, there is a spike in thrust. Figure 4.13b showcases the average altitude changes from the original altitude at the time of dropping weight. As before, when the weight is dropped, there is a sudden increase in altitude when using both the real weight and haptic suit. We notice that when using the real weight, there is a larger increase in altitude likely caused by the friction in the weight release mechanism that was not accounted for in the force functions. This point again highlights one of the advantages of the haptic suit to overcome the



(a) Average thrust (b) Average altitude

Figure 4.13: Thrust and altitude when dropping real weights and using the haptic suit. Time 0s corresponds to time of weight being dropped.

limitations of existing practices that would have required a setup that includes a more sophisticated release mechanism to validate the drone's stability to changes in weights while flying.

RQ1 Finding: The study shows that in each of the five scenarios, the haptic suit is capable of replicating real-world forces. Specifically, we found that the drone's thrust, attitude, pose, and behavior under both real and haptic suit-induced forces appear to be very similar. Furthermore, the study demonstrated that when a failure was detected in a real-world test, the same failure was also detected using the haptic suit. This provides evidence that the haptic suit can detect real-world failures without the costs, limitations, and safety precautions required for real-world testing.

4.2.2.5 Impact of Haptic Suit on Test Execution Costs

Due to the complexities involved in obtaining precise quantitative data, we adopt a qualitative approach to analyze the impact of the haptic suit on reducing test execution costs.

Weight Scenario: In the traditional testing approach, each change of payload weight neces-

sitated a meticulous process: selecting the appropriate weight, accurately weighing it, securely attaching it to the drone, and ensuring stability before flight. This method was time-consuming and prone to errors, often requiring frequent interruptions to adjust or replace the weights. In contrast, the haptic suit streamlined this process remarkably. By simply inputting a command, the desired force could be accurately and consistently applied to the drone, eliminating the need for manual weight adjustments. This not only accelerated the testing procedure but also significantly reduced the potential for errors, demonstrating the haptic suit's efficiency in simulating varied payload scenarios.

Wind Scenarios: Testing the drone under wind conditions typically requires specialized and expensive equipment like wind tunnels, which are not readily accessible. To replicate wind effects, we initially used industrial fans, but this approach had significant limitations. Precisely controlling the wind strength and direction was challenging: it involved measuring the air velocity, carefully angling the fans, adjusting their location for optimal distance, and ensuring a consistent power supply. Moreover, this setup was confined to a controlled lab environment, making real-world replication far more complex due to unpredictable natural wind conditions.

In stark contrast, the haptic suit offered a more practical and controlled solution. By simply setting the suit to simulate equivalent wind forces, we could efficiently create consistent and repeatable wind conditions. This method not only bypassed the logistical hurdles of using fans but also provided a more scalable and realistic way to test the drone's response to varied wind intensities and directions, both in lab and real-world scenarios.

Oscillating Pendulum Scenario: In the traditional approach to simulate an oscillating pendulum effect on the drone, we faced a series of intricate challenges. First, it necessitated the development of a custom pendulum that could swing unidirectionally. This process involved some understanding of material science to ensure the use of materials that were both lightweight and sufficiently strong to support the swinging weight. Additionally, significant 3D design work was required to devise the hinge mechanism and to print it accurately.

Another major practical challenge encountered during real-world testing was during drone landing. The attached pendulum caused the drone to tip over upon landing, as it would land on the

pendulum, causing the drone to hinge and tip. This issue led to the creation of a specialized landing platform, designed to allow the pendulum to hang freely while providing adequate support to the drone's legs.

Conversely, replicating the pendulum forces with the haptic suit and simulation proved to be a less cumbersome task, albeit not as straightforward as the previous scenarios. We adapted an existing pendulum physics simulation, modifying the pendulum's base to correspond to the drone's position. The simulation handled the complex physics of pendulum movement, enabling us to directly apply the calculated forces and directions to the drone. This method significantly reduced the need for specialized mechanical design and material science expertise. Moreover, it simplified the replication process, eliminated the need for a specialized landing platform, and required less skill from the drone pilot, making the overall testing procedure more efficient and practical.

Drop-Weight Scenario: The drop-weight scenario, involving the release of a weight once the drone achieved stable hover, presented unique challenges in real-world replication. Initially, we developed a timed device that would automatically drop the weight after a preset duration. While this mechanism functioned adequately for lighter weights, it proved unreliable with heavier loads, often releasing them prematurely. Additionally, the timed nature of this device imposed undue pressure on test engineers to expedite pre-flight safety checks, risking procedural thoroughness. Any disruptions during these checks necessitated a complete reset of the process: de-powering the drone for safety, resetting the weight-dropping device, and then resuming the pre-flight protocols.

To overcome these issues, we shifted to a manual weight-dropping mechanism, which, while more reliable, introduced its own set of complications. This system required two operators — one to fly the drone and another to release the weight using a long string. Significant engineering effort was also needed to minimize friction in the release mechanism, ensuring that the act of pulling the string did not inadvertently apply additional force to the drone.

In contrast, the haptic suit streamlined this process significantly. It enabled us to simulate the effect of a weight being dropped by simply issuing a single command to cease the application of force. This approach eliminated the need for mechanical devices, reduced manpower requirements, and offered precise control over the timing and magnitude of the simulated weight release. The

haptic suit's capability to mimic real-world scenarios with such ease and accuracy underscored its value in efficiently conducting complex drone tests.

RQ2 Finding: Our findings indicate that the haptic suit increased testing efficiency and reduced both the cost and complexity of executing tests across all five scenarios, compared to a purely real-world test. Additionally, we found that the haptic suit enhances the reliability of certain tests, such as the wind scenario, where it enables more precise replication of conditions that are challenging to reproduce without the suit.

4.2.3 Summary

In this study, we introduced a haptic suit framework designed to apply various forces on a drone for validation purposes. Our findings indicate that the drone's thrust, attitude, pose, and behavior are consistently similar when subjected to both real-world and haptic suit-induced forces. This similarity suggests that the haptic suit effectively replicates real-world forces on the drone. We also observed that the suit successfully mimics unexpected behaviors seen under real forces, such as altitude loss in heavy-weight scenarios, reinforcing its utility as a complement to real-world testing. Additionally, we found that reproducing specific real forces, ranging from constant wind to complex nonlinear functions that vary with the drone's state, can be prohibitively expensive and challenging in a lab setting. However, our haptic suit can approximate these forces with minimal effort, offering a less resource-intensive method for validating drone behavior and testing the physical semantics of autonomous systems in realistic conditions.

4.3 Conclusion

This chapter explored two approaches to support test execution of autonomous systems. The first aimed to reduce the simulation-reality gap, which occurs due to simulations' inability to perfectly replicate the real-world physical environment. We created a mixed-reality environment constructed from sensor readings obtained both in simulation and in reality. Through this approach, we are able to reduce the simulation-reality gap by only simulating portions of the physical environment that

are too costly to actually include in the real-world. We subsequently demonstrated this approach using a commercially available quadrotor under three different scenarios, each with two variations, totaling six unique physical environments. In each case, we highlighted how using our approach, World-In-the-Loop simulation, we were able to detect failures that were not apparent in simulation before they appeared in reality.

The second approach aimed to reduce the cost and enable testing of an autonomous system's physical semantics under various real-world external forces. Specifically, we looked at how we could replicate external forces on a quadrotor from five different scenarios. Our approach involved developing a novel haptic suit capable of applying real-world forces to the drone, using a system that attached directly to the drone without external tethers or ground-based attachments. This system minimized the disturbance to the drone's nominal behavior, while allowing highly customized forces to be applied to the drone. We found that this device was capable of replicating all five real-world forces, and in some cases, was more reliable than traditional approaches.

Chapter 5

Test Adequacy Metrics

In software testing, test adequacy is used in the testing pipeline, after test execution to judge its effectiveness. This concept is best understood in terms of two critical sets: β, which represents all possible values of a adequacy metric, and $\alpha \subseteq \beta$, the subset of these values that are actually observed during testing. One of the goals of test adequacy is to compute both α and β, enabling developers to determine what is called the coverage of their test suites by evaluating the ratio of α to β. Traditionally, for software systems, both α and β are determined through abstractions of the input space. These abstractions allow developers to organize the input space into what are known as equivalent classes. The underlying principle is that inputs within a single equivalent class are expected to elicit the same system behavior. A test suite that covers more equivalent classes is considered more comprehensive, as it exposes a greater range of the system's potential behaviors. This ratio provides a quantitative measure of the extent to which potential system behaviors have been explored and tested.

However the manner and granularity in which these abstractions are computed significantly influences the scope of both α and β, as it changes the size of these equivalent classes. Applying this traditional approach to test adequacy encounters significant challenges when it comes to autonomous systems. The behavior of these systems is profoundly influenced by a dual set of factors: the system's state s, and its current physical environment c. These factors such as the vehicle's pose, speed,

acceleration, the topology of the road, surrounding traffic, signage, and other environmental objects collectively shape the vehicle's actions and subsequent behavior. Incorporating both s and c into a effective abstraction for computing α and β poses multiple challenges. Firstly, the vehicle's state is complex and can be represented in numerous ways, complicating the task of creating a generalized abstraction. Secondly, the real world presents a nearly infinite array of environmental scenarios, making the identification of equivalent classes within and the estimation of the total number of such environments challenging. Thirdly, the intricate and often subtle interactions between s and c significantly affect its behavior. This complexity necessitates that any abstraction must account for these dimensions in a combined and integrated manner.

The result is an intricate combination of system states s intertwined with the nearly boundless variations of the physical world c. This combination of factors leads to an extensive and diverse range of possible input pairs, far exceeding the capabilities of traditional software-focused abstractions. Consequently, this scenario necessitates a reevaluation and redesign of test adequacy measures, ones that are tailored to account for both the system's state and its physical environment.

5.1 Physical Coverage

We address the deficiencies of existing coverage criteria and the above challenges by introducing a complementary abstraction, *RRS* (Reduced Reachability Set), that identifies the most relevant areas of the environment to the autonomous system. It builds on two basic principles. First, an autonomous system's behavior can only be influenced by what it can perceive, and therefore the sensed environment c^{sen} and state s^{sen} of the vehicle are likely valuable and should be retained. Second, the physical space of these inputs can be geometrically approximated, and inputs with the same approximations constitute an equivalence class. From a technical perspective, those insights are leveraged by 1) incorporating the current state s^{sen} to compute the reachable set of the autonomous system to prune regions of the sensed input space that the vehicle cannot reach in a time horizon, 2) performing a geometric vectorization on the remaining input set to approximate its shape, and 3) parameterizing the RSS abstraction to trade approximation precision (resolution of the equivalence

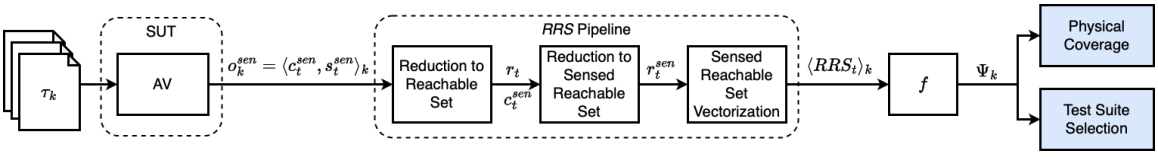


Figure 5.1: Overview - The approach starts with an initial test suite, and an autonomous system. It then executes the test suite on the autonomous system and passes the state and sensor data to the *RRS* abstraction pipeline. This creates *RRS* signatures which can compute *PhysCov*.

classes) for computational cost.

5.1.1 Approach

The goal of the approach is to approximate $\frac{\alpha}{\beta}$, the number of environment-state pairs seen in a given test suite (α), over the possible environment-state pairs (β). Figure 5.1 presents an overview of the *RRS* abstraction pipeline to compute that approximation.

5.1.1.1 Overview

The approach starts with a test suite composed of k tests, denoted as $\tau_k \in \mathcal{T}$. Each test is executed on the autonomous system, during which, at every time step t, the system records two key pieces of information: the environment as it is sensed by the system's sensors c_t^{sen}, and the system's own sensed state s_t^{sen}. This process of recording occurs continuously throughout the test. As a result, for each test of the in the suite, we accumulate a sequence of environment-state pairs, represented as $o_k^{sen} = \langle (c_1^{sen}, s_1^{sen}), (c_2^{sen}, s_2^{sen}), \ldots, (c_t^{sen}, s_t^{sen}) \rangle_k$.

This information is then fed into the pipeline, which has three stages. The first stage "Reduction to Reachable Set", defines the region around the autonomous systems most relevant based on the autonomous systems state. To do this, the *RRS* pipeline uses the autonomous systems state s_t^{sen} at each timestep t, to compute the autonomous systems's physical reachable set r_t at that time step, that is, the area or volume that the vehicle can reach given its current state. This step incorporates the internal state of the autonomous systems into the abstraction, as the reachable set requires both the dynamic and kinematic model and the autonomous systems's internal state s_t^{sen} (e.g., position, velocity, acceleration). We assume that the reachable set is the essential part of the sensed area, as

any object inside the reachable set is an obstacle that has the potential for collision, while any static object outside can not be hit regardless of the autonomous vehicle's behavior.

The second stage "Reduction to Sensed Reachable Set", identifies the parts of the environment most relevant to the current behavior of the autonomous systems. The approach assumes the vehicle's sensors capture c_t^{sen}, the portions of the environment that drive its behavior. Next, it computes $r_t^{sen} = c_t^{sen} \cap r_t$. Thus r_t^{sen} only contains sensor readings from the physical world inside the area or volume defined by the reachable set, which we argue is the region most important to the autonomous systems' decision at time t. By removing all portions of the environment that are not likely to affect the autonomous systems's decision-making, the approach reduces the environment such that $size(r_t^{sen}) << size(c_t^{sen}) << size(c)$. Here, $size()$ is conceptualized as "environmental complexity and extent" quantifying both the detail of the environment and the spatial dimensions under consideration.

The third stage "Sensed Reachable Set Vectorization", takes in r_t^{sen}, which is already significantly smaller than the physical environment but has a complex geometry and is represented as an innumerable continuous space. This stage provides a method to characterize this identified region so that it can be easily compared with others and quantified. The approach employs geometric vectorization to approximate r_t^{sen} using an array of vectors originating from the autonomous vehicle. The unique array of vector magnitudes is called an RRS signature. This process is object- and sensor-agnostic and can account for unexpected sensor readings, which might be experienced when operating in a new and unforeseen environment. The output of this final step is a sequence of RRS signatures. More specifically, for each environment-state pair $(c_t^{sen}, s_t^{sen}) \in o^{sen}$, the approach can compute an RRS_t signature. Therefore, by supplying the pipeline with k sequences o_k^{sen}, each of length t, the approach can compute $\langle RRS_t \rangle_k$, a set of k sequences of t abstract signatures that represent the autonomous systems' sensed physical environments as perceived by their states during the execution of each of the k tests τ_k.

The approach then passes the $\langle RRS_t \rangle_k$ sequence through a function f to convert it into a coverage vector Ψ_k. This function's semantics can vary depending on the user's needs. For example, in our study, f computes the coverage vector by reducing the $\langle RRS_t \rangle_k$ into the set of RRS signatures

$\{RRS_t\}_k$. This represents all unique *RRS* and thus environment-state pairs seen during testing. Alternatively, a more advanced function could count the number of times each *RRS* abstraction was seen, thus representing a count of the environment-state pairs seen during testing. Another f could, for example, account for the transitions between consecutive signatures.

When Ψ_k represents the set of *RRS* signatures, the approach can compute *PhysCov*, an approximation of $\frac{\alpha}{\beta}$. *PhysCov* represents the percentage of environment-state pairs experienced by the autonomous systems over all possible environment-state pairs. The approach computes $\alpha = |\{\Psi_1 \cup \Psi_2 \cup \cdots \cup \Psi_k\}|$, where $\Psi_k = \{RRS_t\}_k$. The approach can then compute β as the total number of possible *RRS* signatures. Computing β is possible as the approach knows the parameters to the pipeline, such as the maximum length of the *RRS* signatures and the total number of vectors used in the *RRS* signatures.

While the approach overcomes the problems defined in the problem statement, it also creates a metric whose properties provide additional benefits. First, this approach is general, with the reachable set accounting for any possible internal state and the geometric approximation accounting for any possible physical environment. Second, tests with the same *RRS* signature are grouped into equivalence classes. For a test to have the same *RRS* signature, it needs to have a similar reachable set and internal state, as well as similar sensed environments. When this happens, the autonomous vehicles would likely behave similarly, and thus we can maintain equivalence class consistency. Third, the approximation can be scaled to increase or decrease the approximation's fidelity by adjusting the number of vectors used. Fourth, the metric is finite; we can compute the denominator β prior to testing. Finally, our metric has a linear cost with respect to the number of vectors used in the geometric vectorization. The other significant computation is the generation of the reachable set, which is the target of much recent work as described in Section 2.3.3.

5.1.1.2 RRS Signature Generation

Given o_k^{sen} produced by test τ_k, the objective of the *RRS* pipeline is to generate a sequence of signatures $\langle RRS_t \rangle_k$. We will use the scenario in Figure 5.2 as a running example to show the three stages to compute such an *RSS* signature.

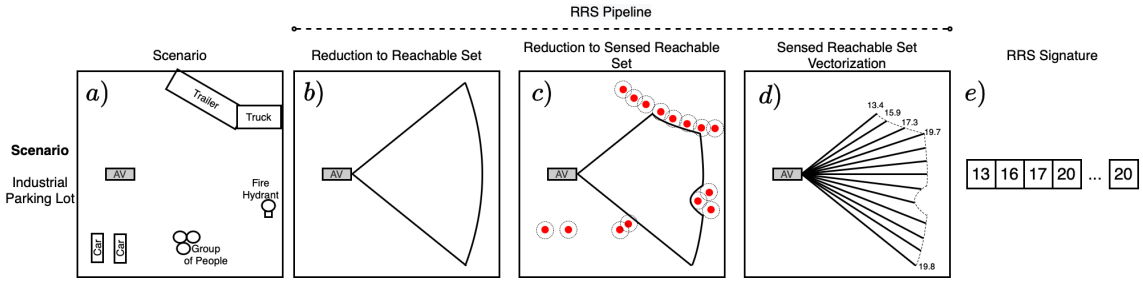


Figure 5.2: Given an autonomous systems and a scenario. The reachable set is computed to identify the regions more likely to affect the autonomous systems behavior given its state within a time horizon. The resultant reachable set is then constrained using the sensed environment. Last, the reduced reachable set is approximated using geometric vectorization, and the RRS signature is produced.

5.1.1.2.1 Reduction to Reachable Set

The reachability analysis component aims to identify which parts of c_t^{sen} may affect the autonomous systems's future actions. This is important as most systems perceive large portions of the environment that are unlikely to influence their behavior given the system's current state. Consider a case where c_t^{sen} is generated using a LiDAR. The LiDARs used on today's self-driving cars operate using a 360-degree field of view and can detect objects up to 300m away from the vehicle's current position [358]. This would result in an environment c_t^{LiDAR} that covers an area of $282743m^2$, most of which is not actively relevant to the autonomous systems. For example, suppose the autonomous system is moving forward at 70mph. In that case, the space behind the autonomous systems is likely irrelevant, while the space ahead of the vehicle is more likely to affect its behavior.

To identify the most relevant area, this component performs physical reachability analysis described in Section 2.3.2 and 2.3.3. An illustration of the reachable set of a ground vehicle is shown in Figure 5.2b (the cone-shaped region). The reachability set computation complexity depends on the system model's complexity. A car's model uses dynamics and kinematics where the state of the car can be described using $s = [x, y, v, \psi]^T$, which describes the x and y position, the velocity v, and the orientation ψ [193, 294]. The input to the system is $u = [a, \delta]$, which describes the acceleration a and the front wheel steering angle δ.

The output of this component is illustrated in Figure 5.2-b. A reachable set extends from the

autonomous vehicle in a conical area in front of it, and it includes the fire hydrant, the truck, and the trailer, as these items are within the potential area the vehicle can visit through a sequence of actions. The parked cars and groups of people are not reachable within the specified state and time horizon and thus lie outside of the reachable set.

5.1.1.2.2 Reduction to the Sensed Reduced Reachable Set

The autonomous system continually captures and processes c_t^{sen}. A wide range of sensors available for different autonomous systems provide some form of spatial awareness, and thus c_t^{sen} might come in various forms. Figure 5.2-c shows c_t^{sen} as a point cloud, a collection of points representing the sensed objects around the autonomous systems.

This component integrates c_t^{sen} with the previously computed r_t. To account for the limited resolution of the sensors, our approach inflates each point using a user-defined parameter λ represented as the dashed lines around each point in Figure 5.2-c. This step ensures that when geometric vectorization occurs, each vector will not pass through obstacles sensed with limited resolutions. However, as the resolution of sensors increases or the number of sensors increases, the need for inflation reduces. Any region of the reachable set that intersects with this inflated region is removed. Figure 5.2-c shows the reduction of r_t to r_t^{sen}, which now contains the reachable set constrained by the sensed environment. The resulting reachable set is constrained by the truck and the fire hydrant the vehicle could reach in the specified time horizon.

5.1.1.2.3 Reduced Reachable Set Vectorization

Once we have computed r_t^{sen}, the final step, shown in Figure 5.2-d is to convert it from its current polytope representation to a concise numeric characterization. We resort to a vector approximation inspired by the centroid-to-boundary shape analysis technique [226]. This technique approximates complex shapes by computing the distance from a central point to all boundary points of the shape. More specifically, given a specified number of vectors, the approach samples the r_t^{sen} space with vectors whose origin is the autonomous systems and magnitude is defined by their intersection with the bounds of r_t^{sen}, quickly providing a characterization of the sensed and reachable space we call

Algorithm 5: Geometric Vectorization Algorithm

```
1 Given r_t^sen, s_t, n, 𝒟, ℐ
2     RRS = []
3     for i in n do
4         angle = 𝒟[i]
5         v = compute_vector(r_t^sen, s_t, angle)
6         discretized_v = round(v, ℐ[i])
7         RRS.append(discretized_v)
8     end
9     return RRS
```

the RRS signature.

Algorithm 5 describes the process in more detail. The inputs are r_t^{sen}, the current state of the autonomous systems s_t, the total number of vectors n used to characterize the space, the spread $\mathcal{D}$ of the vectors which define the angles between vectors, and the tick intervals $\mathcal{I}$ which defines at which intervals vectors can be discretized. The algorithm loops through the total number of vectors n in line 3. For each vector, it first determines at what angle the vector should be from the centerline defined by the autonomous systems's direction of travel from the spread $\mathcal{D}$ as shown in line 4. For example, $\mathcal{D}$ might define that vectors are spaced evenly, 4 degrees apart from each other, spanning from the centerline. In line 5, the algorithm computes the vector's magnitude from the autonomous systems's origin at the specified angle until it reaches the edge of the reachable set r_t^{sen}. Next, in line 6, the vector v is discretized by rounding to the nearest value defined by the tick intervals $\mathcal{I}$. For example, if the currents v's magnitude was 3.25 and $\mathcal{I}$ was defined as $(1, 3, 5)$, then v would be discretized to the value 3. Finally, in line 7, that discretized vector magnitude is added to the RRS signature before being returned in line 9.

5.1.1.3 Usages of RRS Signature

The final step in the pipeline is to convert the sequence of $\langle RRS_t \rangle_k$ signatures into a coverage vector Ψ_k. Multiple types of coverage vectors can be computed, from one based on the unique signatures exposed by a test to one that considers the number of times each signature is executed or the transitions between signatures over time. However, for the rest of the paper, we focus on the simplest notion of signature coverage, identifying all unique RRS signatures in $\langle RRS_t \rangle_k$,

$\langle RRS_t \rangle_k \rightarrow \{RRS_t\}_k = \Psi_k$, which is similar to converting a trace of lines of code covered into a coverage vector just containing the unique lines that were executed.

5.1.1.3.1 PhysCov Computation

PhysCov aims to capture how much of the relevant physical environment was covered by a test suite and is defined in equation 5.1. Intuitively, this metric represents the percentage of distinct environments perceived that are relevant to the autonomous systems given its state. The number of seen distinct *RRS* abstractions α is computed from the union of each of the coverage vectors from all tests, such that $\alpha = \Psi_1 \cup \Psi 2 \cup \cdots \cup \Psi_k$. To compute β, the approach enumerates all possible *RRS* signatures. Equation 5.1's denominator takes the product of all possible magnitudes each vector can obtain and passes it through f to compute β. This, in effect, enumerates all possible combinations of the tick intervals $\mathcal{I}$, and thus we are left with all possible combinations of *RRS* signatures. Then to enumerate all possible Ψ, we pass all possible combinations of *RRS* signatures through f. A byproduct of our approach is that we can control the total number of *RRS* signatures and vary the granularity of *PhysCov* by varying n or $\mathcal{I}$.

$$\text{PhysCov} = \frac{\alpha}{\beta} = \frac{\Psi_1 \cup \Psi 2 \cup \cdots \cup \Psi_k}{f\left(\prod_{i=0}^{n}\left(|\mathcal{I}\,[i]\,|\right)\right)} \tag{5.1}$$

5.1.1.3.2 Test Suite Selection

Test selection works on the principle that coverage vectors Ψ are a good proxy for identifying equivalent tests. If two tests produce the same Ψ then we judge that the tests likely exposed the system to the same environment-state pairs and thus likely test the same behavior. Once the approach identifies each of the equivalent coverage vectors, it can select a subset of the test suite $\mathcal{T}^{select}$ with only a single test from each equivalence class $\mathcal{T}^{select} \subseteq \mathcal{T}$, where $\mathcal{T}^{select} = \{\tau_i, \tau_j \in \mathcal{T} | \Psi_i \neq \Psi_j\}$.

5.1.1.3.3 Test Suite Generation

The missing Ψ, $\Psi^{missing} = \beta - \alpha$, can be the drivers of a targeted test suite generation effort. Once the approach has identified $\Psi^{missing}$, it can construct each of the missing RRS signatures using $RRS = f^{-1}(\Psi^{missing})$. Then for each $RRS \in \Psi^{missing}$, the approach can generate an environment-state pair that would result in that specific RRS signature being formed.

5.1.1.4 RRS Generalization

Computing RRS signatures is not restricted to 2D environments, particular sensor types (as long as they provide a spatial characterization), or autonomous ground vehicles. For example, a quadrotor reachable set would resemble an upside-down cone, with the quadrotor in the middle. Sensed obstacles could be used to remove portions of the reachable set, which could then be approximated using geometric vectorization.

5.1.1.5 Limitations

Our approach has two main limitations. First, it assumes that the approach has access to either a sensor or an internally sensed scene representation that is spatially represented. By this, we mean, for example, LiDAR generates point clouds which have some physical spatial meaning that our approach can consume, while camera images require an additional stage to convert pixel values into a spatial representation of the scene. While many autonomous systems do this internally, some do it explicitly. Examples of these include systems designed with end-to-end neural network architectures, which consume raw images and output actions. In such cases, our approach would need an additional stage to convert the scene into a spatial representation.

The second limitation is how our approach vectorizes the space. Currently, our approach vectorizes it using vectors drawn from a selected point. However, these vectors may miss many potential obstacles, for example, consider an extremely thin pole which the vector passes by. Future versions of this approach may consider alternative methods for vectorizing the space, such as breaking the space up into smaller sectors whose areas can be computed, allowing the identification of even the

smallest obstacles.

5.1.2 Study

We aim to answer the following research questions:

RQ1) How effective is the coverage vector Ψ at grouping equivalent environment inputs such that they cause similar behaviors? Additionally, what is the impact of the *RRS* parameters on *PhysCov*?

RQ2) How effective is *PhysCov* at selecting tests that induce unique failures?

RQ3) Can *PhysCov* distinguish similar from different real scenarios?

5.1.2.1 Setup

We evaluate *PhysCov* on three increasingly complex environments, a traffic kinematic simulation, a high-fidelity simulation, and data taken from a real autonomous system. This mimics real-world development, where autonomous systems are first developed in simple simulated environments, then in complex simulations, and finally tested in the real world. We now describe how we set up our environments, what baselines we compared against, what evaluation criteria were used, and how we instantiated our PhysCov pipeline.

5.1.2.2 Environments

We used three different environments in the study. The first two were simulations, and the final was data collected in the real world, using a real autonomous system.

5.1.2.2.1 HighwayEnv

Figure 5.3, shows the HighwayEnv [214] environment. HighwayEnv is a minimalistic open-source simulator used to explore control and navigation aspects of autonomous driving. The ego vehicle uses an onboard sensor to track the position and velocity of the closest traffic vehicles and a rule-based navigation module to traverse the highway. We configured the scenario by placing the ego vehicle at one end of a highway, and the goal was for the ego vehicle to navigate down the highway as fast as possible. The highway was populated with between 1 and 10 vehicles placed randomly in

front of the ego vehicle. The first traffic vehicle was spawned in a random lane 15m in front of the ego vehicle. Subsequent vehicles were spawned in random lanes using 2m intervals.

The 10 traffic vehicles were allowed to operate between speeds of $15m/s - 25m/s$, while the ego vehicle speeds were $15m/s - 30m/s$. Each run lasts 25 seconds, allowing the ego vehicle to overtake the other vehicles. If the ego vehicle collides with another vehicle, the simulator removes that vehicle and reduces the velocity of the ego vehicle. By allowing and recording multiple collisions during each test, we could ensure that all tests were precisely 25 seconds. We generated 1,000,000 tests to explore the possible scenarios that could occur in HighwayEnv. Of the 1,000,000 tests, 94% executed without any failures.

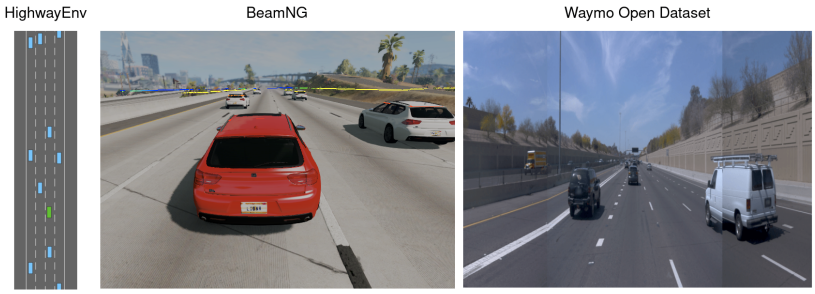


Figure 5.3: HighwayEnv [214], BeamNG [33], and the Waymo Open Perception Dataset [333] environments.

5.1.2.2.2 BeamNG

The second environment, shown in Figure 5.3, is BeamNG.tech [33] (BeamNG), a versatile high-fidelity vehicle simulator with state-of-the-art soft-body physics, collision detection, and sensors. The ego vehicle is controlled by BeamNG's autonomous driver module, which has approximately 2500 lines of code to perform speed and steering modulation, obstacle avoidance, and path planning. We spawn between 1-10 traffic vehicles at random locations in a $60m \times 20m$ rectangle centered $30m$ ahead of the ego vehicle. The vehicles had a speed limit of $120km/h$, while the ego vehicle was $144km/h$ (to allow the ego-vehicle to overtake the traffic vehicles). The traffic vehicles are controlled by BeamNG's traffic module which maintains a traffic density, so when a traffic vehicle is overtaken and thus no longer in the field of view of the ego vehicle, it respawns the vehicle in front but out of sight of the ego vehicle. Each run was configured to last 50 seconds. If a failure occurred during the run, we let the autonomous vehicle software try and recover for the rest of the test. BeamNG can

not run faster than real-time, and it has license restrictions for running multiple instances, which limited the number of runs to 10,000 tests, which took roughly a week of continuous execution on a high-end machine. The ego vehicle achieved an 87.39% success rate over all tests. Although the success rate of both simulated systems is lower than an ideal safety-critical system, it is fitting for our study as it allows us to evaluate the failure detection capabilities of *PhysCov*.

5.1.2.2.3 Waymo Open Perception Dataset

Our third environment is the real world as perceived by the Waymo vehicle (Waymo Open Perception Dataset [333]). Using this real-world dataset allowed us to evaluate *PhysCov*'s applicability to real-world systems, but we note that it has no recorded failures and limited scenarios. So our focus is on the ability of *PhysCov* to form equivalent input classes. Our *PhysCov* pipeline uses sensor data from the mid-range LiDAR on top of the vehicle and four short-range LiDARs (front, side left, side right, and rear). We also use data from the 3 front-facing cameras to understand and explain each scenario. We selected all 798 scenarios from the training set. Each scenario contains a 20-second snippet of autonomous vehicle driving, giving us a total of $15,960$ seconds of real-world driving data captured at 10Hz, which results in 159,600 RRS signatures generated over the entire dataset.

5.1.2.3 Baseline Techniques

We consider several techniques mentioned in the related work: code covered, miles driven, scenario coverage, and trajectory coverage. Miles driven was discarded as all tests drive for a similar time and distance, so it provided no valuable information. Scenario coverage was also dismissed as it requires full knowledge of the entire environment and all relationships between objects in the environment, which is not feasible given the complexity of the BeamNG or Waymo tests. Additionally, it is impractical for long and large numbers of tests, given the amount of data it needs to track. Thus, we ended up employing code and trajectory coverage.

5.1.2.3.1 Code Coverage

Since the control software for Highway-Env is written in Python, we used the existing "Coverage.py" tool to compute both the line and branch coverage [29]. BeamNG's autonomous control software is written in Lua, and coverage tools for Lua are still quite limited (e.g., LuaCov [290]). Therefore, we implemented our own tool to track line, branch, and intraprocedural prime path coverage[1], intraprocedural path coverage, and path coverage. The source code for Waymo's autonomous vehicle is not publicly available; thus, no code coverage was computed for Waymo.

5.1.2.3.2 Trajectory Coverage

Trajectory coverage measures the extent to which an autonomous vehicle covers discrete regions on a road [153]. The measure requires users to define a driving area, and the original work assumes a rectangular bounded area to facilitate its specification. Next, the driving area is divided into equally sized blocks. We set the block size to $1m \times 1m$, matching the original paper. Each time the autonomous vehicle drives over one of the blocks, it is marked as covered. Trajectory coverage is computed as the set of blocks covered over the total number of blocks in the driving area. As defined, the approach is "naive" in that assuming most scenarios will consist of rectangular roads when most areas actually consist of irregular shapes (e.g., a curve on the road). As part of our study, we implement a version that supports irregular shapes that precisely match the curves of the road area while also ignoring portions of the road that should not be covered, for example, lanes with traffic in the opposite direction. We call this approach "improved" trajectory coverage.

5.1.2.4 Evaluation Criteria

To evaluate *PhysCov*, we primarily considered two criteria: the consistency of equivalent classes and failures as it provides a concrete way to judge test's behaviors.

[1] In a prime path, each node cannot appear more than once, and it is not a subpath of any other prime path.

5.1.2.4.1 Equivalent Classes and Inconsistencies

A desirable coverage abstraction will produce the smallest number of equivalent classes, where all the inputs in each class lead to the same behavior. Thus, we evaluate the coverage metrics in terms of the number of equivalence classes they render and the consistency displayed by the rendered classes. For example, for our proposed measure Ψ, two tests that have the same Ψ are said to belong to the same equivalence class and thus should generate the same behavior. For lines of code coverage, two tests that exercise the same lines are said to belong to the same equivalence class and should behave consistently. Similarly, two tests are equivalent for trajectory coverage if they cover the same blocks in the drivable area. We judge an equivalent class containing tests that pass and fail to be inconsistent, while classes that contain just passing or just failing tests to be consistent.

5.1.2.4.2 Failures

There are no failures in the Waymo Perception Open Dataset; thus, this metric was not computed for Waymo. For the simulation environments, we count the number of failures and the number of unique failures as they represent a refinement of the considered exposed behaviors. We define failures as either a crash or a stall. A crash occurs when the ego vehicle collides with any obstacle, while a stall occurs when the ego vehicle comes to a complete stop, even though there is a way to keep moving forward. During each crash, we record the velocity of the traffic vehicle, the velocity of the ego vehicle, and the angle of incident. We then define a unique crash as one whose velocities and angle of the incident match within a threshold of $1m/s$ and 1 degree, respectively. Given two crashes that fall inside this threshold, we argue that there was only one unique crash, as the circumstances around the crash must have been extremely similar to result in the same velocities and angle of incidence. To detect a stall, we determine if the vehicle has a velocity of less than $0.01m/s$ and if the vehicle has an obvious way to move forward. We define having a way forward as there being a 30-degree gap in front of the ego vehicle, with no obstacles. We categorized stalls based on the distance and angle to the closest object. This distance angle pair could then be compared to other stalls to see if the stall happened under similar conditions and thus used to identify unique stalls.

5.1.2.5 PhysCov Implementation

Implementing the *PhysCov* pipeline consists of first collecting the state and sensor data from each vehicle, followed by the three major steps described in the approach: reduction to reachable set, reduction to sensed reachable set, and vectorization. The implementation was made publicly available [140].

5.1.2.5.1 State and Sensor Collection

To account for differences between each of the environments' state and sensor formats, we convert all data into a standard trace format that contains the current time, position, velocity, heading, crash status, stall status, and a 2D point cloud of all detected obstacles around the vehicle in the vehicle's frame of reference. The three environments we study provide the ego vehicle's time, position, velocity, and heading. Since there are no crashes or stalls in the Waymo environment, these cells were set to False. HighwayEnv tracks the crash status of vehicles internally, so we modified it to externalize it. BeamNG reports precise vehicle damage that we simplified as a crash if any damage was reported. Stalls were detected by checking when the ego vehicle's velocity was less than $0.01m/s$, and had a 30-degree gap in front with no obstacles.

To capture the 2D point cloud, we used different approaches for each environment. For HighwayEnv, the only obstacles are the traffic vehicles and the road's edge. Since we know the exact size of the vehicles and the road's edge is straight, we can geometrically compute a 2D point cloud of all objects in the ego vehicle's frame of reference. BeamNG lets us equip the vehicle with a LiDAR that returns a point cloud. We configured the LiDAR of the ego car similar to those in commercial vehicles [358]. To focus the LiDAR on obstacles ahead of the vehicle in a 2D plane, we configured it to return a 180-degrees arc with a 0.1-degree range on the z-axis. Note that in BeamNG, readings are returned with some environmental noise since the LiDAR follows the vehicle's pitch and roll as it throttles, brakes, or hits bumps in the road. Since BeamNG reports the point clouds in the global frame, we convert it to the ego frame. The Waymo dataset includes 5 cameras and 5 LiDARS, where each LiDAR generates a 3D point cloud in the ego frame of reference, with points

up to 75 meters away. We combined each of the individual point clouds into a single high-fidelity point cloud, and then we removed all LiDAR points behind the ego vehicle, as these points could not affect the approximated reachable set, leaving only points within a 180-degree arc in front of the vehicle. Finally, we flatten the LiDAR to generate a 2D cloud to points between $0.75m$ above the ground and below $1.25m$ with respect to the ego vehicle, as these are obstacles with which the vehicle might collide.

5.1.2.5.2 Reachable Set Computation

To compute a reachable set, we need the vehicle's state, including the initial position, linear velocity, and angular velocity. Each of the ego vehicles returns the position and linear velocity. We approximated the angular velocity as 0, using the assumption that the majority of scenarios contain roads without sharp turns and the ego vehicle moving forward, therefore the magnitudes of angular velocity should always be extremely small. Under this assumption, we can efficiently approximate the reachable set as a sector, as shown in Figure 5.2b. The sector's origin was set to the position of the ego vehicle.

The sector's line of symmetry was set to match the direction of travel of the ego vehicle. At a given time t, the sector's radii were computed based on a user-defined time horizon multiplied by the vehicle's maximum speed. When the vehicle is not traveling at maximum speed, the sector results in an over-approximation of the actual reachable space. This is acceptable in that we would rather include additional spaces than ignore potentially dangerous obstacles. The maximum velocity of the ego vehicle for HighwayEnv was $110km/h$. Using $v = d \times t$, and a timestep of 1 second, we can compute that the sector's radii should be $30m$. Similarly, the maximum speed for BeamNG and the Waymo vehicle was set to $144km/h$. Again using a time step of 1 second, we can compute that the sector's radii should be $40m$.

Finally, the sector arc was set to match the maximum steering angle of the vehicle. The maximum steering angle for HighwayEnv was 30 degrees, resulting in a 60-degree arc. For BeamNG and Waymo, we selected a generic Audi vehicle with a maximum steering angle of 33 degrees [152] (66-degree arc). While this sector-based approximation could be more precise by accounting, for

example, for changes to the angular velocity, we favor its application because it provides a safe over-approximation, is applicable across the three environments, and its efficient to compute, which is key given the datasets' sizes.

5.1.2.5.3 Sensed Reachable Set Computation and Vectorization

Our implementation combines these two steps into a single computation. It requires a user-defined point inflation size δ, total vectors n, spread $\mathcal{D}$, and tick intervals $\mathcal{I}$. It starts by converting each point in the cloud to a circle centered on the point with a radius δ, which we set to $0.2m$ (we empirically found this value to reduce the chances that an object goes undetected while also avoiding obstructing other objects). Next, the implementation creates n vectors, which stem outward from the ego vehicle, based on D, which defines how we spread the vectors out. It then computes the length of each vector from the origin of the ego vehicle to the first intersection, which is either the edge of the reachable set or one of the points from the point cloud. The real values representing the length of each vector are then rounded as per I. These computations rely on the "shapely" python package, which offers functionality to manipulate and analyze planar geometric objects [117].

Below we give detail on how each of the parameters was defined.

Total vectors (n): We explore n between 1 and 10 to explore its impact on the signatures generated. When $n = 1$, denoted as (Ψ_1), only a single vector was used to approximate r_t^{sen}, resulting in a RRS signature with a single magnitude. When $n = 10$, denoted as (Ψ_{10}), the RRS signature included 10 magnitudes.

Spread $\mathcal{D}$: We assumed that the region directly in front of the vehicle was the most important. Therefore we designed $\mathcal{D}$ to favor the center of the reachable set. The approach places vectors at roughly 6-degree intervals from the centerline. For example, when $n = 1$, a vector was placed on the centerline. When $n = 2$, two vectors were placed at ± 6 degrees. When $n = 3$, a vector was placed on the centerline, and two vectors were placed at ± 12 degrees, etc.

Tick Intervals $\mathcal{I}$: RQ1 and RQ2 used failures as part of the evaluation criteria, and since crashes occur in the region closest to the vehicle, we set $\mathcal{I} = \{5m, 10m\}$ from the vehicle. RQ3 was applied to a real dataset without failures and focused on categorizing and comparing real-world scenarios.

Therefore we extended the resolution to $\mathcal{I} = \{5m, 15m, 25m, 35m\}$ along each vector.

5.1.2.6 RQ1: Ψ Effectiveness

This research question explores how effective *PhysCov* is at generating equivalent input classes. Tables 5.1 and 5.2 show, each baseline and *PhysCov* for HighwayEnv and BeamNG. Specifically, they show the number of classes generated, and for the classes with more than one test, the number of consistent classes (containing only passing or only failing tests), inconsistent classes (containing both failing and passing tests), the average number of tests per class, and the percentage of inconsistent classes.

First, we consider the baseline metrics. Line coverage groups tests into 2754 and 151 classes for HighwayEnv and BeamNG, respectively. Among the ones with multiple tests, 75% and 65% are inconsistent. Branch coverage groups tests into 7097 and 146 classes, reducing the inconsistency rate to 63% and 58% when compared to line coverage. The more complex code coverage measures in BeamNG do not fare any better. Intraprocedural prime path coverage produces more equivalence classes than both line and branch coverage. However, the number of inconsistent classes actually jumps from 64 and 56 to 113. The more exhaustive intraprocedural path coverage and path coverage are overly specific, producing a unique signature for each test, suggesting an inability to group any tests. Similar to the complex code coverage measures, trajectory coverage generates 650,123 and 10,000 signatures.

Next, we consider the different parameters of Ψ. As expected, as the total number of vectors increases, so does the number of equivalent classes generated while the percentage of inconsistent classes decreases. This highlights the ability of Ψ to vary the granularity of analysis. For example, Ψ_1 results in 2283 and 450 equivalent classes with multiple tests for HighwayEnv and BeamNG, respectively. Of these classes, 55% and 57% are inconsistent. As we increase the approximation granularity, the number of consistent classes with multiple tests increases, while the number of inconsistent classes with multiple tests decreases. Our most detailed approximation, Ψ_{10}, results in 9004 equivalent classes with multiple tests for HighwayEnv, while staying roughly consistent at 440 for BeamNG, 18% and 32% being inconsistent. This highlights our metric's ability to scale its

Table 5.1: Equivalent classes across metrics for HighwayEnv

	All classes	Only considering classes with more than 1 test			
Cov Metric	**Equiv classes**	**Equiv classes**	**Inconsistent classes**	**Avg # tests in classes**	**Percentage inconsistent classes**
Line	2754	2241	1672	446.0	75%
Branch	7097	4589	2889	217.4	63%
Traj	650123	41717	8155	9.4	20%
Ψ_1	3335	2283	1251	437.6	55%
Ψ_5	4096	1887	501	528.8	27%
Ψ_{10}	41443	9004	1640	107.5	18%

Table 5.2: Equivalent classes across metrics for BeamNG

	All classes	Only considering classes with more than 1 test			
Cov Metric	**Equiv classes**	**Equiv classes**	**Inconsistent classes**	**Avg # tests in classes**	**Percentage inconsistent classes**
Line	151	99	64	100.5	65 %
Branch	146	97	56	102.6	58 %
I Prime Path	421	151	113	64.4	75 %
I Path	10000	0	0	0	—
A Path	10000	0	0	0	—
Traj	10000	0	0	0	—
Ψ_1	682	450	258	21.7	57 %
Ψ_5	1594	330	132	26.5	40 %
Ψ_{10}	3628	440	139	15.5	32 %

abstraction granularity while also creating more consistent equivalent classes with multiple tests.

To compare the baseline metrics versus *PhysCov*, we can identify the Ψ_x where the choice of x helps to render the number of equivalent classes observed in the baseline coverage measure. In HighwayEnv, branch coverage groups tests into 7097 equivalent classes. Ψ_5 is the closest, grouping tests into 4096 classes. Ψ_5 has 27% of inconsistent classes, less than half of branch coverage. Trajectory coverage generated $650,123$ equivalent classes, which is 15 times more classes than our most specific metric Ψ_{10}. One might argue that trajectory coverage performs well since it generates $41,717$ classes with multiple tests with 20% inconsistency, while Ψ_{10} only generates 9004 with 18%. However, a class generated by trajectory coverage has, on average, 9 tests while Ψ_{10} classes have, on average, 107 tests. This indicates that trajectory coverage is generating overly specific classes that add no value compared with Ψ_{10}. For BeamNG, establishing a fair comparison against code coverage metrics is more difficult as even Ψ_1 renders more classes. However, to capture the more complex environments Ψ_5 is sufficient to reduce the inconsistency rate to 40%, and with Ψ_{10} the inconsistency rate is 32%, half that of branch coverage. The results for trajectory coverage align

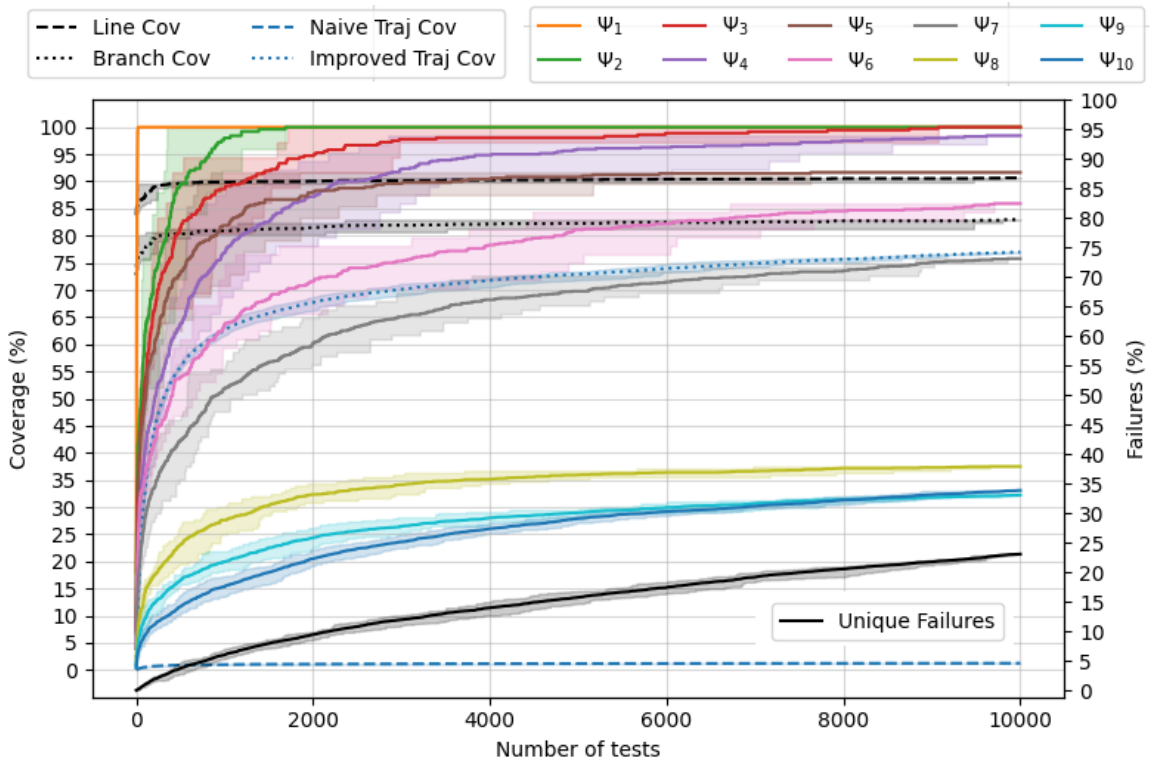


Figure 5.4: *PhysCov* for tests in BeamNG

with those from Highway-Env. It generates overly specific classes, in fact, so specific that no tests were grouped together.

Next, we examine *PhysCov* as the size of the test suite size increases through Figure 5.4 for BeamNG (similar trends can be observed for HighwayEnv, and we share those results in our artifact [140]). The shaded regions show the minimum and maximum coverage for the test suite size. To generate these regions, we computed each line 10 times while randomly varying the order in which tests were added to the test suite. These figures show three trends. First, code coverage measures, naive trajectory coverage, and Ψ_1 saturate within the first 50 tests, while Ψ_2 saturates within the first 1000 tests. These metrics suffer as their resolution is too limited to be helpful as adequacy metrics. Second, when x in Ψ_x increases, the coverage achieved grows rapidly before starting to level off. This is because, as time passes, tests conducted on the same scenario struggle to reveal

new coverage. The improved trajectory coverage is comparable to Ψ_7. While this is promising, one concern is that if we were to add a new scenario, for example, a similar highway in another city, trajectory coverage would require a second derivable area to be defined, and show a sudden vertical drop in coverage as the denominator would have doubled (assuming the new derivable area is the same size as the old), while *PhysCov*'s denominator would not change. This would happen for any scenario, regardless of its similarity or difference. This thought experiment indicates another shortcoming of trajectory coverage, which would either need to know all possible scenarios which may be covered beforehand (so that a static denominator could be computed), or each time a new scenario was added, the denominator would change, and the coverage achieved would drop. The third and final trend is that similar to the higher Ψ, the unique number of failures also levels off with a greater number of tests as new faults become more difficult to expose. Next, we explore whether the correlation between failures and coverage supports this observation.

Since the correlation between unique failures detected and coverage is expected to temper as a metric saturates, we explore this relation over small suites consisting of 10, 50, 100, 500, 1000, and 5000 tests. We generate 1000 suites of each size and compute the correlation between the test suites coverage and the unique failures detected. The resulting Pearson correlation coefficients are shown in Tables 5.3 and 5.4. These tables show a stark contrast between structural code coverage and *PhysCov*. There is almost no correlation between the structural code coverage metrics and unique failures found. When looking at trajectory coverage, there appears to be a moderate correlation between trajectory coverage and failures, with increases for larger suites. Analyzing this increase in correlation revealed that the definition of unique failures artificially favored this metric as crashes in identical circumstances (e.g., velocity and angle of collision, number of vehicles, and obstacles in the vicinity) occurring in different sections of the track were independently counted. This indicates that this metric may be complementary to Ψ and that Ψ parameters may need to be adjusted based on the failure type. Still, Ψ_{10} correlation is greater for 11 of the 12 suites combinations looked at in this study. Finally, when we consider *PhysCov*, there is also a modest correlation between failures and *PhysCov*, and the strength of the correlation varies across two factors. First, using a higher resolution *RRS* abstraction always produces test suites with a higher correlation with failures. Second, and as

Table 5.3: Correlation between coverage and unique failures found for test suites of different sizes in HighwayEnv.

Test Suite Size	Line Coverage	Branch Coverage	Naive Trajectory Coverage	Improved Trajectory Coverage	PhysCov (Ψ_5)	PhysCov (Ψ_{10})
10	0.09	0.10	0.02	—	0.63	0.69
50	0.00	0.02	0.05	—	0.55	0.68
100	0.05	0.05	0.20	—	0.43	0.64
500	-0.02	-0.02	0.23	—	0.21	0.47
1000	nan	-0.02	0.22	—	0.20	0.37
5000	nan	0.08	0.09	—	0.05	0.32

Table 5.4: Correlation between coverage and unique failures found for test suites of different sizes in BeamNG.

Test Suite Size	Line Coverage	Branch Coverage	Naive Trajectory Coverage	Improved Trajectory Coverage	PhysCov (Ψ_5)	PhysCov (Ψ_{10})
10	0.05	0.05	-0.22	-0.22	0.04	0.50
50	0.04	0.05	-0.05	-0.05	0.39	0.49
100	-0.04	-0.03	0.03	0.03	0.27	0.43
500	-0.02	-0.04	0.21	0.21	0.18	0.29
1000	0.07	0.09	0.25	0.25	0.11	0.20
5000	0.07	0.08	0.20	0.20	0.07	0.19

expected, increasing the number of tests using the same scenario weakens the correlation as coverage starts to saturate. Moving to Ψ_x where $x > 10$ is likely to mitigate this.

RQ1 Finding: We find that our metric can scale its abstraction granularity while also creating more consistent equivalent classes than our baselines. Specifically, *PhysCov* outperforms traditional code coverage metrics in terms of the percentage of inconsistent classes, where in both cases, our worst inconsistencies are better than the best instances of traditional code coverage. Furthermore, we find that trajectory coverage achieves lower inconsistency in some cases by creating significantly more equivalent classes than our approach. Despite this, can scale *PhysCov* to create more consistent classes using significantly fewer abstractions (up to 15 times fewer). We also find that our approach scales better than the baselines and is more general in that it can account for new scenarios without significant changes. Finally, we show that *PhysCov* correlates with failures in 11 of the 12 cases examined in this study.

5.1.2.7 RQ2: Test Selection using PhysCov

This research question explores how effective *PhysCov* is as a test selection metric and its ability to select test suites that induce unique failures. The previous research question suggested that *PhysCov*

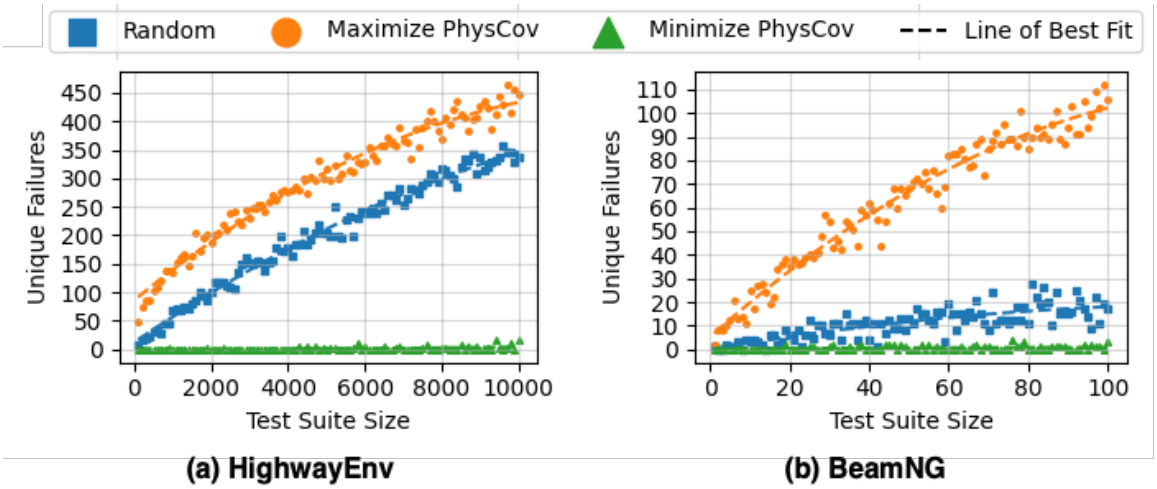


Figure 5.5: Unique failures found when selecting test suites that maximize or minimize *PhysCov*

did indeed correlate with unique failures. If this were true, we should be able to select test suites that maximize *PhysCov*, which in turn would maximize the unique failures found. We generated 100 test suites between 0 and 1% of the original test suite size by repeatedly randomly sampling 100 tests from the original test suite and greedily adding the test that either maximizes or minimizes the *PhysCov* of the current test suite.

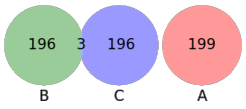
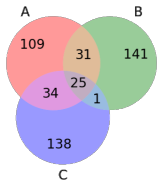
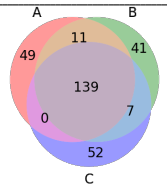
Figure 5.5 shows the unique failures found by each of the 100 test suites for both HighwayEnv and BeamNG. Each figure shows test suites that were selected to maximize *PhysCov*, minimize *PhysCov*, or selected randomly. These figures reveal two insights. First, test suites selected to maximize *PhysCov* always detect more unique failures than randomly selected suites. Second, minimizing *PhysCov* always produces test suites with fewer, generally none, unique failures. This shows that *PhysCov* is a good metric for selecting tests and reducing a test suite size. Once again, there is slightly more variance in the results of BeamNG, likely due to its complexity and noisier environment. Overall these results indicate that *PhysCov* is a viable metric for test selection. We could use *PhysCov* to select a minimal number of tests while retaining the number of unique behaviors and, in turn, failures found.

RQ2 Finding: Our findings show that *PhysCov* is a viable metric for test selection. Specifically, test suites selected to maximize *PhysCov* always detect more unique failures than those randomly selected, while minimizing *PhysCov* always results in tests with fewer, if any, unique failures.

5.1.2.8 RQ3: Real-World Scenarios

This research question explores how effective *PhysCov* is on a real dataset. As configured, this dataset provided 3.1% coverage using Ψ_{10}, so this dataset is clearly missing many potential environment-state pairs that the vehicle will encounter.

Table 5.5: Comparing overlap between RRS when selecting based on Scenarios and RRS signatures.

Selection Method	Distinct	Similar
Given Scenarios		
Resultant RRS		
Given RRS		
Resultant Scenarios		

Beyond this simple characterization, this study aims to assess *PhysCov* potential at distinguishing between similar and different real-world scenarios. The study explores this problem from two angles. First, using the camera images, we identified 3 tests where the vehicle was operating in clearly distinct scenarios (parking lot, two-lane rural road, single-lane urban road), and 3 tests where the vehicle was operating in very similar environments (variations of highways). These are shown in the first row of

Table 5.5. We conjecture that if *PhysCov* was an effective metric at identifying equivalent classes, the 3 distinct scenarios should also produce distinct RRS signatures, while 3 similar scenarios should produce more similar RRS signatures. The results in the second row of Table 5.5 support this conjecture. Each test is roughly 20 seconds, with data recorded at 10Hz, resulting in 199 RRS vectors. Interestingly we see that the distinct scenarios B and C have 3 overlapping RRS signatures, while scenario C has no overlap. This makes sense when considering that despite the differences of scenario B (single-lane urban road) and scenario C (dual-lane rural road), they both are narrow roads, so there is a chance of some overlap in RRS. However, scenario A is a parking lot that is significantly different from both other scenarios and thus has no RRS overlap.

In the second part of this study, we selected 3 tests that produced the least and 3 that produced the most overlap in terms of RRS signatures. To do this, we compared all 3-way combinations of Waymo's 798 tests, a total of 84,376,796 combinations, and then selected the least and most overlapped combination, as shown in the third row of Table 5.5. The Venn diagram indicates that distinct scenarios have no overlap, while similar scenarios produce nearly 139 identical RRS signatures. We conjecture that if *PhysCov* was a good metric, the tests with no overlap should intuitively be required to perform distinct behaviors, while tests with overlap should require similar behaviors. The fifth row in Table 5.5 shows camera data from the selected tests. When selecting distinct RRS vectors, we ended up with 3 distinct scenarios: a busy intersection, a one-way downtown city road, and a two-lane road with a separator. The scenarios with similar RRS signatures corresponded to three scenarios where the Waymo vehicle was stuck in dense traffic. Interestingly two of these scenarios (A and B) stem from the same root test, even though Waymo provided them as separate tests.

RQ3 Finding: We demonstrate that *PhysCov* can be applied to real datasets. Using *PhysCov*, we find that the Waymo Open Perception Dataset provides only 3.1% coverage using Ψ_{10}, indicating that it is missing many potential environment-state pairs. We also show how *PhysCov* can distinguish between different and similar scenarios, which is helpful in deciding how best to optimize and augment a test suite.

5.2 Conclusion

This work introduces a general approach to quantify the number of unique *physical environments* and *physical state* pairs experienced by an autonomous vehicle. It relies on a novel abstraction of the sensed environments, RRS, that employs physical reachability analysis based on the vehicle state and kinematics and dynamics to identify the most relevant area of the input space and efficiently produces a vector-based characterization of that space. Our study illustrates how Ψ can render meaningful equivalent classes to capture the environments, its correlation with failures, and how the RRS parameters can control the quality and cost of *PhysCov*. The study also shows the potential of Ψ to improve the efficiency of the testing process through test selection and distinguish among different real-world scenarios.

Chapter 6

Conclusion and Future Work

Autonomous systems are becoming increasingly common in everyday life, driven by advancements in both software and hardware components. Ensuring the safe operation of these systems, particularly as they begin to interact more frequently with humans, is crucial for their widespread acceptance in society. This thesis argues that rigorous testing is essential to ensure their safety and to detect incorrect behaviors before their deployment. Notably, while the software driving autonomous system behaviors grows increasingly complex, we specifically highlight that they operate within complex *physical environments* and interact with these environments through hardware possessing inherent *physical semantics*. Therefore, while it seems intuitive to build on decades of research in software validation to test these autonomous systems, such validation techniques must evolve, or be completely reinvented, to adequately account for the unique physical attributes driving behaviors.

This thesis enhances the traditional testing pipeline. This pipeline starts with test generation, executes the tests, and then computes test adequacy, ensuring that the system was thoroughly tested. Each introduced approach within the testing pipeline was specifically designed to account for either an autonomous system's physical environments, physical semantics, or both.

More specifically, within test generation, we introduce two approaches. The first is feasible and stressful trajectory generation, which is designed to create tests that account for the physical semantics of autonomous vehicles [138]. This approach constructs trajectories that are valid by

construction with respect to an autonomous vehicle's physical semantics. Our approach utilizes a modified version of the Probabilistic Roadmap (PRM) that incorporates the kinematic and dynamic models of the autonomous system to generate trajectories that are valid by construction. It then filters trajectories using a parameterizable scoring model, which selects the ones most likely to maximize a given stress metric. Finally, we provide a study showcasing how this technique generated tests with more valid trajectories than baselines, and resulted in significant increases in maximum deviation—up to 55.9% and 41.3% for the two autonomous systems studied.

The second test generation approach we introduce is differential testing on existing data[1]. The premise of this approach is that it allows exploration of large portions of the physical environment through the vast collection of already collected sensor data. Our method processes this existing sensor data, identifying the subset likely to result in failures and relevant to a specific autonomous system. We implement this approach in two stages. First, we introduce a differential testing technique that compares behaviors of multiple autonomous systems. Specifically, this component identifies sensor input that produces behavioral differences across these systems, indicating potential failures. Second, we introduce a filtering stage, designed to exclude sensor input not relevant to a specific autonomous system. This component uses both sensor data and an autonomous system's Operational Design Domain (ODD) to generate a compliance vector that indicates which ODD semantic dimensions were met by the sensor input. We demonstrate the application of this approach using 3 versions of a commercial autonomous vehicle on video data containing over 4.6 million sensor input. Our findings indicate a substantial number of sensor inputs that produce behavioral differences indicative of failures. Specifically, we found that 9.8% of all sensor inputs resulted in steering differences of 10 degrees or more between the 3 autonomous systems, with 11,769 instances causing steering differences of 100 degrees or more. We also show how using external data collected independently of the autonomous systems can increase the number of detected failures by up to 569%. Finally, we demonstrate that our filtering can achieve semantic accuracies of up to 94.2%, highlighting several examples of sensor inputs that are relevant with respect to the ODD and producing significant steering difference.

[1]Automated ODD checking portion under submission

Next, we examine test execution, where we introduce two approaches. The first approach addresses ways to overcome the inherent differences between simulated and real-world environments. These differences are collectively known as the simulation-reality gap and is problematic for testers, as it means that behaviors produced in simulation may not be possible in the real world, and behaviors in the real world may be impossible to replicate in simulation, potentially leading to failure-inducing behaviors being incorrectly flagged or going unnoticed. To overcome this, we present World-In-the-Loop simulation [135], an approach that integrates both real and simulated sensor data to allow autonomous systems to operate in a new mixed reality. Our approach allows developers to vary the extent of simulation and reality combined when forming this mixed reality, enabling developers to gradually scale their testing from simulation to reality. We present a study that showcases how, through this gradual increase in realism, failures that are present only in the real world and cannot be detected in pure simulation are detectable using the mixed-reality environment produced by World-In-the-Loop simulation.

The second test execution approach investigates ways to explore the physical semantics of autonomous systems in the real world. The premise of this work is that autonomous systems frequently encounter external forces, necessitating testing under such conditions. However, this process often requires specialized equipment, is costly, and depends on specific real-world conditions. In this work, we focus on drones, such as quadrotors, which must handle forces from carrying objects, contending with wind, managing sudden force reductions from dropping objects, and absorbing impacts during grabbing or catching activities. We introduce an approach capable of replicating a range of forces on a drone without needing physical connections to external sources, allowing for a much wider range of tests to be executed, more quickly and more cheaply. Specifically, we introduce a generalizable haptic suit [143], which attaches directly to a drone. This suit features additional propellers on arms that extend outward and can rotate around the arm's axis to exert extra forces on the drone. We also provide a force-to-control module and a controller, both capable of translating any force into actuation commands for the haptic suit. We demonstrate the suit's ability to replicate four indoor real-world test scenarios and one outdoor scenario. Additionally, we identify a case where a failure occurring in a real-world test is nearly perfectly replicated using the haptic suit. Overall, the haptic

suit offers a promising method for testing a large volume of varied real-world scenarios without the need for complex setups or dependent on specific outdoor conditions.

The final chapter presents a technique for measuring the test adequacy of an autonomous vehicle. The premise of this work is that the behaviors of autonomous vehicles are significantly driven by a system's physical environment and physical semantics. Traditional coverage metrics, which do not take these factors into account, correlate poorly with behavior when applied to autonomous systems—a finding we demonstrate in a later study. Therefore, we present an approach called Physical Coverage [140], which uses kinematic and dynamic models along with reachability analysis to account for a vehicle's physical semantics. This approach combines these elements with the perceived physical environment of the autonomous systems. The resulting metric is approximated using a geometric vectorization technique to provide an abstraction of both the physical environment and physical semantics of the system throughout a test. Our study shows how this coverage metric correlates more closely with behavior compared to other state-of-the-art techniques at the time and significantly improves over traditional coverage metrics when applied to autonomous systems. We also showcase use cases of our approach, for example, to perform test suite reduction or to apply scenario selection on real autonomous system data from Waymo.

6.1 Broader Impacts

Traditional software systems have revolutionized the world we know today. However, before these systems could be deployed in the real world, especially in safety-critical scenarios or where the cost of failure was high, they required rigorous validation to ensure safe and correct operation. This necessity was a major driving factor behind what is now known as traditional software testing and engineering. Similarly, autonomous systems today hold the potential to revolutionize several industries, including transportation, manufacturing, space exploration, environmental monitoring and protection, and healthcare, to name just a few. However, to realize this potential and integrate autonomous systems into our everyday lives, it is imperative that they operate safely and as expected. This dissertation explores methods to adapt traditional software validation techniques, which have proven successful

in ensuring the safety and reliability of software systems in the past. We combine these proven methods with innovative new ideas and approaches tailored to address the unique challenges posed by autonomous systems. By developing effective validation strategies sharing repositories of the tools and data [139, 136, 144, 142, 141], we help pave the way for the integration of these systems into the real world, bringing us one step closer to the future, while also enhancing our ability to ensure their safe and correct operation.

6.2 Future Work

There are numerous avenues for future research across the three areas of the testing pipeline outlined in this thesis. Beyond refining and enhancing the current methodologies, a significant direction for future work involves adapting and applying the approaches presented to a more diverse set of autonomous systems. This adaptation process, while seemingly straightforward, often demands innovative changes to make the work more generalized than it already is. For instance, transitioning the testing techniques from ground vehicles, which primarily operate in a 2D space, to aerial vehicles such as drones, requires updating techniques to accommodate a third spatial dimension. Similarly, expanding these techniques from individual vehicles to swarms introduces complexities such as increased cost and the potential for new behavioral dynamics driven by different vehicle interactions. Below we look at more specific areas of potential future work.

6.2.1 Test Generation

There are several promising research directions for the approaches presented in test generation. Firstly, for feasible and stressful trajectory generation, an immediate step could involve incorporating more advanced kinematic models to enhance the accuracy of predictions regarding autonomous vehicle dynamics under various stress metrics. A more substantial advancement would be to integrate physical environments into the trajectory generation process. Currently, trajectories are generated as if in an empty world. Including richer physical environments could improve this approach in two significant ways. Firstly, by generating trajectories that account for environmental

constraints, it could become applicable to a wider range of developers who do not have access to large open spaces. This means testers could configure the technique to only generate trajectories around obstacles present in their testing environment. Secondly, and potentially more intriguing, is designing environments that use the test trajectories as input. Here, the goal for the autonomous system would not simply be to follow a trajectory but to navigate a physical environment specifically constructed to require the autonomous system to execute the given trajectory. Another potential research direction is to develop trajectories for dynamically changing environments. This could introduce a new layer of complexity, where part of the stress comes from navigating environments that change over time, such as a window that closes, which the autonomous system must reach, adding additional constraints to its operation.

The next step for differential testing on existing data would be to refine the oracle. Currently, the oracle primarily identifies differences based on magnitude and frequency thresholds. However, potential failures may be more complex and could require additional mechanisms for detection. These might include analysis of behaviors over extended periods, where failures are detected based on statistical variance. Another idea is to design an oracle that uses the input scene-state pair, allowing it to track the invariance of the systems with respect to each autonomous system and the input, thus enabling it to vary detection thresholds based on the type of input it currently experiences. For example, divergence in behaviors at intersections is much less concerning than divergence on straight, empty roads. Another promising direction for future research involves a more thorough examination of how the state influences the behavior of the system. This would not only allow us to quantify the contribution of state and environment to failures but also enhance the technique's ability to precisely determine the root cause of failure. Building on the principles of metamorphic testing [313], where one system is given inputs that are semantically similar but slightly different, our approach could vary both the physical environment and state upon identifying a failure. This adjustment could help precisely identify the range of environments and states that lead to problematic behavior. Similarly, while this method could be used to identify ranges of inputs that result in failure, it could also be used to identify those that the system can handle effectively. Finally, there is also room to explore LLM's ability to further classify scenarios. This includes using

LLMs to classify other sensor data such as LiDAR or radar, training from scratch or fine-tuning LLMs to specifically classify sensor data semantics, or exploring LLMs' ability to handle significantly more complex ODDs.

6.2.2 Test Execution

There are several promising directions for future work in test execution. First, let's consider World-In-the-Loop simulation in isolation. One particularly interesting area of future research lies in the context of swarms. Autonomous systems in swarms not only suffer from the same simulation-reality gap as individual systems, but are also notoriously expensive to test in the real-world. This expense arises due to the need for multiple autonomous systems and extensive real-world infrastructure. Consequently, real-world testing often occurs only after significant investment in the project, making any failures particularly costly. World-In-the-Loop simulation could be a potential avenue for reducing these costs. By mixing simulated and physical environments and swarms, we could test a combination of autonomous systems, with some operating in the real-world and others in simulation. This approach could provide a way to test swarms of autonomous systems, potentially detecting faults significantly earlier and more cost-effectively than existing methods. Another interesting direction involves applying World-In-the-Loop to both more sophisticated sensors and using more advanced mixing techniques. Allowing for additional sensors will expand the technique's capabilities to a larger range of autonomous systems, while also necessitating innovations in the area of sensor data integration. On the note of combining sensor data, various advances in computer vision could enable much more sophisticated mixing. These advancements would not only enhance the mixed reality's realism but could also enable functions not yet considered, such as removing objects from the real world.

Next, let's consider the haptic suit. An immediate future step is to test more complex instantiations of this system. Our study initially focused on a two-arm variant, which limited the range of scenarios that could be explored. A four-arm variant may allow testing a whole new range of scenarios. Another potential future step would be to explore a wider range of force-inducing mechanisms commonly used in human haptics. These mechanisms include shifting weights, motors that

spin weights around, among others. The haptic suit also presents alternative use cases yet to be explored. One such use case would be as a safety mechanism. For example, using reinforcement learning in the real world is challenging because the early stages of the approach often expect the autonomous system, such as a drone, to experience failures. The haptic suit could be used to effectively "catch" the autonomous system just before it collides with an obstacle or crashes, potentially preventing damage and allowing for safer real-world training.

Finally, a very compelling future direction for this work is the integration of the two execution methods. Imagine a future where World-In-the-Loop simulation not only incorporates simulated obstacles and potential autonomous agents into the mixed reality, but also is capable of applying physical forces to the system, such as wind. This integration would enable full system tests to be conducted in mixed-reality before real-world testing becomes necessary. The system could simulate the force of another drone's downdraft in a swarm, or the impact of a collision with an obstacle. Additionally, this integration could enable a range of new testing types. For example, consider testing a quadrotor in a lab environment with limited space, intended to simulate flying over long distances. The haptic suit could be used to counteract the forces of the drone trying to fly forward, while the World-In-the-Loop system provides mixed-reality sensor readings, making the drone's software believe it is traversing these long distances.

6.2.3 Test Adequacy

The final chapter of this thesis presents several avenues for future work. As stated in the introduction to this section, transitioning from a 2D to a 3D representation would enable the application of Physical Coverage to autonomous systems operating in 3D spaces. In fact, this is currently being explored [292]. A next step for Physical Coverage, or for a fully developed 3D approach, could be to improve how the final abstraction is made. Currently, we perform a geometric vectorization. This approach could be enhanced by using techniques that reduce the risk of missing obstacles, such as extremely thin poles, by for example dividing the region into smaller sections and computing their areas. Another idea is to incorporate semantic information about the environment into the abstraction. By doing so, we can account not only for the physical environment and semantics but

also for the environment's semantics. For instance, a system might behave very differently when encountering a moving human versus a stationary object like a tree. The idea of using semantics has been a driving force in new and emerging work in the field, such as S3C [378]. However, this work has currently been showcased using solely camera data. There is potential to combine the generality of PhysCov with S3C's ability to discern semantics to result in even more precise coverage metrics for autonomous systems.

Another potential area of research is to consider the world as dynamic rather than as a series of stationary snapshots. While this might seem trivial, it could be quite impactful. For example, the reachable sets of dynamic obstacles could be computed, allowing the technique to foresee not only potential objects the autonomous system may interact with but also if any dynamic obstacles in the world may affect the autonomous system. For instance, another vehicle approaching the same intersection at speed might have overlapping reachable sets with the autonomous system, suggesting it should be considered in the abstraction.

A final avenue of research is to link coverage back to test generation. Since Physical Coverage involves a known number of abstractions, it is possible to identify which abstractions were not exposed by a test suite. Using this information, we could develop a technique capable of constructing environments that would produce these missing abstractions. By doing so, the approach should expose the system to several new environment-state pairs, potentially revealing many behaviors that were not observed during the original test suite.

Bibliography

[1] Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, April 2021.

[2] A. Abd-Alrazaq, R. AlSaad, D. Alhuwail, A. Ahmed, P. M. Healy, S. Latifi, S. Aziz, R. Damseh, S. A. Alrazak, J. Sheikh, et al. Large language models in medical education: Opportunities, challenges, and future directions. *JMIR Medical Education*, 9(1):e48291, 2023.

[3] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[4] A. Adadi and M. Berrada. Peeking inside the black-box: a survey on explainable artificial intelligence (xai). *IEEE access*, 6:52138–52160, 2018.

[5] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.

[6] A. Agossah, F. Krupa, M. Perreira Da Silva, and P. Le Callet. Llm-based interaction for content generation: A case study on the perception of employees in an it department. In *Proceedings of the 2023 ACM International Conference on Interactive Media Experiences*, pages 237–241, 2023.

[7] V. Agostinelli, M. Wild, M. Raffel, K. A. Fuad, and L. Chen. Simul-llm: A framework for exploring high-quality simultaneous translation with large language models. *arXiv preprint arXiv:2312.04691*, 2023.

[8] M. Ahn, A. Brohan, N. Brown, Y. Chebotar, O. Cortes, B. David, C. Finn, C. Fu, K. Gopalakrishnan, K. Hausman, et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

[9] V. S. Alagar, K. Periyasamy, and K. Periyasamy. *Specification of software systems*. Springer, 2011.

[10] M. B. Alatise and G. P. Hancke. A review on challenges of autonomous mobile robot and sensor fusion methods. *IEEE Access*, 8:39830–39846, 2020.

[11] J. Alava, T. M. King, and P. J. Clarke. Automatic validation of java page flows using model-based coverage criteria. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 439–446. IEEE, 2006.

[12] W. Aldrich. Using model coverage analysis to improve the controls development process. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, page 4684, 2002.

[13] M. Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars*. PhD thesis, Technische Universität München, 2010.

[14] J. Altmayer Pizzorno and E. D. Berger. Slipcover: Near zero-overhead code coverage for python. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1195–1206, 2023.

[15] M. Alzantot, S. Chakraborty, and M. Srivastava. Sensegen: A deep learning architecture for synthetic sensor data generation. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 188–193. IEEE, 2017.

[16] H. Alzu'bi, I. Mansour, and O. Rawashdeh. Loon copter: Implementation of a hybrid unmanned aquatic–aerial quadcopter with active buoyancy control. *Journal of field Robotics*, 35(5):764–778, 2018.

[17] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software*, 86(8):1978–2001, 2013.

[18] A. Andrews, M. Abdelgawad, and A. Gario. Towards world model-based test generation in autonomous systems. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 1–12. IEEE, 2015.

[19] A. Andrews, M. Abdelgawad, and A. Gario. World model for testing autonomous systems using petri nets. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pages 65–69. IEEE, 2016.

[20] ardupilot. ardupilot. `https://ardupilot.org`, 2022.

[21] J. Arnold and R. Alexander. Testing autonomous robot control software using procedural content generation. In *International Conference on Computer Safety, Reliability, and Security*, pages 33–44. Springer, 2013.

[22] M. Ataei and A. Yousefi-Koma. Three-dimensional optimal path planning for waypoint guidance of an autonomous underwater vehicle. *Robotics and Autonomous Systems*, 67:23–32, 2015.

[23] J. Bach, S. Otten, and E. Sax. Model based scenario specification for development and test of automated driving functions. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 1149–1155. IEEE, 2016.

[24] I. Bae, J. Moon, and J. Seo. Toward a comfortable driving experience for a self-driving shuttle bus. *Electronics*, 8(9):943, 2019.

[25] S. Bak, J. Betz, A. Chawla, H. Zheng, and R. Mangharam. Stress testing autonomous racing overtake maneuvers with rrt. In *2022 IEEE Intelligent Vehicles Symposium (IV)*, pages 806–812. IEEE, 2022.

[26] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 59–66, 2010.

[27] L. Baresi and M. Young. Test oracles. 2001.

[28] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[29] N. Batchelder. Coverage.py. `https://github.com/nedbat/coveragepy`, 2022.

[30] N. Bauschmann, D. A. Duecker, T. L. Alff, and R. Seifried. Evaluation of underwater apriltag localization for highly agile micro underwater robots. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9926–9932. IEEE, 2023.

[31] R. BBN. Managing academies challenge evaluation (mace). `https://github.com/raytheonbbn/mace`, 2023.

[32] E. Beachly, C. Detweiler, S. Elbaum, B. Duncan, C. Hildebrandt, D. Twidwell, and C. Allen. Fire-aware planning of aerial trajectories and ignitions. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 685–692. IEEE, 2018.

[33] BeamNG GmbH. BeamNG.tech.

[34] J. S. Beggs. *Kinematics*. CRC Press, 1983.

[35] A. Belkin, A. Kuwertz, Y. Fischer, and J. Beyerer. World modeling for autonomous systems. *Innovative information systems modelling techniques*, 1:135–158, 2012.

[36] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 63–74, 2016.

[37] S. Bensalem, L. De Silva, A. Griesmayer, F. Ingrand, A. Legay, and R. Yan. A formal approach for incremental construction with an application to autonomous robotic systems. In *Software Composition: 10th International Conference, SC 2011, Zurich, Switzerland, June 30-July 1, 2011. Proceedings 10*, pages 116–132. Springer, 2011.

[38] H. Bhuiyan, G. Governatori, A. Bond, and A. Rakotonirainy. Traffic rules compliance checking of automated vehicle maneuvers. *Artificial Intelligence and Law*, 32(1):1–56, 2024.

[39] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[40] P. Biswal and P. K. Mohanty. Development of quadruped walking robots: A review. *Ain Shams Engineering Journal*, 12(2):2017–2031, 2021.

[41] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[42] A. Botta, W. De Donato, V. Persico, and A. Pescapé. Integration of cloud computing and internet of things: a survey. *Future generation computer systems*, 56:684–700, 2016.

[43] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150. Springer, 1997.

[44] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise uml for model-based testing. In *Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104, 2007.

[45] L. Bramblett, R. Peddi, and N. Bezzo. Coordinated multi-agent exploration, rendezvous, & task allocation in unknown environments with limited connectivity. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 12706–12712. IEEE, 2022.

[46] D. Brescianini, M. Hehn, and R. D'Andrea. Quadrocopter pole acrobatics. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3472–3479. IEEE, 2013.

[47] Brian Garrett-Glaser. Avionics - Drone Delivery Crash in Switzerland Raises Safety Concerns As UPS Forms Subsidiary. `https://www.aviationtoday.com/2019/08/08/drone-delivery-crash-in-switzerland-raises-safety-concerns/`, 2019. [Online; accessed 09-July-2023].

[48] R. A. Bridges, N. Imam, and T. M. Mintz. Understanding gpu power: A survey of profiling, modeling, and simulation methods. *ACM Computing Surveys (CSUR)*, 49(3):1–27, 2016.

[49] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[50] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa. Generating avoidable collision scenarios for testing autonomous driving systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 375–386. IEEE, 2020.

[51] L. Chen, X. Hu, W. Tian, H. Wang, D. Cao, and F.-Y. Wang. Parallel planning: A new motion planning framework for autonomous driving. *IEEE/CAA Journal of Automatica Sinica*, 6(1):236–246, 2018.

[52] L. Chen, P. Wu, K. Chitta, B. Jaeger, A. Geiger, and H. Li. End-to-end autonomous driving: Challenges and frontiers. *arXiv preprint arXiv:2306.16927*, 2023.

[53] L. Chen, M. Zaharia, and J. Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[54] M. Chen, S. Herbert, and C. J. Tomlin. Fast reachable set approximations via state decoupling disturbances. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 191–196. IEEE, 2016.

[55] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys (CSUR)*, 51(1):1–27, 2018.

[56] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. Code coverage of adaptive random testing. *IEEE Transactions on Reliability*, 62(1):226–237, 2013.

[57] X. Chen and S. Sankaranarayanan. Reachability analysis for cyber-physical systems: Are we there yet? In *NASA Formal Methods Symposium*, pages 109–130. Springer, 2022.

[58] Y. Chen, X. C. Ding, A. Stefanescu, and C. Belta. Formal approach to the deployment of distributed robotic teams. *IEEE Transactions on Robotics*, 28(1):158–171, 2011.

[59] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez, et al. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality. *See https://vicuna. lmsys. org (accessed 14 April 2023)*, 2023.

[60] G. S. Chirikjian. Synthesis of discretely actuated manipulator workspaces via harmonic analysis. In *Recent Advances in Robot Kinematics*, pages 169–178. Springer, 1996.

[61] G. Christian, T. Woodlief, and S. Elbaum. Generating realistic and diverse tests for lidar-based perception systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2604–2616. IEEE, 2023.

[62] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based coverage-driven test suite generation for software product lines. In *Model Driven Engineering Languages and Systems: 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings 14*, pages 425–439. Springer, 2011.

[63] D. Coelho and M. Oliveira. A review of end-to-end autonomous driving in urban environments. *IEEE Access*, 10:75296–75311, 2022.

[64] J. Collins, S. Chand, A. Vanderkop, and D. Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9:51416–51431, 2021.

[65] I. Colwell, B. Phan, S. Saleem, R. Salay, and K. Czarnecki. An automated vehicle safety concept based on runtime restriction of the operational design domain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1910–1917. IEEE, 2018.

[66] Comma AI. OpenPilot is an open source advanced driver assistance system. `https://comma.ai/openpilot`, 2023. [Online; accessed 11-August-2023].

[67] comma.ai. openpilot 5159878, 2022.

[68] commaai. Media. `https://www.comma.ai/media`, 2023.

[69] commaai. openpilot. `https://github.com/commaai/openpilot/blob/b816b5b/docs/LIMITATIONS.md`, 2023.

[70] commaai. openpilot. `https://github.com/commaai/openpilot/blob/b816b5b/docs/LIMITATIONS.md`, 2023.

[71] comma.ai. openpilot 2ebd7ab, 2023.

[72] comma.ai. openpilot cb2a53a, 2023.

[73] commaai. Openpilot supports 250+ vehicles. `https://comma.ai/vehicles`, 2023.

[74] comma.ai Team. Scaling for 10x user growth. `https://blog.comma.ai/scaling-for-10x-user-growth/`, 2021.

[75] Y. Cong, C. Gu, T. Zhang, and Y. Gao. Underwater robot sensing technology: A survey. *Fundamental Research*, 1(3):337–345, 2021.

[76] Cruise. Driverless is here. `https://getcruise.com/`, 2023. [Online; accessed 09-July-2023].

[77] E. Daka and G. Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[78] R. Darwish, L. N. Gwosuta, and R. Torkar. A controlled experiment on coverage maximization of automated model-based software test cases in the automotive industry. In *2017 IEEE*

International Conference on Software Testing, Verification and Validation (ICST), pages 546–547. IEEE, 2017.

[79] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM'81 Conference*, pages 254–257, 1981.

[80] M. S. P. De Melo, J. G. da Silva Neto, P. J. L. Da Silva, J. M. X. N. Teixeira, and V. Teichrieb. Analysis and comparison of robotics 3d simulators. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 242–251. IEEE, 2019.

[81] H. Delecki, M. Itkina, B. Lange, R. Senanayake, and M. J. Kochenderfer. How do we fail? stress testing perception in autonomous vehicles. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5139–5146. IEEE, 2022.

[82] M. Demir and A. Çavuşoğlu. A new driver behavior model to create realistic urban traffic environment. *Transportation research part F: traffic psychology and behaviour*, 15(3):289–296, 2012.

[83] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass. Pentestgpt: An llm-empowered automatic penetration testing tool. *arXiv preprint arXiv:2308.06782*, 2023.

[84] M. Dikmen and C. M. Burns. Autonomous driving in the real world: Experiences with tesla autopilot and summon. In *Proceedings of the 8th international conference on automotive user interfaces and interactive vehicular applications*, pages 225–228, 2016.

[85] DJI. Flame wheel arf kit. `https://www.dji.com/flame-wheel-arf/download`, 2022.

[86] W. Dong, G.-Y. Gu, Y. Ding, X. Zhu, and H. Ding. Ball juggling with an under-actuated flying robot. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 68–73. IEEE, 2015.

[87] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[88] T. Dreossi, A. Donzé, and S. A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. *Journal of Automated Reasoning*, 63:1031–1053, 2019.

[89] E. Ebeid, M. Skriver, K. H. Terkildsen, K. Jensen, and U. P. Schultz. A survey of open-source uav flight controllers and flight simulators. *Microprocessors and Microsystems*, 61:11–20, 2018.

[90] H. D. Eckhardt. *Kinematic design of machines and mechanisms*. McGraw-Hill New York, 1998.

[91] K. I. Eder, W.-l. Huang, and J. Peleska. Complete agent-driven model-based system testing for autonomous systems. *arXiv preprint arXiv:2110.12586*, 2021.

[92] U. G. Engine. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d.com*, pages 1534–4320, 2008.

[93] L. H. Erickson and S. M. LaValle. Survivability: Measuring and ensuring path diversity. In *2009 IEEE International Conference on Robotics and Automation*, pages 2068–2073. IEEE, 2009.

[94] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *Software Engineering and Formal Methods: 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings 9*, pages 204–220. Springer, 2011.

[95] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

[96] S. Fan, X. Jiang, X. Li, X. Meng, P. Han, S. Shang, A. Sun, Y. Wang, and Z. Wang. Not all layers of llms are necessary during inference. *arXiv preprint arXiv:2403.02181*, 2024.

[97] L. Feng, Q. Li, Z. Peng, S. Tan, and B. Zhou. Trafficgen: Learning to generate diverse and realistic traffic scenarios. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3567–3575. IEEE, 2023.

[98] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 177–188, 2020.

[99] J. Fitzgerald and P. G. Larsen. *Modelling systems: practical tools and techniques in software development*. Cambridge University Press, 2009.

[100] A. for Standardization of Automation and M. S. (ASAM). ASAM OpenODD: Concept Paper. Technical report, ASAM, October 2021.

[101] D. Fremont, X. Yue, T. Dreossi, S. Ghosh, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: Language-based scene generation. *arXiv preprint arXiv:1809.09310*, 2018.

[102] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019.

[103] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2020.

[104] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart. Rotors—a modular gazebo mav simulator framework. In *Robot Operating System (ROS)*, pages 595–625. Springer, 2016.

[105] A. Gambi, T. Huynh, and G. Fraser. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 257–267, 2019.

[106] A. Gambi, M. Mueller, and G. Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.

[107] D. Gandhi, L. Pinto, and A. Gupta. Learning to fly by crashing. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3948–3955. IEEE, 2017.

[108] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1147–1158. IEEE, 2020.

[109] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen. A comprehensive study of autonomous vehicle bugs. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020.

[110] M. Gautier. Identification of robots dynamics. *IFAC Proceedings Volumes*, 19(14):125–130, 1986.

[111] Gazebo. Robot simulation made easy. `http://gazebosim.org`, 2014. [Online; accessed 28-October-2020].

[112] J. Ge, J. Zhang, C. Chang, Y. Zhang, D. Yao, Y. Tian, and L. Li. Dynamic testing for autonomous vehicles using random quasi monte carlo. *IEEE Transactions on Intelligent Vehicles*, 2024.

[113] A. Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[114] Z. Ghodsi, S. K. S. Hari, I. Frosio, T. Tsai, A. Troccoli, S. W. Keckler, S. Garg, and A. Anandkumar. Generating and characterizing scenarios for safety testing of autonomous vehicles. In *2021 IEEE Intelligent Vehicles Symposium (IV)*, pages 157–164. IEEE, 2021.

[115] B. K. Ghosh, T.-J. Tarn, and N. Xi. *Control in Robotics and Automation: Sensor Based Integration*. Elsevier, 1999.

[116] S. Ghosh, Y. V. Pant, H. Ravanbakhsh, and S. A. Seshia. Counterexample-guided synthesis of perception models and control. In *2021 American Control Conference (ACC)*, pages 3447–3454. IEEE, 2021.

[117] S. Gillies et al. Shapely: manipulation and analysis of geometric objects, 2007–.

[118] A. Girard and C. Le Guernic. Efficient reachability analysis for linear systems using support functions. *IFAC Proceedings Volumes*, 41(2):8966–8971, 2008.

[119] GMC. Yukon/Yukon XL/Denali Owner's Manual, 2023.

[120] A. Goel, A. Gueta, O. Gilon, C. Liu, S. Erell, L. H. Nguyen, X. Hao, B. Jaber, S. Reddy, R. Kartha, et al. Llms accelerate annotation for medical information extraction. In *Machine Learning for Health (ML4H)*, pages 82–100. PMLR, 2023.

[121] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on software Engineering*, (2):156–173, 1975.

[122] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *29th International Conference on Software Engineering (ICSE'07)*, pages 621–631. IEEE, 2007.

[123] W. Guerra, E. Tal, V. Murali, G. Ryou, and S. Karaman. Flightgoggles: Photorealistic sensor simulation for perception-driven robotics using photogrammetry and virtual reality. *arXiv preprint arXiv:1905.11377*, 2019.

[124] S. Gupte, P. I. T. Mohandas, and J. M. Conrad. A survey of quadrotor unmanned aerial vehicles. *2012 Proceedings of IEEE Southeastcon*, pages 1–6, 2012.

[125] R. Gutiérrez, E. López-Guillén, L. M. Bergasa, R. Barea, Ó. Pérez, C. Gómez-Huélamo, F. Arango, J. Del Egido, and J. López-Fernández. A waypoint tracking controller for autonomous road vehicles using ros framework. *Sensors*, 20(14):4062, 2020.

[126] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer, 1984.

[127] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. *ACM SIGSOFT software engineering notes*, 14(8):158–167, 1989.

[128] H. Haughton and K. Lano. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[129] N. Havrikov. Efficient fuzz testing leveraging input, code, and execution. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 417–420. IEEE, 2017.

[130] M. Hehn and R. D'Andrea. A flying inverted pendulum. In *2011 IEEE International Conference on Robotics and Automation*, pages 763–770. IEEE, 2011.

[131] H. Hemmati. How effective are code coverage criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, 2015.

[132] S. Heo, C. Chung, G. Lee, and D. Wigdor. Thor's hammer: An ungrounded force feedback device utilizing propeller-induced propulsive force. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2018.

[133] J.-J. Hernandez-Lopez, A.-L. Quintanilla-Olvera, J.-L. López-Ramírez, F.-J. Rangel-Butanda, M.-A. Ibarra-Manzano, and D.-L. Almanza-Ojeda. Detecting objects using color and depth segmentation with kinect sensor. *Procedia Technology*, 3:196–204, 2012.

[134] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):1–76, 2009.

[135] C. Hildebrandt and S. Elbaum. World-in-the-loop simulation for autonomous systems validation. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10912–10919. IEEE, 2021.

[136] C. Hildebrandt and S. Elbaum. World-in-the-loop simulation for autonomous systems validation. `https://github.com/hildebrandt-carl/WorldInTheLoop`, 2022.

[137] C. Hildebrandt, S. Elbaum, and N. Bezzo. Blending kinematic and software models for tighter reachability analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36, 2020.

[138] C. Hildebrandt, S. Elbaum, N. Bezzo, and M. B. Dwyer. Feasible and stressful trajectory generation for mobile robots. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 349–362, 2020.

[139] C. Hildebrandt, S. Elbaum, N. Bezzo, and M. B. Dwyer. Feasible and stressful trajectory generation for mobile robots. `https://github.com/hildebrandt-carl/RobotTestGeneration`, 2020.

[140] C. Hildebrandt, M. von Stein, and S. Elbaum. Physcov: Physical test coverage for autonomous vehicles. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 449–461, 2023.

[141] C. Hildebrandt, M. von Stein, and S. Elbaum. Physcov: Physical test coverage for autonomous vehicles. `https://github.com/hildebrandt-carl/PhysicalCoverage`, 2023.

[142] C. Hildebrandt, T. Woodlief, and S. Elbaum. Odd-dillmma: Driving automation system odd compliance checking using llms. `https://github.com/hildebrandt-carl/ODD_diLLMma_Artifact`, 2023.

[143] C. Hildebrandt, W. Ying, S. Heo, and S. Elbaum. Mimicking real forces on a drone through a haptic suit to enable cost-effective validation. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10518–10524. IEEE, 2023.

[144] C. Hildebrandt, W. Ying, S. Heo, and S. Elbaum. Mimicking real forces on a drone through a haptic suit to enable cost-effective validation. `https://github.com/hildebrandt-carl/HapticSuit`, 2023.

[145] E. Hiroaki and S. Nader. *Gearbox simulation models with gear and bearing faults*, volume 2. chapter, 2012.

[146] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[147] A. V. Hoe, R. Sethi, and J. D. Ullman. *Compilers—principles, techniques, and tools*. Pearson Addison Wesley Longman, 1986.

[148] W. Hoenig, C. Milanes, L. Scaria, T. Phan, M. Bolas, and N. Ayanian. Mixed reality for robotics. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5382–5387. IEEE, 2015.

[149] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 327–341. Springer, 2002.

[150] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[151] J. R. Horgan and S. London. Data flow coverage and the c language. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 87–97, 1991.

[152] M. House. Typical maximum steering angle of a real car, 2016.

[153] Z. Hu, S. Guo, Z. Zhong, and K. Li. Coverage-based scene fuzzing for virtual autonomous driving testing. *arXiv preprint arXiv:2106.00873*, 2021.

[154] D. Humeniuk, F. Khomh, and G. Antoniol. Ambiegen: A search-based framework for autonomous systems testing. *arXiv preprint arXiv:2301.01234*, 2023.

[155] I. Hwang, D. M. Stipanović, and C. J. Tomlin. Polytopic approximations of reachable sets applied to linear dynamic games and a class of nonlinear systems. In *Advances in control, communication networks, and transportation systems*, pages 3–19. Springer, 2005.

[156] IEEE Connected Vehicles. Google reports self-driving car disengagements. https://site.ieee.org/connected-vehicles/2015/12/15/google-reports-self-driving-car-disengagements/, 2015. [Online; accessed 09-July-2023].

[157] INTCATCH. INTCATCH 2020. http://www.intcatch.eu, 2020. [Online; accessed 01-February-2024].

[158] S. International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE*, 2018.

[159] M. Ivanković, G. Petrović, R. Just, and G. Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[160] Jackie Wattles. CNN Business - Tesla on Autopilot crashed when the driver's hands were not detected on the wheel. https://www.cnn.com/2019/05/16/cars/tesla-autopilot-crash/index.html, 2019. [Online; accessed 09-July-2023].

[161] D. Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[162] G. Jahangirova, A. Stocco, and P. Tonella. Quality metrics and oracles for autonomous vehicles testing. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*, pages 194–204. IEEE, 2021.

[163] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in Artificial Life: Third European Conference on Artificial Life Granada, Spain, June 4–6, 1995 Proceedings 3*, pages 704–720. Springer, 1995.

[164] R. N. Jazar. *Theory of applied robotics: kinematics, dynamics, and control*. Springer Science & Business Media, 2010.

[165] S. Je, M. J. Kim, W. Lee, B. Lee, X.-D. Yang, P. Lopes, and A. Bianchi. Aero-plane: A handheld force-feedback device that renders weight motion illusion on a virtual 2d plane. In

Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, pages 763–775, 2019.

[166] X. Jia, P. Wu, L. Chen, Y. Liu, H. Li, and J. Yan. Hdgt: Heterogeneous driving graph transformer for multi-agent trajectory prediction via scene encoding. *IEEE transactions on pattern analysis and machine intelligence*, 2023.

[167] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[168] S. Jiang, Y. Zhang, and D. Yi. Test data generation approach for basis path coverage. *ACM SIGSOFT Software Engineering Notes*, 37(3):1–7, 2012.

[169] B. Jin, X. Liu, Y. Zheng, P. Li, H. Zhao, T. Zhang, Y. Zheng, G. Zhou, and J. Liu. Adapt: Action-aware driving caption transformer. *arXiv preprint arXiv:2302.00673*, 2023.

[170] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy. Inferfix: End-to-end program repair with llms. *arXiv preprint arXiv:2303.07263*, 2023.

[171] S. Jung, S. Hwang, H. Shin, and D. H. Shim. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. *IEEE Robotics and Automation Letters*, 3(3):2539–2544, 2018.

[172] JUtah. Driving around the world, 30+ countries. `https://www.youtube.com/@jutah`, December 2023.

[173] B. Kaiser, H. Weber, J. Hiller, and B. Engel. Towards the definition of metrics for the assessment of operational design domains. *Open Research Europe*, 3, 2023.

[174] J.-G. Kang, D. Lee, and S. Han. A highly maneuverable flying squirrel drone with controllable foldable wings. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6652–6659. IEEE, 2023.

[175] S. S. Kannan, V. L. Venkatesh, and B.-C. Min. Smart-llm: Smart multi-agent robot task planning using large language models. *arXiv preprint arXiv:2309.10062*, 2023.

[176] A. Karimoddini, M. A. Khan, S. Gebreyohannes, M. Heiges, E. Trewhitt, and A. Homaifar. Automatic test and evaluation of autonomous systems. *IEEE Access*, 10:72227–72238, 2022.

[177] K. Karur, N. Sharma, C. Dharmatti, and J. E. Siegel. A survey of path planning algorithms for mobile robots. *Vehicles*, 3(3):448–468, 2021.

[178] P. Kaur, S. Taghavi, Z. Tian, and W. Shi. A survey on simulators for testing self-driving cars. In *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*, pages 62–70. IEEE, 2021.

[179] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[180] Kevin Armstrong. Tesla FSD Milestone: FSD Now Drives 1 Million Miles Per Day. `https://www.notateslaapp.com/news/1327/tesla-fsd-milestone-1-million-miles-per-day-actively-driven`, 2023. [Online; accessed 09-July-2023].

[181] T. A. Khan, O. Runge, and R. Heckel. Testing against visual contracts: Model-based coverage. In *Graph Transformations: 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings 6*, pages 279–293. Springer, 2012.

[182] B. Kim, A. Jarandikar, J. Shum, S. Shiraishi, and M. Yamaura. The smt-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE, 2016.

[183] J. Kim, R. Feldt, and S. Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[184] T. Kim, C. H. Kim, J. Rhee, F. Fei, Z. Tu, G. Walkup, X. Zhang, X. Deng, and D. Xu. Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 425–442, 2019.

[185] D. Kingston, R. Beard, T. McLain, M. Larsen, and W. Ren. Autonomous vehicle technologies for small fixed wing uavs. In *2nd AIAA" Unmanned Unlimited" Conf. and Workshop & Exhibit*, page 6559, 2003.

[186] W. Kirkwood. Auv incidents and outcomes. In *OCEANS 2009*, pages 1–5. IEEE, 2009.

[187] R. Kirner. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems*, 2009:1–16, 2009.

[188] Kisling, Nestico & Redick LLC. Self-Driving Car Accident Statistics. `https://www.knrlegal.com/car-accident-lawyer/self-driving-car-accident-statistics/`, 2021. [Online; accessed 23-March-2024].

[189] L. Klampfl, F. Klück, and F. Wotawa. Using genetic algorithms for automating automated lane-keeping system testing. *Journal of Software: Evolution and Process*, 36(3):e2520, 2024.

[190] M. Klischies, M. Rothenbeck, A. Steinführer, I. A. Yeo, C. dos Santos Ferreira, J. Mohrmann, C. Faber, and C. Schirnick. Auv abyss workflow: autonomous deep sea exploration for ocean research. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6. IEEE, 2018.

[191] J. Kocić, N. Jovičić, and V. Drndarević. An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms. *Sensors*, 19(9):2064, 2019.

[192] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[193] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE intelligent vehicles symposium (IV)*, pages 1094–1099. IEEE, 2015.

[194] S. Konur, C. Dixon, and M. Fisher. Formal verification of probabilistic swarm behaviours. In *Swarm Intelligence: 7th International Conference, ANTS 2010, Brussels, Belgium, September 8-10, 2010. Proceedings 7*, pages 440–447. Springer, 2010.

[195] P. Koopman. The heavy tail safety ceiling. In *Automated and Connected Vehicle Systems Testing Symposium*, volume 1145, pages 8950–8961. SAE, 2018.

[196] P. Koopman and F. Fratrik. How many operational design domains, objects, and events? *Safeai@ aaai*, 4, 2019.

[197] P. Koopman and M. Wagner. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety*, 4(1):15–24, 2016.

[198] T. Krisher. Us report: Nearly 400 crashes of automated tech vehicles, 2022. [Online; accessed 22-July-2023].

[199] S. Kucuk and Z. Bingul. *Robot kinematics: Forward and inverse kinematics*. INTECH Open Access Publisher London, UK, 2006.

[200] R. Kuhn, R. N. Kacker, Y. Lei, and D. Simos. Input space coverage matters. *Computer*, 53(1):37–44, 2020.

[201] T. H. Kung, M. Cheatham, A. Medenilla, C. Sillos, L. De Leon, C. Elepaño, M. Madriaga, R. Aggabao, G. Diaz-Candido, J. Maningo, et al. Performance of chatgpt on usmle: Potential for ai-assisted medical education using large language models. *PLoS digital health*, 2(2):e0000198, 2023.

[202] T. R. Kurfess et al. *Robotics and automation handbook*, volume 414. CRC press Boca Raton, FL, 2005.

[203] Kurt Barnhart. Partners Kansas State University Salina and Westar Energy build one of the largest enclosed flight facilities for UAS in the nation. `https://www.k-state.edu/media/newsreleases/oct15/pavilion101415.html`, 2015. [Online; accessed 22-August-2019].

[204] A. A. Kurzhanskiy and P. Varaiya. Ellipsoidal toolbox (et). In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 1498–1503. IEEE, 2006.

[205] J. Lai, W. Gan, J. Wu, Z. Qi, and P. S. Yu. Large language models in law: A survey. *arXiv preprint arXiv:2312.03718*, 2023.

[206] A. v. Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159, 2000.

[207] T. Laurent, S. Klikovits, P. Arcaini, F. Ishikawa, and A. Ventresque. Parameter coverage for testing of autonomous driving systems under uncertainty. *ACM Transactions on Software Engineering and Methodology*, 32(3):1–31, 2023.

[208] J. Leathrum, Y. Shen, R. Mielke, and N. Gonda. Integrating virtual and augmented reality based testing into the development of autonomous vehicles. *Proceedings of ModSim World 2018*, pages 24–26, 2018.

[209] C. W. Lee, N. Nayeer, D. E. Garcia, A. Agrawal, and B. Liu. Identifying the operational design domain for an automated driving system through assessed risk. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1317–1322. IEEE, 2020.

[210] M.-j. Lee and Y.-g. Ha. Autonomous driving control using end-to-end deep learning. In *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 470–473. IEEE, 2020.

[211] T.-C. Lee and P.-A. Hsiung. Mutation coverage estimation for model checking. In *International Symposium on Automated Technology for Verification and Analysis*, pages 354–368. Springer, 2004.

[212] V. Lekic and Z. Babic. Automotive radar and camera fusion using generative adversarial networks. *Computer Vision and Image Understanding*, 184:1–8, 2019.

[213] H. K. Leung and L. White. Insights into regression testing (software testing). In *Proceedings. Conference on Software Maintenance-1989*, pages 60–69. IEEE, 1989.

[214] E. Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[215] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer. Av-fuzzer: Finding safety violations in autonomous driving systems. In *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*, pages 25–36. IEEE, 2020.

[216] H. Li, Y. Hao, Y. Zhai, and Z. Qian. The hitchhiker's guide to program analysis: A journey with large language models. *arXiv preprint arXiv:2308.00245*, 2023.

[217] H. Liang, J. S. Dong, J. Sun, and W. E. Wong. Software monitoring through formal specification animation. *Innovations in Systems and Software Engineering*, 5:231–241, 2009.

[218] M. Liffiton, B. Sheese, J. Savelka, and P. Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes. *arXiv preprint arXiv:2308.06921*, 2023.

[219] A. Ligot and M. Birattari. On mimicking the effects of the reality gap with simulation-only experiments. In *Swarm Intelligence: 11th International Conference, ANTS 2018, Rome, Italy, October 29–31, 2018, Proceedings 11*, pages 109–122. Springer, 2018.

[220] R. Lill and F. Saglietti. Model-based testing of autonomous systems based on coloured petri nets. In *ARCS 2012*, pages 1–5. IEEE, 2012.

[221] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze. Metamorphic model-based testing of autonomous systems. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 35–41. IEEE, 2017.

[222] H. Liu and H. B. K. Tan. Covering code behavior on input validation in functional testing. *Information and Software Technology*, 51(2):546–553, 2009.

[223] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Li, and L. Sun. Harnessing the power of llm to support binary taint analysis. *arXiv preprint arXiv:2310.08275*, 2023.

[224] Z. Liu, H. Tang, A. Amini, X. Yang, H. Mao, D. L. Rus, and S. Han. Bevfusion: Multi-task multi-sensor fusion with unified bird's-eye view representation. In *2023 IEEE international conference on robotics and automation (ICRA)*, pages 2774–2781. IEEE, 2023.

[225] C. Llanes, M. Abate, and S. Coogan. Safety from fast, in-the-loop reachability with application to uavs. In *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*, pages 127–136. IEEE, 2022.

[226] S. Loncaric. A survey of shape analysis techniques. *Pattern recognition*, 31(8):983–1001, 1998.

[227] A. Loquercio, E. Kaufmann, R. Ranftl, A. Dosovitskiy, V. Koltun, and D. Scaramuzza. Deep drone racing: From simulation to reality with domain randomization. *IEEE Transactions on Robotics*, 36(1):1–14, 2019.

[228] C. Lu, H. Zhang, T. Yue, and S. Ali. Search-based selection and prioritization of test scenarios for autonomous driving systems. In *International Symposium on Search Based Software Engineering*, pages 41–55. Springer, 2021.

[229] V. L. Lucieer and A. L. Forrest. Emerging mapping techniques for autonomous underwater vehicles (auvs). *Seafloor Mapping along Continental Shelves: Research and Techniques for Visualizing Benthic Environments*, pages 53–67, 2016.

[230] W. Luo, B. Yang, and R. Urtasun. Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3569–3577, 2018.

[231] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

[232] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[233] M. Machin, J. Guiochet, H. Waeselynck, J.-P. Blanquart, M. Roy, and L. Masson. Smof: A safety monitoring framework for autonomous systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 48(5):702–715, 2016.

[234] M. J. Madou. *Fundamentals of microfabrication: the science of miniaturization*. CRC press, 2018.

[235] I. Majzik, O. Semeráth, C. Hajdu, K. Marussy, Z. Szatmári, Z. Micskei, A. Vörös, A. A. Babikian, and D. Varró. Towards system-level testing with coverage guarantees for autonomous vehicles. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 89–94. IEEE, 2019.

[236] S. Makridakis. The forthcoming artificial intelligence (ai) revolution: Its impact on society and firms. *Futures*, 90:46–60, 2017.

[237] P. Mallozzi, P. Pelliccione, A. Knauss, C. Berger, and N. Mohammadiha. Autonomous vehicles: state of the art, future trends, and challenges. *Automotive systems and software engineering: State of the art and future trends*, pages 347–367, 2019.

[238] T. Mashimo and S. Izuhara. Recent advances in micromotors. *IEEE Access*, 8:213489–213501, 2020.

[239] Massachusetts Institute of Technology. Flight Goggles. `https://flightgoggles.mit.edu`, 2019. [Online; accessed 01-January-2020].

[240] Mathworks. Matlab and Python. `https://www.mathworks.com/products/matlab/matlab-and-python.html`, 2020. [Online; accessed 26-January-2020].

[241] MATLAB. *version 9.12.0 (R2020a)*. The MathWorks Inc., Natick, Massachusetts, 2020.

[242] M. McCaffrey. *Unreal Engine VR Cookbook: Developing Virtual Reality with UE4*. Addison-Wesley Professional, 2017. Chater 7: Character Inverse Kinematics.

[243] P. McCausland. Self-driving uber car that hit and killed woman did not recognize that pedestrians jaywalk, 2019. [Online; accessed 09-July-2023].

[244] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[245] P. McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[246] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[247] L. R. Medsker and L. Jain. Recurrent neural networks. *Design and Applications*, 5(64-67):2, 2001.

[248] D. Mellinger and V. Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525. IEEE, 2011.

[249] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin. A time-dependent hamilton-jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on automatic control*, 50(7):947–957, 2005.

[250] S. Mittal and J. S. Vetter. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)*, 47(4):1–35, 2015.

[251] N. Mohammad and N. Bezzo. A robust and fast occlusion-based frontier method for autonomous navigation in unknown cluttered environments. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6324–6331. IEEE, 2022.

[252] G. M. Muktadir, A. Jawad, I. Paranjape, J. Whitehead, and A. Shepelev. Procedural generation of high-definition road networks for autonomous vehicle testing and traffic simulations. *SAE International Journal of Connected and Automated Vehicles*, 6(12-06-01-0007):99–120, 2022.

[253] M. Müller, S. Lupashin, and R. D'Andrea. Quadrocopter ball juggling. In *2011 IEEE/RSJ international conference on Intelligent Robots and Systems*, pages 5113–5120. IEEE, 2011.

[254] V. C. Müller and N. Bostrom. Future progress in artificial intelligence: A survey of expert opinion. *Fundamental issues of artificial intelligence*, pages 555–572, 2016.

[255] M. Musuvathi and S. Qadeer. Chess: Systematic stress testing of concurrent software. In *Logic-Based Program Synthesis and Transformation: 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers 16*, pages 15–16. Springer, 2007.

[256] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[257] A. Nadeem, A. Awais, et al. Testfilter: a statement-coverage based test case reduction technique. In *2006 IEEE International Multitopic Co nference*, pages 275–280. IEEE, 2006.

[258] J. Nakama, R. Parada, J. P. Matos-Carvalho, F. Azevedo, D. Pedro, and L. Campos. Autonomous environment generator for uav-based simulation. *Applied Sciences*, 11(5):2185, 2021.

[259] K. D. Nguyen and C. Ha. Development of hardware-in-the-loop simulation based on gazebo and pixhawk for unmanned aerial vehicles. *International Journal of Aeronautical and Space Sciences*, 19(1):238–249, 2018.

[260] Nicole Casal Moore. M-Air autonomous aerial vehicle outdoor lab opens. `https://news.umich.edu/m-air-autonomous-aerial-vehicle-outdoor-lab-opens/`, 2019. [Online; accessed 22-August-2019].

[261] S. Nidhra and J. Dondeti. Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[262] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, (6):795–803, 1984.

[263] Nuro. Reimagining local delivery. `https://www.nuro.ai/technology`, 2023. [Online; accessed 09-July-2023].

[264] M. Odelga, P. Stegagno, H. H. Bülthoff, and A. Ahmad. A setup for multi-uav hardware-in-the-loop simulations. In *2015 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS)*, pages 204–210. IEEE, 2015.

[265] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In *International Conference on Machine Learning*, pages 4901–4911. PMLR, 2019.

[266] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software testing, verification and reliability*, 13(1):25–53, 2003.

[267] S. Oosterwaal, A. v. Deursen, R. Coelho, A. A. Sawant, and A. Bacchelli. Visualizing code and coverage changes for code review. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 1038–1041, 2016.

[268] OpenAI. Introducing ChatGPT, 2022.

[269] OpenAI. GPT-4 Technical Report, 2023.

[270] OpenAI. GPT-4V(ision) System Card, 2023.

[271] OpenAI. Rate limits. `https://platform.openai.com/docs/guides/rate-limits?context=tier-free`, 2023.

[272] J.-P. Ore, C. Detweiler, and S. Elbaum. Phriky-units: a lightweight, annotation-free physical unit inconsistency detection tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 352–355, 2017.

[273] J.-P. Ore, S. Elbaum, A. Burgin, and C. Detweiler. Autonomous aerial water sampling. *Journal of Field Robotics*, 32(8):1095–1113, 2015.

[274] S. Park, B. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[275] Parrot. Anafi. `https://www.parrot.com/us/drones/anafi`, 2019. [Online; accessed 11-November-2019].

[276] Parrot. Olympe Documentation. `https://developer.parrot.com/docs/olympe/`, 2019. [Online; accessed 20-November-2019].

[277] Parrot. Parrot-Sphinx. `https://developer.parrot.com/docs/sphinx/whatissphinx.html`, 2019. [Online; accessed 22-August-2019].

[278] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karsai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 83–90. IEEE, 2009.

[279] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11:339–353, 2009.

[280] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[281] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[282] M. A. Persinger. Support for eddington's number and his approach to astronomy: recent developments in the physics and chemistry of the human brain. *International Letters of Chemistry, Physics and Astronomy*, 8, 2013.

[283] G. Petrovic, M. Ivankovic, B. Kurtz, P. Ammann, and R. Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53. IEEE, 2018.

[284] S. Pichai. An important next step on our AI journey, 2023.

[285] B. Pietsch. 2 killed in driverless tesla car crash, officials say, 2021. [Online; accessed 09-July-2023].

[286] Pixhawk. Pixhawk. `https://pixhawk.org/`, 2022.

[287] K. Popp and W. Schiehlen. *Ground vehicle dynamics*. Springer Science & Business Media, 2010.

[288] D. Powell, J. Arlat, H. N. Chu, F. Ingrand, and M.-O. Killijian. Testing the input timing robustness of real-time control software for autonomous systems. In *2012 Ninth European Dependable Computing Conference*, pages 73–83. IEEE, 2012.

[289] M. Proetzsch, K. Berns, T. Schuele, and K. Schneider. Formal verification of safety behaviours of the outdoor robot ravon. In *International Conference on Informatics in Control, Automation and Robotics*, volume 2, pages 157–164. SCITEPRESS, 2007.

[290] K. Project. luacov. `https://github.com/keplerproject/luacov`, 2022.

[291] Y. Qin, S. Hu, Y. Lin, W. Chen, N. Ding, G. Cui, Z. Zeng, Y. Huang, C. Xiao, C. Han, et al. Tool learning with foundation models. *arXiv preprint arXiv:2304.08354*, 2023.

[292] Z. Qureshi, S. Bhat, C. Hildebrandt, and S. Elbaum. Physcov - quantifying physical coverage for autonomous drones (microsoft airsim). `https://github.com/ZoraizQ/physcov_airsim_drone`, 2021.

[293] M. Rahnemoonfar, J. Johnson, and J. Paden. Ai radar sensor: Creating radar depth sounder images based on generative adversarial network. *Sensors*, 19(24):5479, 2019.

[294] R. Rajamani. *Vehicle dynamics and control*. Springer Science & Business Media, 2011.

[295] Z. Rasool, S. Barnett, S. Kurniawan, S. Balugo, R. Vasa, C. Chesser, and A. Bahar-Fuchs. Evaluating llms on document-based qa: Exact answer selection and numerical extraction using cogtale datase. *arXiv preprint arXiv:2311.07878*, 2023.

[296] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[297] Q. A. Ribeiro, M. Ribeiro, and J. Castro. Requirements engineering for autonomous vehicles: a systematic literature review. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 1299–1308, 2022.

[298] D. J. Richardson, S. L. Aha, and T. O. O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118, 1992.

[299] S. Riedmaier, T. Ponn, D. Ludwig, B. Schick, and F. Diermeyer. Survey on scenario-based safety assessment of automated vehicles. *IEEE access*, 8:87456–87477, 2020.

[300] A. Romdhana, M. Ceccato, G. C. Georgiu, A. Merlo, and P. Tonella. Cosmo: code coverage made easier for android. In *2021 14th IEEE conference on software testing, verification and validation (ICST)*, pages 417–423. IEEE, 2021.

[301] ROS. ROS common messages. `http://wiki.ros.org/common_msgs`, 2017. [Online; accessed 20-September-2020].

[302] ROS. ROS transform library. `http://wiki.ros.org/tf`, 2017. [Online; accessed 20-September-2020].

[303] A. Roush, E. Zakirov, A. Shirokov, P. Lunina, J. Gane, A. Duffy, C. Basil, A. Whitcomb, J. Benedetto, and C. DeWolfe. Llm as an art director (ladi): Using llms to improve text-to-media generators. *arXiv preprint arXiv:2311.03716*, 2023.

[304] C. Sakaridis, D. Dai, and L. Van Gool. Semantic foggy scene understanding with synthetic data. *International Journal of Computer Vision*, 126(9):973–992, 2018.

[305] E. Santana and G. Hotz. Learning a driving simulator. *arXiv preprint arXiv:1608.01230*, 2016.

[306] L. V. Santana, A. S. Brandao, and M. Sarcinelli-Filho. Outdoor waypoint navigation with the ar. drone quadrotor. In *2015 international conference on unmanned aircraft systems (ICUAS)*, pages 303–311. IEEE, 2015.

[307] A. Santos, A. Cunha, and N. Macedo. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 56–62, 2018.

[308] A. Sarkar and K. Czamecki. A behavior driven approach for sampling rare event situations for autonomous vehicles. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6407–6414. IEEE, 2019.

[309] T. Sayre-McCord, W. Guerra, A. Antonini, J. Arneberg, A. Brown, G. Cavalheiro, Y. Fang, A. Gorodetsky, D. McCoy, S. Quilter, et al. Visual-inertial navigation algorithm development using photorealistic camera simulation in the loop. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2566–2573. IEEE, 2018.

[310] H. Schafer, E. Santana, A. Haden, and R. Biasini. A commute in data: The comma2k19 dataset. *arXiv preprint arXiv:1812.05752*, 2018.

[311] M. Schwall, T. Daniel, T. Victor, F. Favaro, and H. Hohnhold. Waymo public road safety performance data. *arXiv preprint arXiv:2011.00038*, 2020.

[312] D. M. Schwarz, L. Rolland, and J. B. Johnston. Identifying real-world problems with automated vehicles by detecting behavioral differences in steering movements between the human driver and machine. In *2022 IEEE 28th International Conference on Engineering, Technology and Innovation (ICE/ITMC) & 31st International Association For Management of Technology (IAMOT) Joint Conference*, pages 1–9. IEEE, 2022.

[313] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[314] U. K. G. D. Service. The highway code. `https://www.gov.uk/guidance/the-highway-code/using-the-road-159-to-203`, 2015.

[315] S. Shah, D. Dey, C. Lovett, and A. Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[316] S. Shair, J. Chandler, V. Gonzalez-Villela, R. M. Parkin, and M. Jackson. The use of aerial images and gps for mobile robot waypoint navigation. *IEEE/ASME Transactions On Mechatronics*, 13(6):692–699, 2008.

[317] A. Shankar, S. Elbaum, and C. Detweiler. In-air exchange of small payloads between multirotor aerial systems. In *International Symposium on Experimental Robotics*, pages 511–523. Springer, 2018.

[318] A. Shankar, S. Elbaum, and C. Detweiler. Freyja: A full multirotor system for agile & precise outdoor flights. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 217–223. IEEE, 2021.

[319] Y. Shaoqiang, L. Zhong, and L. Xingshan. Modeling and simulation of robot based on matlab/simmechanics. In *2008 27th Chinese Control Conference*, pages 161–165. IEEE, 2008.

[320] Q. Shen, L. Jiang, and H. Xiong. Person tracking and frontal face capture with uav. In *2018 IEEE 18th International Conference on Communication Technology (ICCT)*, pages 1412–1416. IEEE, 2018.

[321] Y. Shen, J. Zhou, D. Xu, F. Zhao, J. Xu, J. Chen, and S. Li. Aggressive trajectory generation for a swarm of autonomous racing drones. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7436–7441. IEEE, 2023.

[322] S. Shi, L. Jiang, D. Dai, and B. Schiele. Motion transformer with global intention localization and local movement refinement. *Advances in Neural Information Processing Systems*, 35:6531–6543, 2022.

[323] T. Simonite. Tesla Tests Self-Driving Functions with Secret Updates to Its Customers' Cars, 2016.

[324] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes. A survey on procedural modelling for virtual worlds. In *Computer graphics forum*, volume 33, pages 31–50. Wiley Online Library, 2014.

[325] J. M. Spivey and J.-R. Abrial. *The Z notation*, volume 29. Prentice Hall Hemel Hempstead, 1992.

[326] M. Srikanth, A. Soto, A. Annaswamy, E. Lavretsky, and J.-J. Slotine. Controlled manipulation with multiple quadrotors. In *AIAA Guidance, Navigation, and Control Conference*, page 6547, 2011.

[327] D. Stanev and K. Moustakas. Modeling musculoskeletal kinematic and dynamic redundancy using null space projection. *PloS one*, 14(1), 2019.

[328] Stanford Artificial Intelligence Laboratory et al. Robotic operating system.

[329] C. Stark, C. Medrano-Berumen, and M. İ. Akbaş. Generation of autonomous vehicle validation scenarios using crash data. In *2020 SoutheastCon*, pages 1–6. IEEE, 2020.

[330] L. Steccanella, D. D. Bloisi, A. Castellini, and A. Farinelli. Waterline and obstacle detection in images from low-cost autonomous boats for environmental monitoring. *Robotics and Autonomous Systems*, 124:103346, 2020.

[331] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, 1996.

[332] H. W. Stone. *Kinematic modeling, identification, and control of robotic manipulators*, volume 29. Springer Science & Business Media, 1987.

[333] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, V. Vasudevan, W. Han, J. Ngiam, H. Zhao, A. Timofeev, S. Ettinger, M. Krivokon, A. Gao, A. Joshi, Y. Zhang, J. Shlens, Z. Chen, and D. Anguelov. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[334] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore. Structural test coverage criteria for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–23, 2019.

[335] Y. Sun, C. M. Poskitt, X. Zhang, and J. Sun. Redriver: Runtime enforcement for autonomous vehicles. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[336] S. Suo, S. Regalado, S. Casas, and R. Urtasun. Trafficsim: Learning to simulate realistic multi-agent behaviors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10400–10409, 2021.

[337] Swiss Post. Swiss Post drone transport in the healthcare sector. `https://www.post.ch/en/about-us/innovation/innovations-in-development/drones`, 2023. [Online; accessed 09-July-2023].

[338] Z. Tahir and R. Alexander. Coverage based testing for v&v and safety assurance of self-driving autonomous vehicles: A systematic literature review. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 23–30. IEEE, 2020.

[339] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad. A survey of end-to-end driving: Architectures and training methods. *IEEE Transactions on Neural Networks and Learning Systems*, 33(4):1364–1384, 2020.

[340] TASS International. PreScan - A Simulation and Verification Environment for Intelligent Vehicle Systems. `https://tass.plm.automation.siemens.com/prescan`, 2019. [Online; accessed 22-August-2019].

[341] K. Teeyapan, J. Wang, T. Kunz, and M. Stilman. Robot limbo: Optimized planning and control for dynamically stable robots under vertical obstacles. In *2010 IEEE International Conference on Robotics and Automation*, pages 4519–4524. IEEE, 2010.

[342] Tesla. Model Y Owner's Manual Software version: 2023.32 North America, 2023.

[343] E. Thorn, S. C. Kimmel, M. Chaka, B. A. Hamilton, et al. A framework for automated driving system testable cases and scenarios. Technical report, United States. Department of Transportation. National Highway Traffic Safety . . . , 2018.

[344] S. Thrun. Probabilistic algorithms in robotics. *Ai Magazine*, 21(4):93–93, 2000.

[345] Y. Tian, K. Pei, S. Jana, and B. Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[346] S. Tomar. Converting video formats with ffmpeg. *Linux journal*, 2006(146):10, 2006.

[347] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[348] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[349] P. Tsankov, M. T. Dashti, and D. Basin. Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 56–66, 2013.

[350] C. E. Tuncali and G. Fainekos. Rapidly-exploring random trees for testing automated vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 661–666. IEEE, 2019.

[351] Unity. Unity Asset Store. https://assetstore.unity.com/, 2024. [Online; accessed 01-January-2024].

[352] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[353] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

[354] K. P. Valavanis and G. J. Vachtsevanos. *Handbook of Unmanned Aerial Vehicles-5 Volume Set*. New York, NY, USA: Springer, 2014.

[355] K. P. Valavanis and G. J. Vachtsevanos. *Handbook of unmanned aerial vehicles*. Springer, 2015.

[356] K. L. Vasudev. Review of autonomous underwater vehicles. *Autonomous Vehicles*, pages 31–48, 2020.

[357] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[358] Velodyne Lidar. Alpha prime.

[359] VICON. Motion capture system. https://www.vicon.com, 2020. [Online; accessed 20-September-2020].

[360] Virginia Tech. Virginia Tech Drone Park officially open . https://vtnews.vt.edu/articles/2018/04/ictas-droneparkopens.html, 2018. [Online; accessed 22-August-2019].

[361] M. von Stein, D. Shriver, and S. Elbaum. Deepmaneuver: Adversarial test generation for trajectory manipulation of autonomous vehicles. *IEEE Transactions on Software Engineering*, 2023.

[362] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng. Robot: Robustness-oriented testing for deep learning systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE, 2021.

[363] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221*, 2023.

[364] J. Wang, L. Zhang, Y. Huang, J. Zhao, and F. Bella. Safety of autonomous vehicles. *Journal of advanced transportation*, 2020:1–13, 2020.

[365] L. Wang, M. Han, X. Li, N. Zhang, and H. Cheng. Review of classification methods on unbalanced data sets. *IEEE Access*, 9:64606–64628, 2021.

[366] M. Wang, Y. Su, H. Li, J. Li, J. Liang, and H. Liu. Aggregating single-wheeled mobile robots for omnidirectional movements. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 314–320. IEEE, 2023.

[367] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 397–409. IEEE, 2021.

[368] Waymo. First Million Rider-Only Miles: How the Waymo Driver is Improving Road Safety. `https://waymo.com/blog/2023/02/first-million-rider-only-miles-how.html`, 2023. [Online; accessed 09-July-2023].

[369] Waymo LLC. Waymo - Self Driving Car. `https://waymo.com`, 2019. [Online; accessed 09-July-2023].

[370] Wayve. Pioneering a new approach to self-driving: AV2.0. `https://wayve.ai/`, 2023. [Online; accessed 09-July-2023].

[371] J. Wei, J. M. Snider, T. Gu, J. M. Dolan, and B. Litkouhi. A behavioral planning framework for autonomous driving. In *2014 IEEE Intelligent Vehicles Symposium Proceedings*, pages 458–464. IEEE, 2014.

[372] E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Software Eng.*, 17:703–711, 1991.

[373] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):641–655, 1983.

[374] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[375] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[376] N. Williams. Towards exhaustive branch coverage with pathcrawler. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 117–120. IEEE, 2021.

[377] T. Woodlief, S. Elbaum, and K. Sullivan. Semantic image fuzzing of ai perception systems. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1958–1969, 2022.

[378] T. Woodlief, F. Toledo, S. Elbaum, and M. B. Dwyer. S3c: Spatial semantic scene coverage for autonomous vehicles. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[379] A. C. Woods and H. M. La. Dynamic target tracking and obstacle avoidance using a drone. In *International Symposium on Visual Computing*, pages 857–866. Springer, 2015.

[380] J. Wu, R. Antonova, A. Kan, M. Lepert, A. Zeng, S. Song, J. Bohg, S. Rusinkiewicz, and T. Funkhouser. Tidybot: Personalized robot assistance with large language models. *arXiv preprint arXiv:2305.05658*, 2023.

[381] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

[382] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*, 2023.

[383] R. B. Wynn, V. A. Huvenne, T. P. Le Bas, B. J. Murton, D. P. Connelly, B. J. Bett, H. A. Ruhl, K. J. Morris, J. Peakall, D. R. Parsons, et al. Autonomous underwater vehicles (auvs): Their past, present and future contributions to the advancement of marine geoscience. *Marine geology*, 352:451–468, 2014.

[384] M. Xanthidis, N. Karapetyan, H. Damron, S. Rahman, J. Johnson, A. O'Connell, J. M. O'Kane, and I. Rekleitis. Navigation in the presence of obstacles for an agile autonomous underwater vehicle. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 892–899. IEEE, 2020.

[385] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. M. López. Multimodal end-to-end autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):537–547, 2020.

[386] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[387] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li. Diffchaser: Detecting disagreements for deep neural networks. International Joint Conferences on Artificial Intelligence Organization, 2019.

[388] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, Y. Xia, et al. Survey of robot 3d path planning algorithms. *Journal of Control Science and Engineering*, 2016, 2016.

[389] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, 2006.

[390] Z. Yang, X. Jia, H. Li, and J. Yan. Llm4drive: A survey of large language models for autonomous driving. *arXiv e-prints*, pages arXiv–2311, 2023.

[391] B. Yao, M. Jiang, D. Yang, and J. Hu. Empowering llm-based machine translation with cultural awareness. *arXiv preprint arXiv:2305.14328*, 2023.

[392] Yaroslav S. Yatskiv. Kinematics and Physics of Celestial Bodies. `https://www.springer.com/journal/11963`, 2007. [ISSN: 0884-5913].

[393] W. Yeadon and T. Hardy. The impact of ai in physics education: A comprehensive review from gcse to university levels. *arXiv preprint arXiv:2309.05163*, 2023.

[394] E. Yel and N. Bezzo. Reachability-based adaptive uav scheduling and planning in cluttered and dynamic environments. In *Workshop on Informative Path Planning and Adaptive Sampling at ICRA*, 2018.

[395] E. Yel, T. X. Lin, and N. Bezzo. Reachability-based self-triggered scheduling and replanning of uav operations. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 221–228. IEEE, 2017.

[396] O. Zaki, M. Dunnigan, V. Robu, and D. Flynn. Reliability and safety of autonomous systems based on semantic modelling for self-certification. *Robotics*, 10(1):10, 2021.

[397] C. Zhang, Y. Liu, D. Zhao, and Y. Su. Roadview: A traffic scene simulator for autonomous vehicle simulation testing. In *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 1160–1165. IEEE, 2014.

[398] H. Zhang, J. Chen, F. Jiang, F. Yu, Z. Chen, J. Li, G. Chen, X. Wu, Z. Zhang, Q. Xiao, et al. Huatuogpt, towards taming language model to be a doctor. *arXiv preprint arXiv:2305.15075*, 2023.

[399] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of*

the 33rd ACM/IEEE International Conference on Automated Software Engineering, pages 132–142, 2018.

[400] Q. Zhang, D. K. Hong, Z. Zhang, Q. A. Chen, S. Mahlke, and Z. M. Mao. A systematic framework to identify violations of scenario-dependent driving rules in autonomous vehicle software. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(2):1–25, 2021.

[401] X. Zhang, S. Khastgir, and P. Jennings. An odd-based scalable assurance framework for automated driving systems. Technical report, SAE Technical Paper, 2023.

[402] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[403] Z. Zhong, G. Kaiser, and B. Ray. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. *IEEE Transactions on Software Engineering*, 2022.

[404] Z. Q. Zhou and L. Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, 2019.

[405] D. Zhu, J. Chen, X. Shen, X. Li, and M. Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.

[406] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

[407] H. Zhuang, Z. Qin, K. Hui, J. Wu, L. Yan, X. Wang, and M. Berdersky. Beyond yes and no: Improving zero-shot llm rankers via scoring fine-grained relevance labels. *arXiv preprint arXiv:2310.14122*, 2023.

[408] Zipline. Welcome to the best delivery experience not on earth. `https://www.flyzipline.com/`, 2023. [Online; accessed 09-July-2023].

Appendix

A : Oracle Analysis

This section enumerates the papers reviewed and details our observations on the types of inputs and outputs employed. In cases where oracles were not explicitly identified or described by the approach, we analyzed the included studies to determine the criteria used for pass/fail evaluations. The first table identifies instances where oracles were absent, and the second table lists occurrences where oracles were explicitly identified.

Papers reviewed from the last two years, where oracles were not used or identified.

Paper	Oracle Type	Input Type	Output Type
Dynamic Testing for Autonomous Vehicles Using Random Quasi Monte Carlo [112]	Unknown		
Procedural generation of high-definition road networks for autonomous vehicle testing and traffic simulations [252]	NA		
Coordinated multi-agent exploration, rendezvous, & task allocation in unknown environments with limited connectivity [45]	NA		
A Highly Maneuverable Flying Squirrel Drone with Controllable Foldable Wings [174]	NA		
Aggressive Trajectory Generation for A Swarm of Autonomous Racing Drones [321]	NA		
A robust and fast occlusion-based frontier method for autonomous navigation in unknown cluttered environments [251]	NA		
Aggregating Single-wheeled Mobile Robots for Omnidirectional Movements [366]	NA		

Papers reviewed from the last two years, where oracles were identified.

Paper	Oracle Type	Input Type	Output Type
REDriver: Runtime Enforcement for Autonomous Vehicles [335]	Specifications	Objects	Trajectories
Traffic rules compliance checking of automated vehicle maneuvers [38]	Specifications	Objects	Trajectories
Safety from fast, in-the-loop reachability with application to UAVs [225]	Derived (Differential)	-	Position
Semantic image fuzzing of AI perception systems [377]	Specifications (Metamorphic)	-	Perception
Generating Realistic and Diverse Tests for LiDAR-Based Perception Systems [61]	Specifications (Metamorphic)	-	Perception
AmbieGen: A Search-based Framework for Autonomous Systems Testing [154]	Implicit	Road	Position
Stress testing autonomous racing overtake maneuvers with rrt [25]	Implicit	-	Crash
Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles [403]	Implicit	-	Crash
	Implicit	Road	Position
Parameter Coverage for Testing of Autonomous Driving Systems Under Uncertainty [207]	Derived (Differential)	Objects	Trajectory
Do as i can, not as i say: Grounding language in robotic affordances [8]	Implicit	-	Plan
A review of end-to-end autonomous driving in urban environments [63]	Implicit	-	Crash
	Implicit	Road	Position
S3C: Spatial Semantic Scene Coverage for Autonomous Vehicles [378]	Implicit	-	Steering
DeepManeuver: Adversarial test generation for trajectory manipulation of autonomous vehicles [361]	Implicit	-	Crash
	Implicit	Road	Position
How do we fail? stress testing perception in autonomous vehicles [81]	Implicit	-	Position
Trafficgen: Learning to generate diverse and realistic traffic scenarios [97]	Implicit	-	Crash
Identifying real-world problems with automated vehicles by detecting behavioral differences in steering movements between the human driver and machine [312]	Derived (Differential)	-	Steering Angle
Automatic test and evaluation of autonomous systems [176]	Implicit	-	Perception
Using genetic algorithms for automating automated lane-keeping system testing [189]	Implicit	Road	Position
Evaluation of Underwater AprilTag Localization for Highly Agile Micro Underwater Robots [30]	Implicit	-	Perception