PhysCov: Physical Test Coverage for Autonomous Vehicles

Carl Hildebrandt
The University of Virginia
Charlottesville, USA
hildebrandt.carl@virginia.edu

Meriel von Stein
The University of Virginia
Charlottesville, USA
meriel@virginia.edu

Sebastian Elbaum
The University of Virginia
Charlottesville, USA
selbaum@virginia.edu

ABSTRACT

Adequately exercising the behaviors of autonomous vehicles is fundamental to their validation. However, quantifying an autonomous vehicle's testing adequacy is challenging as the system's behavior is influenced both by its *state* as well as its *physical environment*. To address this challenge, our work builds on two insights. First, data sensed by an autonomous vehicle provides a unique spatial signature of the physical environment inputs. Second, given the vehicle's current state, inputs residing outside the autonomous vehicle's physically reachable regions are less relevant to its behavior. Building on those insights, we introduce an abstraction that enables the computation of a physical environment-state coverage metric, *PhysCov*. The abstraction combines the sensor readings with a physical reachability analysis based on the vehicle's state and dynamics to determine the region of the environment that may affect the autonomous vehicle. It then characterizes that region through a parameterizable geometric approximation that can trade quality for cost. Tests with the same characterizations are deemed to have had similar internal states and exposed to similar environments and thus likely to exercise the same set of behaviors, while tests with distinct characterizations will increase *PhysCov*. A study on two simulated and one real system's dataset examines *PhysCovs*'s ability to quantify an autonomous vehicle's test suite, showcases its characterization cost and precision, investigates its correlation with failures found and potential for test selection, and assesses its ability to distinguish among real-world scenarios.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Test Adequacy, Coverage Metrics, Autonomous Systems

ACM Reference Format:
Carl Hildebrandt, Meriel von Stein, and Sebastian Elbaum. 2023. PhysCov: Physical Test Coverage for Autonomous Vehicles. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598069

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598069

1 INTRODUCTION

This work explores a fundamental and open question in testing autonomous vehicles: to what extent does a system test suite exercise the potential system behaviors?

Typically, software engineers rely on *abstractions of the input space to define equivalent input classes*. The underlying principle is that inputs within an equivalent class exercise similar behavior. If the abstraction is effective at clustering inputs into classes that lead to similar behavior, then the percentage of classes covered provides a means to quantify the extent that a test suite exercises the system.

In the context of autonomous systems, such as autonomous cars and drones, the system behavior is significantly influenced by the system's state and its surrounding physical environment. The vehicle's pose, speed, and acceleration, the road topology, the surrounding traffic, the signage, and other objects in the environment influence the vehicle's actions. Yet, existing adequacy criteria are insufficient to abstract autonomous vehicles' system state and environment into equivalent classes.

Structural code coverage [62, 65] and the coverage of learned components [27, 64] are not cognizant of the system's physical state and environment attributes, resulting in distinct scenarios that render the same coverage. The industry reported miles driven criterion [6, 30] does not consider the state of the vehicle nor the scenarios traveled, so miles driven at high or low speeds or through suburban traffic or multi-lane highway are considered equivalent. Coverage of requirements defined by domain experts as per the system state [28] or the environment [49] are valuable to establish acceptance tests but are not scalable given the space of behaviors triggered by state and environment. Scenario coverage [40] 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 pre-computed, severely curtailing its applicability beyond limited simulation environments. Trajectory coverage relies on a vehicle position [26] but ignores other aspects of the system state and the environment. This means, for example, that two tests that cause the vehicle to visit the same positions are deemed equivalent even if one does so at high speed while changing lanes while the other does it at slower speeds while avoiding obstacles.

However, incorporating a vehicle's state and physical environment into a coverage criterion is challenging for several reasons. First, the vehicle's state is rich and amenable to a myriad of representations, making the generalization of a coverage measure difficult. Second, the world has a practically infinite number of complex environments, which makes identifying equivalence classes and estimating the total number of classes hard and onerous. Third, the many subtle interactions between vehicle state and the environment and how that affects a vehicle's behavior means that any coverage metrics must consider those dimensions jointly.

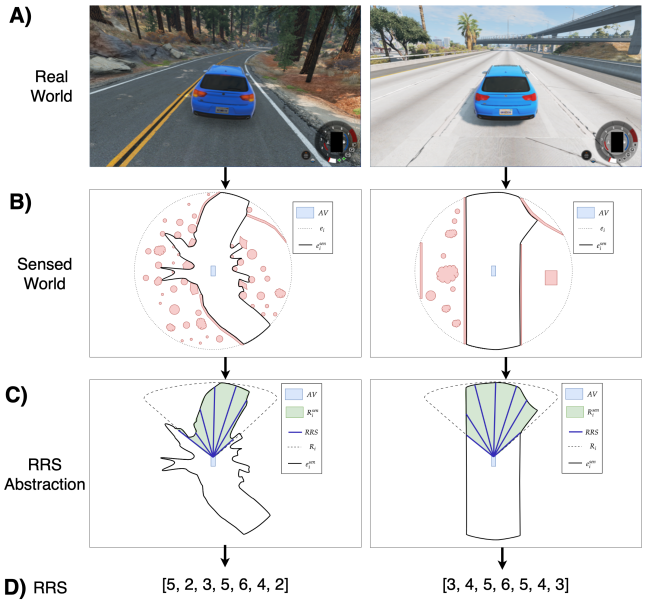


Figure 1: A) An *ASUT* in two environments using two different behaviors. B) The *ASUT* senses the environment (solid lines). C) Our approach characterizes the sensed reachable region as it is the most likely to influence its behavior (dashed lines), through a series of vectors (blue solid lines). D) These vectors provide distinct signatures, which we call *RRS*.

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 under test (*ASUT*). It builds on two basic principles. First, sensed environment inputs in the spatial trajectory of the vehicle are likely more valuable and should be retained. Second, the physical space where those inputs reside 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 and physical dynamics of the autonomous vehicle 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 classes) for computational cost.

Motivating example. Consider the vehicle in Figure 1 operating in two distinct testing scenarios (A). Both are scenarios that the autonomous vehicle is expected to navigate, and both can trigger a different set of behaviors. The multi-lane highway requires the car to be aware of other vehicles merging, it should overtake slow-moving vehicles, and can expect that vehicles on the road are moving in the same direction. The single-lane highway requires the car to be aware of any intersections, it should avoid overtaking, and it needs to be aware that the other lane is for traffic in the opposite direction. If an abstraction places both of these scenarios into the same equivalence class, and cannot distinguish between them, then their joint coverage will be underestimated. If another scenario like the first one is added but the abstractions place it in a different equivalence class, then the test suite coverage will be overestimated even though they would exhibit the same behaviors.

The goal of our approach is to identify meaningful differences in the environment that may reveal distinct behaviors while grouping tests that may trigger the same behavior in the same equivalence classes. It starts by reading the *ASUT* spatial sensors (e.g., LiDAR, radars, ultrasound) values, which capture some portion of the physical world around the vehicle (B, solid lines). Then, using the vehicle's kinematic and dynamic models and current state (pose, velocity, acceleration), it computes the region of the physical environment it is most likely to interact with (C, dashed lines). The assumption is that this is the most relevant portion of the environment as no physical action produced by the *ASUT* could result in the vehicle colliding with any part of the environment outside this region (similar to how we stop worrying about a pedestrian looking to cross the road after we have driven past them). We conjecture that the intersection of these regions, the sensed and reachable region, is the most relevant to the vehicle's behavior. Our approach then efficiently approximates the shape of that region through a parameterizable set of vectors (C, blue lines), 7 in this example, that extends from the vehicle's current position to the edge of the region. The magnitude of those vectors provides a signature that characterizes that particular input space at a given time, which we call the *RRS* signature (D of Figure 1). Executing a test suite will result in a sequence of such *RRS* signatures. Tests with the same set of *RRS* signatures will be grouped into equivalence classes as they are likely to have had similar internal states and external environments and, thus, likely to produce the same set of behaviors. An additional benefit of our approach is that the total number of unique *RRS* can be computed. The quotient of the exposed over the potential total signatures is *PhysCov*.

Contributions. The main contributions of the paper are:
1) An abstraction *RRS*, and a new metric, *PhysCov*, that allows quantifying the extent to which a test suite exposes an *ASUT* to distinct physical environments. The process underlying *RRS* is novel in how it reduces the environments by integrating physical reachability analysis and geometric vectorization of the reachable space.
2) The implementation and study of *RRS* and *PhysCov* on two simulated and one real system datasets. The study shows *RRS* range of applicability through its parameterization and *PhysCov* value to judge the thoroughness of test suites in terms of the number of distinct environments explored, how it can effectively assist in test case selection, and how it can identify similar and distinct scenarios.

2 BACKGROUND

Testing software aims to expose faults and increase developers' confidence that the software is correct [59]. Test adequacy criteria are useful to determine when enough testing has been done [42, 65]. The complexity of the input space and of the systems that consume those inputs makes it challenging to assert the adequacy of a test suite. We describe existing approaches to conquer those sources of complexity.

2.1 Input and System Models Coverage

The first type of approach develops models of the *input space* and judges to what extent a test suite covers those models. Goodenough

and Gerhart [19], Hamlet [20], Ntafos et al. [43], and Weyuker et al. [58], set the foundations for modeling the input space as equivalence classes. For such approaches to be cost-effective, the rendered classes must be homogeneous in the behavior they cause, and the partitioning into classes must be efficient. Today we see a range of input models, from grammars for systems that consume strings [22] to system configurations [35] and finite sets of constraints [51]. The second type of approach develops models of the *system* and judges the extent a suite covers them. These models partition the input space but do so relative to the implementation of the system. A common type of this model is based on structural code coverage metrics [5, 21, 24, 32, 60, 62, 65].

While these techniques have found success in traditional software systems, they struggle with some attributes of autonomous vehicles [34]. First, autonomous vehicle components are inherently statistical in nature and thus result in non-deterministic behaviors [50]. Emerging work on coverage criteria for learned components [39, 48] mitigates this challenge, but they constitute just one of the sources of non-determinism. Second, regardless of the test, many system components, such as those for control and planning, tend to have a linear control flow which causes code metrics to saturate quickly. Alternative emerging criteria such as those based on requirement coverage such as vehicle stability, assured clear distance ahead, minimum separation [28], and covered features such as dirt roads, bridges, or highways [49] are helpful in defining critical acceptance tests. However, such enumerations cannot be expected to define the long tail of possible scenarios.

2.2 Physical Environment Coverage

In the context of autonomous systems, there are many approaches for automatic test generation [1, 4, 11, 31, 46, 56]. This work is novel in its focus on the environment to guide the testing process [15, 16, 23, 52, 61]. For example, Fremont et al. develop a probabilistic language to specify environments for testing the perception components of autonomous systems [14], Gambi et al. use procedural content generation to automatically create challenging scenarios for autonomous vehicles [15, 16], and Althoff et al. create critical situations with a small solution space where the autonomous vehicle must avoid a collision [4]. However, none of these efforts use a coverage metric to determine test adequacy. Majzik et al. defined scenario coverage [40], where a scenario describes a sequence of scenes, and a scene describes a snapshot of the environment [53]. This approach falls short in that the process of converting any real physical environment into an abstract representation does not scale, and there is no mechanism to compute the total number of possible scenes. Hu et al. compute coverage of road regions within a scenario [26]. However, as they point out, this solution is limited as it assumes rectangular roads, overlooks the temporal aspect of the trajectory, and ignores the physical environment. Quantifying test adequacy for autonomous vehicles per the state and environment covered over the possible scenes remains an open problem.

3 PROBLEM DEFINITION

Consider an *ASUT* , with various spatial sensors (e.g., LiDAR, radar, sonar). Given a test τ, the *ASUT* will complete it by repeating three basic operations: sensing the environment, processing the available information, and acting on this knowledge. The ASUT starts in an initial environment $e_0 \in \mathcal{E}$ from the set of all environments and an initial state $s_0 \in \mathcal{S}$ from the set of all possible states. At each point in time t, the *ASUT*'s sensors provide an abstraction of the environment. This abstraction e_t^{sen} is a function of both the environment e_t at time t, but also the state s_t (e.g., the pose of the vehicle affects what portions of the environment it senses), and can be described as $e_t^{sen} = sensors(s_t, e_t)$. The combination of system state and sensed environment are then used by the *ASUT* to determine its next action $a_i = processing(e_i^{sen}, s_i)$. Through the completion of an action, the *ASUT* will be presented with a new environment e_{i+1} and state s_{i+1}. Therefore, by performing τ, the *ASUT* will sense environments $E^{sen} = \langle e_0^{sen}, e_1^{sen}, \ldots, e_t^{sen} \rangle$, that will result in a sequence of actions $A = \langle a_1, a_2, \ldots, a_t \rangle$.

Given the *ASUT* and a test suite $\mathcal{T} = (\tau_1, \tau_2, \ldots, \tau_k)$, we aim to measure the test adequacy of $\mathcal{T}$. We judge $\mathcal{T}$ to be adequate if it exposes the *ASUT* to all environment state pairs $\beta = E^{sen} \times \mathcal{S}$ since that would guarantee exploration of all possible behaviors A. If, after observing all resulting A, no failures are found, the system is tested adequately. It is unlikely that $\mathcal{T}$ will expose the system to β, and thus we measure $\alpha \subseteq \beta$, where α is the set of environment-state pairs experienced during $\mathcal{T}$. We can then compute test adequacy using $\frac{\alpha}{\beta}$, and subsequently determine how well a system is tested.

Computing $\frac{\alpha}{\beta}$ for a test suite is challenging for four main reasons. First, the internal state of the *ASUT* is highly complex and can be represented differently among implementations. Therefore, to be broadly applicable, the approach needs to be system implementation and sensor agnostic. Second, autonomous systems operate in the real world, which has a practically infinite number of E and thus E_i^{sen}, where each of the E are highly complex. Third, autonomous systems can have several sensors and sensor types rendering distinct E_i^{sen}, further increasing the number of sensed environments. Finally, sensors are often susceptible to noise. Therefore in the presence of the same physical environment, they may render different values when faced with the same environment.

Thus, to compute $\beta = E^{sen} \times \mathcal{S}$, we first need an implementation-independent way of identifying $\mathcal{S}$. Second, we need a way to abstract E^{sen} such that the abstraction is finite and efficient while also allowing for the abstractions of the sensed environment to be comparable so that they can be grouped into equivalence classes. Third, we need the abstraction of E^{sen} to be sensor independent.

4 APPROACH

We now define the *RRS* abstraction to compute *PhysCov*, a metric to quantify the unique, relevant physical environments perceived by the ASUT during the execution of a test suite.

4.1 Pipeline Overview

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 2 presents an overview of the *RRS* abstraction pipeline to compute that approximation.

The pipeline takes as input the sequence of all visited environment-state pairs $\langle E^{sen}, S \rangle_k$, collected by executing each test k in $\tau \in \mathcal{T}$ on the *ASUT*. The pipeline has three stages. First, the approach needs to define the region around the *ASUT* most relevant based on the

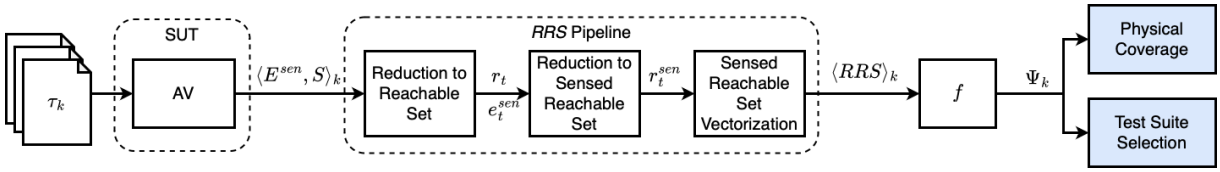


Figure 2: Overview - The approach starts with an initial test suite, and an *ASUT*. It then executes the test suite on the *ASUT* and passes the state and sensor data to the *RRS* abstraction pipeline. This creates *RRS* signatures which can compute *PhysCov*.

ASUT state. To do this, the *RRS* pipeline uses the *ASUT* state s_t at each timestep t, to compute the *ASUT*'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 *ASUT* into the abstraction, as the reachable set requires both the dynamic and kinematic model K and the *ASUT*'s internal state s_t (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 object outside can not be hit regardless of the autonomous vehicle's behavior.

Second, the approach needs to identify the parts of the environment most relevant to the current behavior of the *ASUT*. The approach assumes the vehicle's sensors capture e_t^{sen}, the portions of the environment that drive its behavior. Then, it computes $r_t^{sen} = e_t^{sen} \cap r_t$. 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 *ASUT*'s decision at time t. By removing all portions of the environment that are not likely to affect the *ASUT*'s decision-making, the approach reduces the environment such that $|R^{sen}| << |E^{sen}| << |E|$.

Third, although r_t^{sen} is significantly smaller than the physical environment, it has a complex geometry and is represented as an innumerable continuous space. The approach needs a way to characterize this identified region so that it can be easily compared with others and counted. The approach uses geometric vectorization to approximate r_t^{sen} using an array of vectors originating from the autonomous vehicle. The unique sequence of vector magnitudes is called an *RRS* signature. This process is object and sensor-agnostic and can account for unexpected sensor readings, which one might experience when operating in a new and unexpected environment. The output of this final step is a sequence of *RRS* signatures, where for each (s_t, e_t^{sen}) pair, there are an *RRS* signatures. The sequence $\langle RRS\rangle_k$ represents an abstraction of the *ASUT*'s sensed physical environments as perceived by its states during the execution of τ_k.

The approach then passes the $\langle RRS\rangle_k$ 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 $\langle RRS\rangle_k$ into the set of *RRS* signatures $\{RRS\}_k$. 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 *ASUT* over all possible environment state pairs. The approach computes $\alpha = |\{\Psi_1 \cup \Psi_2 \cup \cdots \cup \Psi_k\}|$, where $\Psi_k = \{RRS\}_k$. 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 $O(n)$ 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 [12, 18, 55, 63].

4.2 RRS Signature Generation

Given $\langle E^{sen}, S\rangle_k = \langle (e_1^{sen}, s_1), (e_2^{sen}, s_2), \ldots, (e_t^{sen}, s_t)\rangle_k$ produced by test τ_k, the objective of *RRS* is to generate a sequence of signatures $\langle RRS\rangle_k = \langle RRS_1, RRS_2, \ldots RRS_t\rangle_k$. We will use the scenario in Figure 3 as a running example to show the three stages to compute such an *RSS* signature.

4.2.1 Reduction to Reachable Set. The reachability analysis component aims to identify which parts of e_t^{sen} may affect the *ASUT*'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 e_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 [54]. This would result in an environment e_t^{lidar} that covers an area of $282743m^2$, most of which is not actively relevant to the *ASUT*. For example, suppose the *ASUT* is moving forward at 70mph. In that case, the space behind the *ASUT* 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[1] [2, 29], a well-known approach at

[1]Distinct from code reachability [10] as it relies on the system's physical attributes.

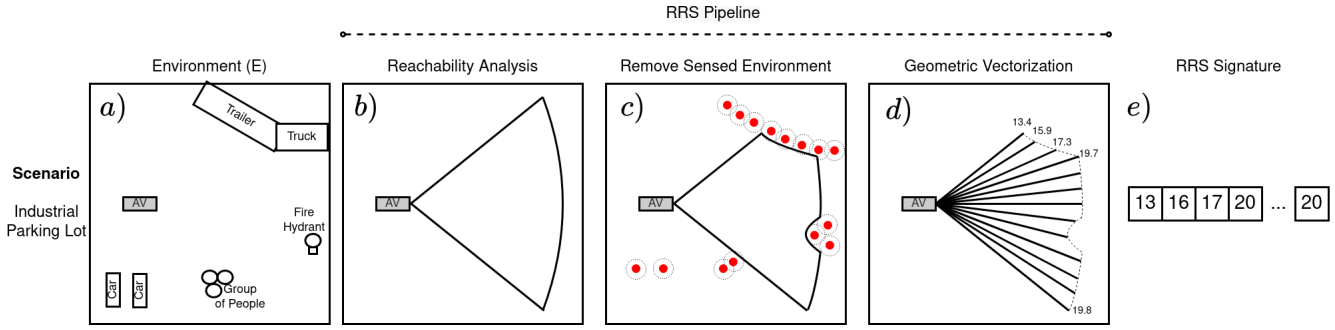


Figure 3: Given an *ASUT* and its environment. The reachable set is computed to identify the regions more likely to affect the *ASUT* behavior given its state within a time horizon. The resultant reachable set is then constrained using the sensed environment. Last, the reduced reachable set is approximated using geometric vectorization, and the *RRS* signature is produced.

the center of many challenging tasks for mobile autonomous vehicles. Given the system's kinematics and dynamics, current state, and a time horizon, this analysis computes a system's attainable region [38, 41, 55]. It does so by iteratively applying the set of admissible control sequences to a system model with a known state. More formally, the future state of a vehicle can be described using nonlinear dynamics $s_{i+1} = \mathcal{P}(s_i, a_i)$, where $\mathcal{P}$ is the plant model, s_i is the current state, and a_i the current action. Given all valid actions A at s_i, the vehicle's reachable set r_i is computed through equation 1 with $\oplus$ representing the geometric sum.

$$r_i = s_i \oplus \int_i^{i+1} \mathcal{P}(s_i, A)dt \tag{1}$$

An illustration of the reachable set of a ground vehicle is shown in Figure 3b (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 [33, 45]. The input to the system is $u = [a, \delta]$, which describes the acceleration a and the front wheel steering angle δ.

There are many techniques to compute the reachability set, trading accuracy and cost [3, 7, 13]. For example, an inexpensive approximation might use a geometric polytope that is constructed using the *ASUT* maximum steering angle and a radius equivalent to its maximum velocity. We use this approach in Figure 3b, where the reachable set for the *ASUT* is described by a sector whose origin is the *ASUT*'s current position and the radius is defined by the maximum distance the *ASUT* can traverse given its current linear and angular speed. Alternatively, a more sophisticated reachability analytical method would use the full *ASUT* kinematic and dynamic models to describe the *ASUT*'s motion [57].

The output of this component is illustrated in Figure 3-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.

4.2.2 Reduction to the Sensed Reduced Reachable Set. The *ASUT* continually captures and processes e_t^{sen}. A wide range of sensors

Algorithm 1: Geometric Vectorization Algorithm

```
Given r_t^sen, s_t, n, D, I
    RRS = []
    for i in n do
        angle = D[i]
        v = compute_vector(r_t^sen, s_t, angle)
        discretized_v = round(v, I[i])
        RRS.append(discretized_v)
    end
    return RRS
```

available for different ASUT provide some form of spatial awareness, and thus e_t^{sen} might come in various forms. Figure 3c shows e_t^{sen} as a point cloud, a collection of points representing the sensed objects around the *ASUT*.

This component integrates e_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 δ represented as the dashed lines in Figure 3c. 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. Figures 3c 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.

4.2.3 Reduced Reachable Set Vectorization. Once we have computed r_t^{sen}, the final step 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[37]. 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 *ASUT* 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 the *RRS* signature.

Algorithm 1 describes the process in more detail. The inputs are r_t^{sen}, the current state of the *ASUT* 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 *ASUT*'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 *ASUT*'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.

4.3 Usages of RRS Signature

The final step in the pipeline is to convert the sequence of $\langle RRS \rangle_k$ 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 $\langle RRS \rangle_k$, $\langle RRS \rangle_k \rightarrow \{RRS\}_k = \Psi_k$, which is similar to converting a trace of lines of code covered into a coverage vector just containing the unique lines that were executed.

4.3.1 PhysCov Computation. *PhysCov* aims to capture how much of the relevant physical environment was covered by a test suite and is defined in equation 2. Intuitively, this metric represents the percentage of distinct environments perceived that are relevant to the *ASUT* given its state. The number of 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 2'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 $\mathcal{I}$.

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

4.3.2 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 construct a new test suite $\mathcal{T}^{select}$ with only a single test from each equivalence class $\mathcal{T}^{select} \subseteq \mathcal{T}$, where $\mathcal{T}^{select} = \{\tau_i, \tau_j \in \mathcal{T} | \Psi_i \neq \Psi_j\}$.

4.3.3 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.

4.4 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 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 Experimental 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.

5.2 Environments

5.2.1 HighwayEnv. Shown in Figure 4, HighwayEnv [36] 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

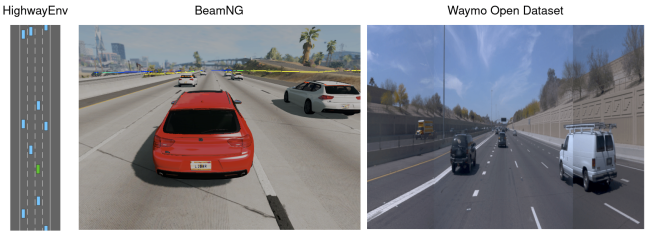


Figure 4: HighwayEnv[36], BeamNG[9], and the Waymo Open Perception Dataset [47] environments.

explore the possible scenarios that could occur in HighwayEnv. Of the 1,000,000 tests, 94% executed without any failures.

5.2.2 BeamNG. The second environment, shown in Figure 4, is BeamNG.tech [9] (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*.

5.2.3 Waymo Open Perception Dataset. Our third environment is the real world as perceived by the Waymo vehicle (Waymo Open Perception Dataset [47]). 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.

5.3 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.

5.3.1 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[8]. BeamNG's autonomous control software is written in Lua, and coverage tools for Lua are still quite limited (e.g., LuaCov[44]). Therefore, we implemented our own tool to track line, branch, and intraprocedural prime path coverage[2], 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.

5.3.2 Trajectory Coverage. Trajectory coverage measures the extent to which an autonomous vehicle covers discrete regions on a road [26]. 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.

5.4 Evaluation Criteria

To evaluate *PhysCov*, we primarily looked at two criteria, consistency of equivalent classes and failures.

5.4.1 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.

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

5.4.2 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.

5.5 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[3].

5.5.1 State and Sensor Collection. To account for differences between each of the environment's 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 [54]. 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.

5.5.2 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 3b. 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 [25] (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.

5.5.3 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

[3] www.github.com/hildebrandt-carl/PhysicalCoverage

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 [17].

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.

5.6 RQ#1 - Ψ Effectiveness

This research question explores how effective *PhysCov* is at generating equivalent input classes. Tables 1 and 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

Table 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 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 %

the approximation quality, 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 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 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 for BeamNG (similar trends can be observed for HighwayEnv, and we share those results in our artifact[3]). The shaded regions show the minimum and maximum coverage for the test suite size. To generate these regions, we computed

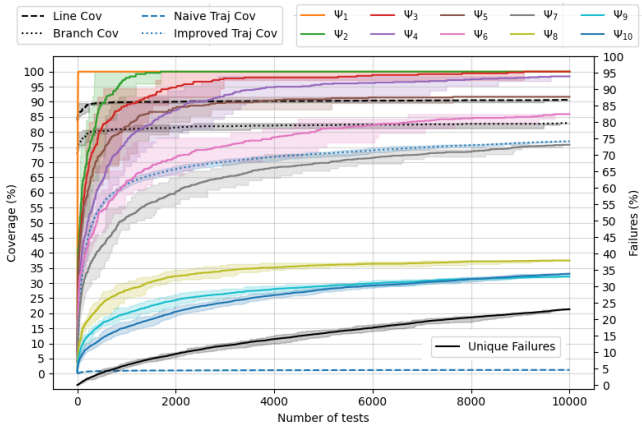


Figure 5: *PhysCov* for tests in BeamNG

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 3 and 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 all suites. 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 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.

Table 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 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

5.7 RQ#2 - 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* 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 6 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.

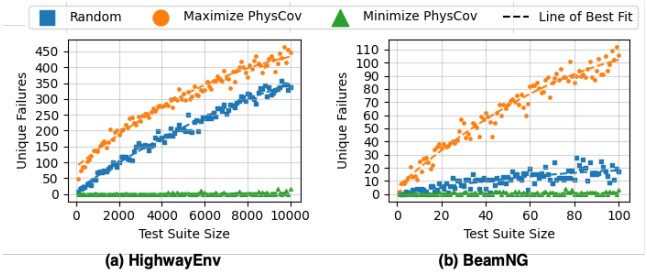


Figure 6: Unique failures found when selecting test suites that maximize or minimize *PhysCov*

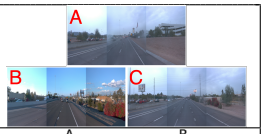
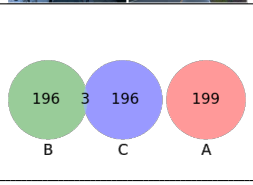
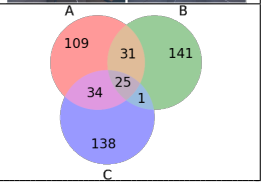
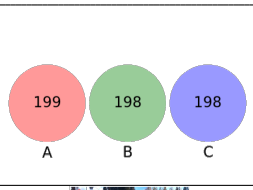
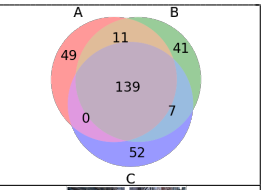
5.8 RQ#3 - Real-World Scenarios

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 environments the vehicle will encounter.

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. 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 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. 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 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.

Table 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		

Overall, RQ3 shows that our technique can be applied to real world datasets and can distinguish between different and similar scenarios, which is helpful in deciding how best to optimize and augment a test suite.

6 CONCLUSION

This paper introduces a general approach to quantify the number of unique environments 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.

In the future, we would like to dig deeper into richer state and reachability models that offer more precision and more sophisticated characterization techniques of the sensed reachable space. We would also like to empirically explore several factors and parameters, like the time horizons and the application to other autonomous vehicles such as drones. Last, we would like to start exploiting the potential synergy between the proposed approach and the techniques being developed for test generation for autonomous vehicles, where Ψ could be used to assess and guide those techniques.

ACKNOWLEDGMENTS

This work was funded in part through NSF grants #1924777 and #1909414, and AFOSR grant #FA9550-21-1-0164.

REFERENCES

[1] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1016–1026.

[2] Matthias Althoff and John M Dolan. 2014. Online verification of automated road vehicles using reachability analysis. *IEEE Transactions on Robotics* 30, 4 (2014), 903–918.

[3] Matthias Althoff, Goran Frehse, and Antoine Girard. 2021. Set propagation techniques for reachability analysis. *Annual Review of Control, Robotics, and Autonomous Systems* 4, 1 (2021).

[4] Matthias Althoff and Sebastian Lutz. 2018. Automatic generation of safety-critical test scenarios for collision avoidance of road vehicles. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1326–1333.

[5] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. 2010. Structural coverage of feasible code. In *Proceedings of the 5th Workshop on Automation of Software Test*. 59–66.

[6] Subho S Banerjee, Saurabh Jha, James Cyriac, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2018. Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 586–597.

[7] Somil Bansal, Mo Chen, Sylvia Herbert, and Claire J Tomlin. 2017. Hamilton-jacobi reachability: A brief overview and recent advances. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2242–2253.

[8] Ned Batchelder. 2022. Coverage.py. https://github.com/nedbat/coveragepy.

[9] BeamNG GmbH. [n. d.]. *BeamNG.tech*. https://www.beamng.tech/

[10] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*. Springer, 135–150.

[11] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. 2020. Generating avoidable collision scenarios for testing autonomous driving systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 375–386.

[12] Mo Chen, Sylvia Herbert, and Claire J Tomlin. 2016. Fast reachable set approximations via state decoupling disturbances. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, 191–196.

[13] Martin Fauré, Jérôme Cieslak, David Henry, Anatole Verhaegen, and Finn Ankersen. 2022. A Survey on Reachable Set Techniques for Fault Recoverability Assessment. *IFAC-PapersOnLine* 55, 6 (2022), 272–277.

[14] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 63–78.

[15] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. 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*. 257–267.

[16] Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. 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*. 318–328.

[17] Sean Gillies et al. 2007–. Shapely: manipulation and analysis of geometric objects. https://github.com/Toblerity/Shapely

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

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

[20] Dick Hamlet. 2000. On subdomains: Testing, profiles, and components. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. 71–76.

[21] Mary Jean Harrold and Mary Lou Soffa. 1989. Interprocedual data flow testing. *ACM SIGSOFT software engineering notes* 14, 8 (1989), 158–167.

[22] Nikolas Havrikov. 2017. Efficient fuzz testing leveraging input, code, and execution. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 417–420.

[23] Carl Hildebrandt, Sebastian Elbaum, Nicola Bezzo, and Matthew B Dwyer. 2020. Feasible and stressful trajectory generation for mobile robots. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 349–362.

[24] Joseph Robert Horgan and Saul London. 1991. Data flow coverage and the C language. In *Proceedings of the symposium on Testing, analysis, and verification*. 87–97.

[25] Michael House. 2016. Typical Maximum steering angle of a real car. https://gamedev.stackexchange.com/questions/50022/typical-maximum-steering-angle-of-a-real-car

[26] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. 2021. Coverage-based scene fuzzing for virtual autonomous driving testing. *arXiv preprint arXiv:2106.00873* (2021).

[27] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. 2020. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review* 37 (2020), 100270.

[28] Sae International. 2018. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE* (2018).

[29] Reza N Jazar. 2010. *Theory of applied robotics: kinematics, dynamics, and control*. Springer Science & Business Media.

[30] Nidhi Kalra and Susan M Paddock. 2016. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice* 94 (2016), 182–193.

[31] BaekGyu Kim, Akshay Jarandikar, Jonathan Shum, Shinichi Shiraishi, and Masahiro Yamaura. 2016. The SMT-based automatic road network generation in vehicle simulation environment. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.

[32] Raimund Kirner. 2009. Towards preserving model coverage and structural code coverage. *EURASIP Journal on Embedded Systems* 2009 (2009), 1–16.

[33] Jason Kong, Mark Pfeiffer, Georg Schildbach, and Francesco Borrelli. 2015. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1094–1099.

[34] Philip Koopman and Michael Wagner. 2016. Challenges in autonomous vehicle testing and validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.

[35] Rick Kuhn, Raghu N Kacker, Yu Lei, and Dimitris Simos. 2020. Input Space Coverage Matters. *Computer* 53, 1 (2020), 37–44.

[36] Edouard Leurent. 2018. An Environment for Autonomous Driving Decision-Making. https://github.com/eleurent/highway-env.

[37] Sven Loncaric. 1998. A survey of shape analysis techniques. *Pattern recognition* 31, 8 (1998), 983–1001.

[38] John Lygeros. 2004. On reachability and minimum cost optimal control. *Automatica* 40, 6 (2004), 917–927.

[39] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[40] István Majzik, Oszkár Semeráth, Csaba Hajdu, Kristóf Marussy, Zoltán Szatmári, Zoltán Micskei, András Vörös, Aren A Babikian, and Dániel Varró. 2019. 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)*. IEEE, 89–94.

[41] Ian M Mitchell, Alexandre M Bayen, and Claire J Tomlin. 2005. A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. *IEEE Transactions on automatic control* 50, 7 (2005), 947–957.

[42] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[43] Simeon C Ntafos. 1984. On required element testing. *IEEE Transactions on Software Engineering* 6 (1984), 795–803.

[44] Kepler Project. 2022. luacov. https://github.com/keplerproject/luacov.

[45] Rajesh Rajamani. 2011. *Vehicle dynamics and control*. Springer Science & Business Media.

[46] Elias Rocklage, Heiko Kraft, Abdullah Karatas, and Jörg Seewig. 2017. Automated scenario generation for regression testing of autonomous vehicles. In *2017 ieee 20th international conference on intelligent transportation systems (itsc)*. IEEE, 476–483.

[47] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. 2020. Scalability in Perception for Autonomous Driving: Waymo Open Dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[48] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. 2019. Structural test coverage criteria for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–23.

[49] Eric Thorn, Shawn C Kimmel, Michelle Chaka, Booz Allen Hamilton, et al. 2018. *A framework for automated driving system testable cases and scenarios*. Technical Report. United States. Department of Transportation. National Highway Traffic Safety ....

[50] Sebastian Thrun. 2000. Probabilistic algorithms in robotics. *Ai Magazine* 21, 4 (2000), 93–93.

[51] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2013. Semi-valid input coverage for fuzz testing. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 56–66.

[52] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. 2018. Simulation-based adversarial test generation for autonomous vehicles with

machine learning components. In *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 1555–1562.

[53] Simon Ulbrich, Till Menzel, Andreas Reschka, Fabian Schuldt, and Markus Maurer. 2015. Defining and substantiating the terms scene, situation, and scenario for automated driving. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. IEEE, 982–988.

[54] Velodyne Lidar. [n. d.]. *Alpha Prime*. https://velodynelidar.com/products/alpha-prime/

[55] Abraham P Vinod, Baisravan HomChaudhuri, and Meeko MK Oishi. 2017. Forward stochastic reachability analysis for uncontrolled linear systems using fourier transforms. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. 35–44.

[56] Meriel von Stein and Sebastian Elbaum. 2021. Automated Environment Reduction for Debugging Robotic Systems. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3985–3991.

[57] Kenneth J Waldron and James Schmiedeler. 2016. Kinematics. In *Springer handbook of robotics*. Springer, 11–36.

[58] E. Weyuker and B. Jeng. 1991. Analyzing Partition Testing Strategies. *IEEE Trans. Software Eng.* 17 (1991), 703–711.

[59] Elaine J Weyuker. 1983. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 641–655.

[60] Elaine J. Weyuker. 1988. The evaluation of program-based software test data adequacy criteria. *Commun. ACM* 31, 6 (1988), 668–675.

[61] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. 2021. Fuzzing Mobile Robot Environments for Fast Automated Crash Detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 5417–5423.

[62] Qian Yang, J Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*. 99–103.

[63] Esen Yel, Tony X Lin, and Nicola Bezzo. 2018. Self-triggered adaptive planning and scheduling of uav operations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 7518–7524.

[64] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[65] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur)* 29, 4 (1997), 366–427.

Received 2023-02-16; accepted 2023-05-03