Paracosm: A Language and Tool for Testing Autonomous Driving Systems

Rupak Majumdar
MPI-SWS
Germany
rupak@mpi-sws.org

Aman Mathur
MPI-SWS
Germany
mathur@mpi-sws.org

Marcus Pirron
MPI-SWS
Germany
mpirron@mpi-sws.org

Laura Stegner
University of Cincinnati
Germany
stegnelm@mail.uc.edu

Damien Zufferey
MPI-SWS
Germany
zufferey@mpi-sws.org

Abstract

Systematic testing of autonomous vehicles operating in complex real-world scenarios is a difficult and expensive problem. We present Paracosm, a reactive language for writing test scenarios for autonomous driving systems. Paracosm allows users to programmatically describe complex driving situations with specific visual features, e.g., road layout in an urban environment, as well as reactive temporal behaviors of cars and pedestrians. Paracosm programs are executed on top of a game engine that provides realistic physics simulation and visual rendering. The infrastructure allows systematic exploration of the state space, both for visual features (lighting, shadows, fog) and for reactive interactions with the environment (pedestrians, other traffic).

We define a notion of test coverage for Paracosm configurations based on combinatorial testing and low dispersion sequences. Paracosm comes with an automatic test case generator that uses random sampling for discrete parameters and deterministic quasi-Monte Carlo generation for continuous parameters. Through an empirical evaluation, we demonstrate the modeling and testing capabilities of Paracosm on a suite of autonomous driving systems implemented using deep neural networks developed in research and education. We show how Paracosm can expose incorrect behaviors or degraded performance.

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

ACM Reference Format:
Rupak Majumdar, Aman Mathur, Marcus Pirron, Laura Stegner, and Damien Zufferey. 2019. Paracosm: A Language and Tool for Testing Autonomous Driving Systems. In *Proceedings of Conference.* ACM, New York, NY, USA, 14 pages.

1 Introduction

Autonomous driving systems are marvels of engineering. In the recent years, such progress has been made that the first commercial taxi service based on self-driving cars is set to launch before the end of 2018.[1] However, the complexity of developing such systems and ensuring reliability and safety of the software stack at the core of these systems is an incredibly complex task and this could well be a safety disaster waiting to happen.[2]

The software in self-driving cars combines well-defined tasks like controlling the car, i.e., trajectory planning, steering, acceleration and braking, with the often underspecified tasks related to building a semantic model of the surrounding world from raw sensor data and making decisions using this model. Unfortunately, these fuzzier tasks are no less critical to the safe operation of these systems. Therefore, end-to-end testing in realistic conditions is the only way to build confidence in the correctness of the overall system.

Running real tests is a necessary but slow and costly process. It can also be difficult to reproduce corner cases due to infrastructure or safety issues; one can neither run over an actual pedestrian just to demonstrate a failing test case, nor wait for specific seasonal weather conditions to complete all the tests. Therefore, in recent years, training and testing of autonomous systems in simulation has gained traction [17, 19, 22, 40, 49, 53, 54, 59]. Simulation reduces the cost per test but, more importantly, it gives precise control over all the parameters and makes it possible to precisely recreate corner cases.

A major limitation of current tools is the lack of programmability of test environments. Most tools today present a direct manipulation interface to set up a scenario and focus on tests relative to that scenario. In this paper, we present Paracosm, a language to *programmatically construct* reactive

[1] https://www.bloomberg.com/news/articles/2018-11-13/waymo-to-start-first-driverless-car-service-next-month
[2] https://www.ntsb.gov/investigations/AccidentReports/Reports/HWY18MH010-prelim.pdf

environments for testing autonomous vehicles in realistic settings. Like several previous projects [17, 18, 43, 49, 54], Paracosm is built on top of a high-fidelity visual environment and physics simulator provided by a game engine (Unity [55], in our case). Unlike other projects, where each virtual world needs to be pre-fabricated by a designer, Paracosm focuses on providing language support for programmatically designing *parameterized environments and test cases*. The test parameters define the environment and the behaviors of the actors. As they can be programmatically adapted, Paracosm can do systematic testing and we can describe a notion of test coverage for these test families.

Paracosm is based on a synchronous reactive programming model [8, 28, 31, 57]. Components in Paracosm, such as road segments or cars, receive streams of inputs and produce streams of outputs over time. In addition, components can also have graphical assets to describe their geometric properties for an underlying graphics rendering engine and physical properties relating to their dynamics for an underlying physics engine. For example, a car modelled in Paracosm will not only have code that reads in camera feeds and outputs steering angle or braking, but also have a textured mesh representing its shape and its position and orientation in 3D space, and a physics model for its dynamical behavior. A Paracosm configuration consists of a composition of a number of components. Using a set of system-defined components (road segments, cars, and pedestrians, etc.) combined together using the expressive operations from the underlying reactive programming model, users can set up complex driving scenarios where the environment's behavior changes over time. For example, one can build an urban road network with intersections, pedestrian and vehicular traffic, and parameterize both environment conditions (lighting, fog) and behaviors (when a pedestrian crosses a street).

Streams in the world description can be left "open" and, during testing, Paracosm will generate sequences of values for these streams. This forms the basis of an automatic testing infrastructure on which many different testing strategies from randomized to exhaustive can be tried. We formalize a notion of coverage for our setting based on *k-wise combinatorial coverage* [9, 30] for discrete variables and *dispersion* for continuous variables. Intuitively, k-wise coverage ensures that, for a programmer-specified parameter k, all possible combinations of values of any k discrete parameters are covered by tests. Low dispersion [44] ensures that there are no "large empty holes" left in the continuous parameter space. We show an automatic test generation strategy that offers high coverage based on random sampling over discrete parameters and *deterministic* quasi-Monte Carlo methods for continuous parameters [37, 44]. We provide theoretical guarantees on the number of tests to ensure coverage with high probability. Perhaps surprisingly, quasi-Monte Carlo methods provide better guarantees than pure random exploration for dispersion.

An empirical evaluation of autonomous driving models in Paracosm environments shows, through various case studies, how Paracosm can be an effective testing framework for both qualitative properties (crash) and quantitative properties (distance covered in a given time).

Our main contributions are the following. (I) We provide the first programmable and expressive mocking framework for autonomous systems interacting with the physical world. (II) We define an appropriate notion of test coverage based on combinatorial k-wise coverage in discrete space and low dispersion in continuous space. We show a test generation strategy based on random generation and quasi-Monte Carlo methods that theoretically guarantees good coverage. (III) We demonstrate empirically that our system is able to construct complex scenarios to automatically test autonomous driving agents and find incorrect behaviors or degraded performance.

2 Language through Examples

Paracosm is based on a synchronous reactive programming model [8, 28], where communication occurs through streams of values, and computation occurs through stream transformations. The core of Paracosm is a *reactive object*. Reactive objects capture geometric and graphical features of a physical object, as well as their behavior over time. Internally, reactive objects contain graphical assets (3D meshes) for their visual representation; the assets prescribe their geometric properties to an underlying graphics engine. The behavioral interface for each reactive object has a set of *input* streams and a set of *output* streams. The evolution of the world is computed in steps of fixed duration which corresponds to events in a predefined `tick` stream. For streams that correspond to physical quantities updated by the physics engine, such as position and speeds of the cars, collisions detection, etc., the appropriate events are generated by the underlying physics engine.

Input streams provide input values from the environment over time; output streams represent output values computed by the object. The object's constructor sets up the internal state of the object. An object is updated by the event triggered computations. Paracosm provides a set of assets as base classes. While users can use any new asset, typically, users would derive new behaviors but preserve the graphical assets.

We differentiate between static *environment* reactive objects and dynamic *actor* reactive objects. Environment reactive objects represent "static" components of the world, such as road segments, intersections, houses, or trees, and a special component called the *world*. Actor reactive objects represent components with "dynamic" behavior: cars, bicycles, or pedestrians. The world object is used to model features of the world such as lighting or weather conditions.

Reactive objects can be *composed* to generate complex assemblies from simple objects. The composition process can be used to connect static components structurally–such as two road segments connecting at an intersection. Composition also connects the behavior of an object to another by binding output streams to input streams. At run time, the values on that input stream of the second object are obtained from the output values of the first. Composition must respect geometric properties—the runtime system ensures that a composition maintains invariants such as no collision.

A *test configuration* in PARACOSM consists of a composition of reactive objects. A test configuration may contain reactive objects some of whose input streams are not connected yet. A *test case* replaces each such "open" input stream with a concrete input stream.

An instantiation of world and actor reactive objects gives a scene, i.e., all the visual elements present in a model of the world. A test case determines how the scene evolves over time: how the car moves, how pedestrians move, and how road conditions change over time.

It may seem surprising that we model static components as reactive objects as well. This serves two purposes. First, one can treat the number of lanes in a road segment as a constant input stream that is set by the test case. This allows writing parameterized test cases. Second, certain features of static objects can also change over time: for example, the coefficient of friction on a road segment may depend on the weather condition, and the weather condition can be a function of time.

We demonstrate the descriptive power of PARACOSM through a series of examples building up on each other. We have simplified the actual API syntax for clarity.

A Simple World. The first example sets up the world, a road segment, a car on the road segment, and a pedestrian trying to cross:

```
World(rxinout light, rxinout fog) { ... }
RoadSegment(rxin friction, const rxinout len,
  const rxinout nlanes) { ... }
AutonomousVehicle(value loc,
  rxout pos, rxout vel, ...) { ... }
Pedestrian(value start, value target,
  rxin dist, rxin car, rxin speed) { ... }

w = World(light=Const(1.0),fog=Const(0))
r = RoadSegment(friction=Const(0.8),
        len=Const(200.0), nlanes=Const(2))
v = AutonomousVehicle(loc=r.onLane(1, 10))
p = Pedestrian(start=r.onLane(r.SideWalk,100),
        target = r.onLane(-r.SideWalk,100),
        dist = Const(30), car = v.pos,
        speed= Const(4))
w.place(r, p, v)
```

The reactive objects `World`, `RoadSegment` are pre-defined in our system, and consist of pre-defined visual assets. `AutonomousVehicle` and `Pedestrian` extend pre-defined elements. The `Pedestrian` also contains code for the behavior for this test (code below). The pedestrian starts at `start` and, when the car is closer than `dist`, tries to cross toward `target`. We use some syntactic sugar: we write `rxin` and `rxout`, for input and output streams respectively. We write `rxinout` for input streams which are "passed through" and available as identical output streams as well. A `value` is a single value which can be used, for instance, to compute the initial state of the system, e.g., initial value of the output stream `pos`. We can connect streams explicitly, or by assigning them during creation of an object. As a shorthand, we can pass an entire object as a stream; the intended use is that any output stream of the object is available to the receiver. The keyword `Const` creates a constant stream.

The `World` is a special object which also deals with the graphics and physics engine. It has some predefined reactive values related to the global environment conditions like the lighting. The `place` operation also adds objects to the scene and starts the simulation. The road `r` has no given position and it is placed at the origin by default. The vehicle `v` and pedestrian `p` have an initial position relative to `r`. The call `r.onLane(lane, at)` returns a position at a specified distance on a given lane. Negative lane numbers denote the opposite direction.

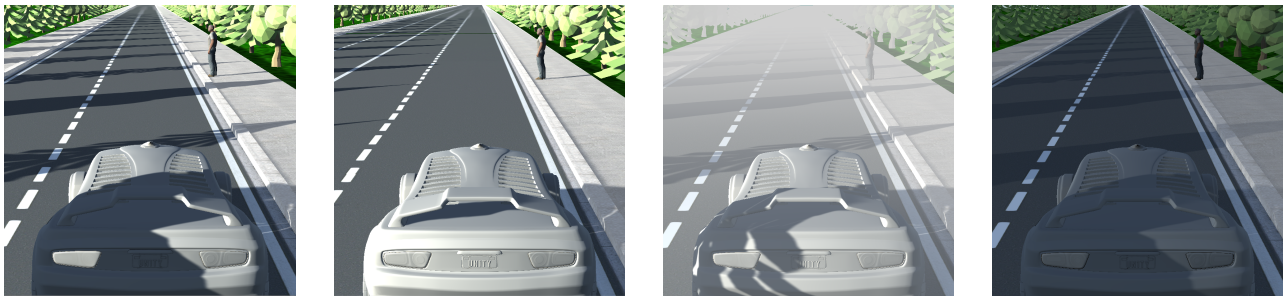
Figure 1(a) shows the representation of the world in the graphics engine. In this example, there are no open streams. Hence, the behavior is completely determined by the initial state. By setting parameters, assets, or by using derived classes, one can easily generate different worlds (see 1(b)-(d)).

A Family of Worlds. In the previous example, we used constant streams to set input streams to constant values. Instead, one can allow test cases with general streams of values but one can constrain the possible values passed into an object [47]. We do this by constraining streams to a given range. For the example above, we could define

```
light    = VarInterval(0.0, 1.0)
fog      = VarInterval(0.0, 1.0)
friction = VarInterval(0.2, 0.8)
nlanes   = VarEnum({2,4,6})
dist     = VarInterval(10, 50)
speed    = VarInterval(0.5, 10)
```

Such constraints allow test generators to change properties of the simulation, thereby leading to different test cases (see Figure 1).

Reactive Behaviors. Reactive objects often model some physical elements in the world and, therefore, need to interface with a physics and rendering engine. There are two dedicated types for this task. First, we have `Geometric` objects which have a model and support collision detection. The position



(a) Initial state **(b)** Four lanes **(c)** Moderately dense fog **(d)** Lower light intensity

Figure 1. Environment generated from Example

of geometric object is under control of the programmer. Second, the `Physical` objects extends geometric objects but their motion is computed by the physics engine.

Geometric objects need to provide a textured mesh and have a pose (position and orientation in 3D space). The world evolves in fixed time interval `dt`. We call each passing interval a `tick`. `tick` is a stream of events with unit type. Each time the simulation engine makes a step, an event is generated on the `tick` stream. The loading and initialization of models is done at object creation and stored in the appropriate field inherited from the parent class. The models used for display and collision detection may be different. While the object's mesh may be used for accurate collision detection, it is also possible to use simpler models, typically bounding boxes, on which detecting collision is computationally cheaper.

Not every model need have complex dynamics and a simple and precise way of controlling an object is sometime preferable. For example, we can build a simple model of a pedestrian crossing the road which simply updates the position of the pedestrian by a constant at each `tick`. In general, the reactive nature of objects enables complex scenarios to be built; for example, a pedestrian model can start crossing the road only when a car is a certain distance away. In the code below, we use '_' as shorthand for a lamdba expression, i.e., "`f(_)`" is the same as "`x => f(x)`".

```
Pedestrian(value start, value target, rxin
    dist, rxin car, rxin speed) extends
    Geometric {
  ... // initialization
  // generate an event when the car gets close
  trigger = car.filter( |_ - start| < dist )
  // target location reached
  done = pos.filter( _ == target )
  //walk to the target after trigger fires
  tick.skipUntil(trigger).takeUntil(done).
      foreach( ... /* walk */ )
}
```

Reactive objects whose state is also modified by the physics engine have a sightly richer interface. Their internal state is restricted to an API understood by the physics engine. The state is a collection of rigid bodies connected by joints. Each body has a mesh for rendering and physical properties like mass, moments of inertia, etc., that makes it possible to simulate the effect of forces on the object. Joints connect the multiple bodies in an object and constrain their relative positions. For instance, a simple car has five different bodies: one for the frame and four for the wheels. The wheels are attached to the body and allowed to rotate around the center. The car model can be modified to change the type of drive (front, rear or four wheel drive), centre of mass, maximum steering angle, torque, maximum speed, etc.

Invariants and Monitors. So far, we discussed static and dynamic elements that make up the world. Additionally, Paracosm provides an API to provide qualitative and quantitative temporal specifications. For instance, in our pedestrian crossing example, we want to check that there is no collision and detect that the collision was not trivially avoided because the car did not move at all.

```
// no collision
CollisionMonitor(rxin AutonomousVehicle v) {
  assert(v.collider.isEmpty())
}
// check that the car did not trivially pass
    the test by staying put
DistanceMonitor(rxin AutonomousVehicle v,
    value minD, rxout D) {
  pOld = v.pos.take(1).concat(v.pos)
  D = v.pos.zip(pOld).map( |_ - _| ).Sum()
  assert(D >= minD)
}
```

The ability to write monitors which read streams of system-generated events provides an expressive framework to write temporal properties. For example, one can encode monitors for metric and signal temporal logic specifications in the usual way [14, 26].

Sensors. Self-driving cars have a large array of sensors. In simple systems like openpilot [10], only a dash-cam is used but more complex systems mix cameras, LiDARs, radars, and GPS. Paracosm can emulate common types of sensors.

Sensors produce streams of data which can be used by the tested system. In Figure 2, we show data coming from a few sensors. Normal cameras can be mounted at specific points on the vehicle and it is possible to vary parameters like the focal length. The camera can be ideal or also include imperfections like blur or noise. During the rendering process for RGB images we can also extract depth map (Fig. 2b) to cheaply emulate LiDAR. It is also possible to use ray casting techniques to more accurately simulate a LiDAR but the computational cost is significantly higher.

In general, it is possible to provide new sensors which are not supported out-of-the-box by PARACOSM. However, new types of sensors need to be implemented directly on top of the rendering engine's API.

System Under Test (SUT). Autonomous systems naturally fit reactive programming models. They consume sensor input streams and produce actuator streams for the car model. Our default vehicle model has two control inputs: the throttle (*acceleration* $\in [-1, 1]$) and the steering angle (*steering_angle* $\in [-1, 1]$). The values are normalized. The actual maximum steering angle and acceleration/braking depends on the car.

Let us be more specific. Suppose we want to test the implementation of a controller for an autonomous vehicle. Suppose the controller is implemented using a neural network that takes a video stream, frame-by-frame, and outputs the acceleration and the steering angle. We design the tests for this controller in PARACOSM in the following way. We encapsulate the controller code inside an `AutonomousVehicle` object. The `AutonomousVehicle` object subclasses `Physical`, and contains both geometric information about the shape and physical dimensions of the car, the position and properties of the camera, as well as a vehicle model for the physics of its dynamics. The object has an input stream for the video feed, which is provided by the rendering engine using its knowledge about the geometry of the scene. Each new image coming from the video feed is forwarded to the neural network. The neural network returns the actuation values for the acceleration and the steering angle.[3] The values from this output stream goes to the physics engine, which calculates the new position and orientation of the vehicle.

More Complex Worlds. Until now, our tests had a very simple environment. However, constructing a world in a procedural manner allows a PARACOSM's user to quickly create large environment to run longer tests. Road segments can be composed into more complicated road networks. PARACOSM provides abstractions to generate such compositions with a `connect` operation between road elements. Furthermore, PARACOSM contains more complex road elements such as cross-intersections, T-intersections, and roundabouts. Connections can be established using a `connect` methods that takes physical connection identifiers and road elements as arguments. The connections are directed in order to compute the positions of the elements. One road element becomes the parent and it's children are positioned relative its position and the specified connection points. After an object is connected a new *composite* road element which encapsulates all road elements along with requisite transformations (rotations and translations) is returned. The following example shows how road segments can be connected into a road network.

```
// Create a T-intersection and three straight
    road elements (east, south, west)
t = TIntersection(numLane = Const(4),
                  position = Const(origin))
e = RoadSegment(len=Const(50),nlanes=Const(4))
s = RoadSegment(len=Const(50),nlanes=Const(4))
w = RoadSegment(len=Const(20),nlanes=Const(4))
// connect and get new composite object
net = t.connect((t.ONE, e, e.TWO),
                (t.TWO, s, s.ONE),
                (t.THREE, w, w.ONE))
```

Connecting elements has two purposes. First, it allows PARACOSM to perform sanity checks like the positions of road elements. Second, it creates an overlay graph of the road networks which can easily be followed by environment controlled cars not under test. When a road network is created, the runtime system of PARACOSM checks that compositions of road elements and intersections are topologically and geometrically valid. All road elements must be connected to a matching road correctly (for example, a 2-lane road segment cannot be connected to a 6-lane road segment directly), there are no spatial overlaps between road segments, and the positions of the connection points match.

In general, PARACOSM inherits all programming features of the underlying imperative programming model as well as reactive programming with streams à la functional reactive programming. Thus, one can build complex urban settings through composition and iteration. For instance, the grid world shown in Figure 3 was created by iterating a simple road network.

3 Testing PARACOSM Worlds

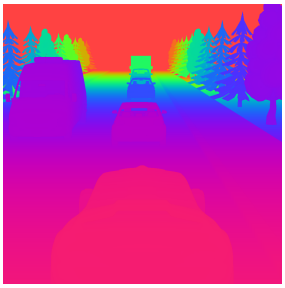
3.1 Tests and Coverage

Worlds built using the PARACOSM API directly describe a parameterized family of tests; the parameterization is provided by the reactive streams left "open" in the world. The testing framework of PARACOSM allows the user to specify different strategies to generate input streams for both the static and the dynamic reactive objects in the world.

Test Cases. A *test* of *duration* T executes a configuration of reactive objects by providing inputs to every open input stream in the configuration for T ticks. The inputs for each

[3] In our implementation, we also allow the neural network to run in its own process. The processing uses remote procedure calls.

(a) RGB view



(b) Depth map. The distance is projected onto the hue angle in the HSV color space.

(c) Car mounted camera with long focal length (small field of view)

(d) Car mounted camera with small focal length (large field of view)

Figure 2. Simulating sensors

Figure 3. A large grid world with several connected road segments and cross-intersections.

stream must satisfy `const` parameters and respect the range constraints from `VarInterval` and `VarEnum`. The runtime system manages the scheduling of inputs and pushing input streams to the reactive objects.

Suppose that In denotes the set of all input streams, and let $\mathsf{In} = \mathsf{In}_D \cup \mathsf{In}_C$ denote the partition of In into *discrete* streams and *continuous* streams respectively. Discrete streams take their value over a finite, discrete range; for example, the color of a car, the number of lanes in a road segment, or the position of the next pedestrian (left or right) are discrete streams. Continuous streams take their values in a continuous (bounded) interval. For example, the fog density or light intensity, or the speed of an AI vehicle in the environment are examples of continuous streams.

A test case must provide discrete values to each stream in In_D and continuous values to each stream in In_C at each step. Thus, a test of duration T can be seen as a vector over $(\mathbb{N}^{|\mathsf{In}_D|} \times \mathbb{R}^{|\mathsf{In}_C|})^T$: for each time index $t = 0, \ldots, T-1$, the vector gives values to each stream in In. The test is valid if each input satisfies the constraints on the stream. For example, a constant stream will have the same value for each time step, and a stream with a constraint $[0, 2]$ will take values in that range at each time step.

Coverage. As usual in testing, one can use programmer-provided invariants and monitors as *test oracles*. In the setting of autonomous vehicles testing, however, one often wants to explore the state space of a parameterized world to check "how well" an autonomous vehicle works under various situations, both qualitatively and quantitatively. Thus, we now introduce a notion of coverage. Instead of structural coverage criteria such as line or branch coverage, our goal is to cover the parameter space "uniformly."

In what follows, for simplicity of notation, we assume that all discrete streams take values $\{0, 1\}$, and all continuous streams take values in the real interval $[0, 1]$. Any input stream over bounded intervals—discrete or continuous—can be encoded into such streams. Further, in test vectors, we only keep a single co-ordinate for constant streams instead of a co-ordinate for every time step.

For discrete streams, there are finitely many tests, since each co-ordinate is Boolean and there is a fixed number of co-ordinates. One can define the coverage as the fraction of the number of vectors tested to the total number of vectors. Unfortunately, the total number of vectors is very high: if each stream is constant, then there are already 2^n tests for n streams. Instead, we consider the notion of *k-wise testing* from combinatorial testing [30]. In k-wise testing, we fix a parameter k and ask that every interaction between every k elements is tested.

Let us be more precise. Suppose that a test vector has N co-ordinates, where each co-ordinate can get the value 0 or 1. A set of tests A is a *k-wise covering family* if for every subset $\{i_1, i_2, \ldots, i_k\} \subseteq \{1, \ldots, N\}$ of co-ordinates and every vector $v \in \{0, 1\}^k$, there is a test $t \in A$ whose restriction to the co-ordinates $i_1, \ldots, i_k$ is precisely v.

Example 3.1. Consider a configuration with constant input streams corresponding to the number of lanes on the road (possible values $\{2, 4\}$), the direction of n vehicles along the

road (toward a test vehicle or in the same direction as the test vehicle), and the color of the vehicles (possible values red and green). There are $2 \times 2^n \times 2^n$ possible tests. For 2-wise testing, we want to ensure every pair of parameters is covered by some test. Naively, there are at most $2n(2n+1)$ test cases, but the theory of covering arrays can be used to find much smaller test sets. For example, for $n = 3$, just 6 tests are enough: $(2, 011, GGG)$, $(2, 101, GRR)$, $(2, 010, RR*)$, $(4, 001, RGR)$, $(4, 110, GGR)$, and $(4, 100, RRG)$. Here, the first column denotes the number of lanes, the next denotes the direction of the three cars, and the next denotes their colors. An "$*$" denotes either color can be used in the test. □

For continuous streams, the situation is more complex: since any continuous interval has infinitely many points, each one corresponding to a different test case, we cannot directly define coverage as a ratio (the denominator will be infinite). Instead, we define coverage using the notion of *dispersion* [37, 44].

Recall that we assume a (continuous) test is a vector in $[0,1]^N$: each entry is picked from the interval $[0,1]$ and there are N co-ordinates. Dispersion over $[0,1]^N$ can be defined relative to sets of neighborhoods, such as N-dimensional balls or axis-parallel rectangles. Let us define $\mathcal{B}$ to be the family of N-dimensional axis-parallel rectangles in $[0,1]^N$, our results also hold for other notions of neighborhoods such as balls or ellipsoids. For a neighborhood $B \in \mathcal{B}$, let $vol(B)$ denote the volume of B.

Given a set $A \subseteq [0,1]^N$ of tests, we define the *dispersion* as the largest volume neighborhood in $\mathcal{B}$ without any test:

$$\text{dispersion}(A) = \sup\{\text{vol}(B) \mid B \in \mathcal{B} \text{ and } A \cap B = \emptyset\}$$

Intuitively, dispersion measures the largest empty space left by a set of tests. We use dispersion as a measure of coverage for continuous vectors: the lower the dispersion, the better coverage for a configuration.[4]

Let us summarize. Suppose that a test vector consists of N_D discrete co-ordinates and N_C continuous co-ordinates; that is, a test is a vector (t_D, t_C) in $\{0,1\}^{N_D} \times [0,1]^{N_C}$. We say a set of tests A is (k, ε)-*covering* if

1. for each set of k co-ordinates $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, N_D\}$ and each vector $v \in \{0,1\}^k$, there is a test $(t_D, t_C) \in \{0,1\}^{N_D} \times [0,1]^{N_C}$ such that the restriction of t_D to the co-ordinates $i_1, \ldots, i_k$ is v; and
2. for each $(t_D, t_C) \in A$, the set $\{t_C \mid (t_D, t_C) \in A\}$ has dispersion at most ϵ.

[4] The notion of *discrepancy* is closely related to dispersion. The discrepancy of a set A of tests is defined as

$$\text{discr}(A) = \sup\left\{\left|\frac{|A \cap B|}{|A|} - \text{vol}(B)\right| \mid B \in \mathcal{B}\right\}$$

By focusing on those neighborhoods R for which $A \cap R = \emptyset$, one can show that $\text{dispersion}(A) \leq \text{discr}(A)$. Thus, we can alternatively use discrepancy as a coverage.

Our goal is to automatically generate (k, ϵ)-covering test sets for programmer-specified k and ϵ.

3.2 Test Generation

We now show an automatic test generation algorithm based on random sampling from discrete parameters and deterministic quasi-Monte Carlo generation for continuous parameters. Our test generation strategy has two parts. First, we construct a k-wise covering family for the discrete part of the test. Then, for each (discrete) vector generated in the family, we generate a set of (continuous) vectors of low dispersion.

k-Wise Covering Family. One can use explicit construction results from combinatorial testing to generate k-wise covering families [9]. However, a simple way to generate such families with high probability is through random testing. The proof is by the probabilistic method [1] (see also [34]).

Proposition 3.2. *Let A be a set of $2^k(k \log N - \log \delta)$ uniformly randomly generated $\{0,1\}^N$ vectors. Then A is a k-wise covering family with probability at least $1 - \delta$.*

Low Dispersion Sequences. It is tempting to think that uniformly generating vectors from $[0,1]^N$ would similarly give low dispersion sequences. Indeed, as the number of tests goes to infinity, the set of uniformly randomly generated tests has dispersion 0 almost surely. However, we do not plan to test with an infinite number of tests. When we fix the number of tests, it is well known that uniform random sampling can lead to high dispersion [37, 44]; in fact, one can show that the dispersion of n uniformly randomly generated tests grows asymptotically as $O((\log \log n/n)^{\frac{1}{2}})$ almost surely.

Our test generation strategy will be based on *deterministic quasi-Monte Carlo sequences*, which have much better dispersion properties, asymptotically of the order of $O(1/n)$, than the dispersion behavior of uniformly random tests.

There are many different algorithms for generating quasi-Monte Carlo sequences deterministically (see, e.g., [37, 44]). We shall use *Halton sequences* in our experiments. Fix an integer $q \geq 2$. For an integer j, suppose the base-q representation of j is $j = \sum_{i \geq 0} b_j q^j$. Define the function

$$\gamma_q(j) = \sum_{i \geq 0} b_i q^{-i-1}$$

Let $q_1, \ldots, q_N \geq 2$ be co-prime integers. The *Halton sequence* is the set of points

$$E_n = \{(\gamma_{q_1}(j), \gamma_{q_2}(j), \ldots, \gamma_{q_N}(j)) \mid 0 \leq j < n\}$$

The following estimate shows that the Halton sequence has dispersion asymptotically better than uniform random sequences; in fact, the dispersion decreases linearly with the number of tests.

Proposition 3.3 ([44]). $\text{dispersion}(E_n) = O(\frac{1}{n})$.

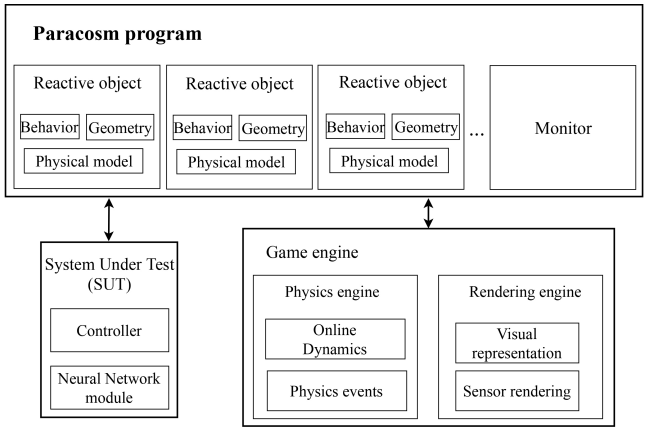


Figure 4. Architecture of a simulation in Paracosm

Thus, for a given ϵ, one needs to generate $O(\frac{1}{\epsilon})$ tests. In Section 4, we show an empirical comparison between uniformly random and quasi-Monte Carlo test generation.

3.3 Reducing the Dimension

In general, one can expect that the dimension N of a test case is extremely large, and that any test generation technique is fruitless. For example, consider that a video feed can get 1000×600 pixels per tick, where each pixel is an RGB value (total 256^3 values). Thus, a single test case for $T = 100$ ticks is already $256^{3\times1000\times600\times100}$ dimensional, and one cannot hope to cover even a minuscule portion of the space of tests.

However, in many testing situations, we sample from a low-dimensional feature space embedded in the high dimensional sensor space [18, 19]: instead of covering the space of pixels, we focus on a set of features (e.g., the presence or position of an object in the environment over time). Further, one can use parameterized representations [15]: instead of modeling fog over time a 1000-dimensional vector, where each dimension is independently picked, we can pick a low-degree polynomial whose evaluation gives the fog intensity, and only test over the coefficient ranges of the polynomial.

4 Implementation and Tests

4.1 Runtime System and Implementation

Paracosm uses the Unity game engine [55] to render scenes and to simulate physics. The physics engine in Unity is PhysX [11]. The reactive objects are build on top of UniRx [29]. Like other simulation environments [17, 49], the core reason for using a modern game engine is visually realistic rendering. In addition to this, a game engine manages global positions and orientations of 3D objects and also offers abstractions for generating realistic environmental effects. Encoding behaviours for actors, managing 3D assets, and dynamic checks (system inactivity, collisions, etc.) are implemented using the game engine interface.

Figure 4 shows the architecture of Paracosm's runtime. A simulation proceeds as follows. First, the program is read and requisite reactive components are instantiated. This involves initialization and placement of their 3D mesh (if they have one). The connection to the *SUT* is also established in case it runs on its own process. In our experiments, we test a number of driving systems based on neural networks. We use a modified version of Udacity's self-driving car simulator [54] to interface with the neural networks we test.

For practical purposes, game engines typically treat physics and rendering pipelines separately. The physics engine is informed about the physical components (subclasses of `Physical`). Physics simulations in Paracosm work on rigid bodies, i.e., solid bodies with an assumption of no deformation. This enables accurate detection of collision, but not deformations caused by the collision. This is an acceptable limitation as a collision typically marks the end of a failure case.

4.2 Case studies

The previous sections discuss Paracosm's programming interface and automatic test generation algorithm. We now demonstrate two case studies that illustrate how these can be utilized for authoring a wide array of interesting test scenarios. The first study (Section 4.2.1) covers tests on pre-trained machine-learnt models built for object segmentation, specifically, for road detection and vehicle detection. The second study (Section 4.2.2) discusses tests on an autonomous vehicle being controlled by a convolutional neural network.

4.2.1 Testing Pre-Trained Networks

Most autonomous vehicles and advanced driver-assistance systems (ADAS) have on-board components for computer vision tasks like road detection, traffic light and traffic sign classification, vehicle detection and optical flow. Using Paracosm's programming interface, we created a 6-lane road network with a cross intersection and heterogeneously spread-out traffic. We tested two object detection systems: road detection, and vehicle detection.

All the systems we test in this section are already trained. We took existing code for pre-trained networks and integrated it in Paracosm test scenarios.

Road detection. A road detection system takes RGB images as input and outputs those pixels that are estimated to be a part of the road. We use a network[5] based on the fully convolutional neural network from Simonyan and Zisserman [50], trained with the KITTI road segmentation data set [23].

In this experiment, we vary the focal length of the car's top-mounted camera to see its effect on road segmentation. Only the focal length change;, the rest of the scene is kept the same. We varied the focal length between 10 and 50mm (wide to standard angle) and measured how much of the road was

[5]We use the code from https://github.com/ndrplz/self-driving-car/tree/master/project_12_road_segmentation

(a) A short focal length (10mm) results in poor road detection.

(b) A longer focal length (34mm) results in better road detection. The red area depicts a falsely detected road segment.

Figure 5. Sample output from the road detection system.

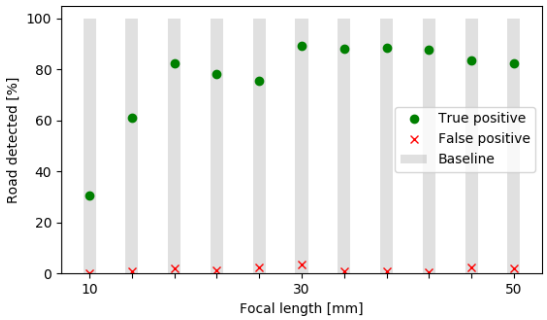


Figure 6. Influence of focal length on the road detection. The normalized baseline is obtained by manual annotation. A true positive indicates the percentage of the baseline that was recognized as road. A false positive shows the percentage of non-road (compared to baseline) identified as road.

correctly identified and how much of the scene was falsely identified as road. For each value, we manually tag the road in the image and compare the output of the network to this. Correctly detected road has to be detected in both images; falsely detected means that the neural network marked a part of the image being part of the road while manual labeling did not mark that part. Figure 5 shows two example images and Figure 6 shows the test's results.

Vehicle detection. A vehicle detection system takes RGB images as input and outputs bounding boxes around pixels that correspond to vehicles. Figure 8 shows an example of a vehicle detection system's output. To detect other vehicles in the vicinity of the autonomous car, we used (i) the single shot multibox detector (SSD)[6], a single deep neural network [32], trained with the Pascal Object Recognition Database Collection, (ii) YOLO [42], a neural network based approach

[6]SSD code from https://github.com/ndrplz/self-driving-car/tree/master/project_5_vehicle_detection

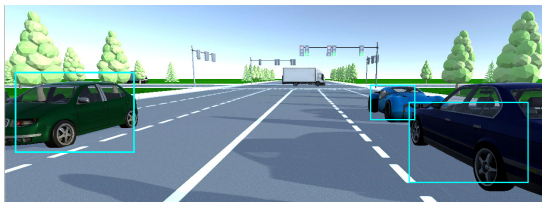


Figure 7. Sample output from the vehicle detection system.

which frames object detection as a regression problem, and (iii) a non-neural network based approach based on History Oriented Gradients (HOG) [12], trained on the GTI Vehicle image database [4]. HOG methods were first used for human detection and then adapted to cars.[7]

In this experiment, the car is driven in a straight line at constant speed through traffic and the camera images are sent to the vehicle detection system. Using the same road network as in the example of the road segmentation experiment, the car follows the innermost lane through the traffic into the crossing, on which a white truck is passing from left to right. Every 10 meters, we save the output these systems. We manually identify the total number of vehicles to get a baseline. Then we inspect the output of the vehicle detection systems and count the number of vehicles correctly identified and the false positives, e.g., road features identified as a car.

Figure 8 summarizes the results. *Baseline* denotes the total number of cars visible, *true positive* the correctly identified vehicles, and *false positive* the erroneously detected ones.

We observe that overall, the systems we used are not able to identify vehicles that are too far away, explaining why the detection is so poor in the beginning. More worrying, no system was able to detect the truck in the middle of the crossing. As the training data consists mostly of images where all the vehicle follow the same road, our hypotheses is that the systems learned features corresponding to the front or back of the vehicles. This experiment shows the usefulness of testing pre-trained networks under different driving conditions.

4.2.2 Driving Behaviour

We now test an end-to-end system, i.e., a system which takes sensor values as input and transforms these into steering or throttle commands. Paracosm's architecture enables a qualitative and quantitative analysis of such dynamic closed-loop behavior. We run tests over time to catch situations in which decisions made by a controller can have long-term consequences. In an optimistic scenario, if the output of the SUT changes abruptly due to some error in the vision system, it may be harmless if it is just for one frame as it gets smoothed out by correct behavior in surrounding frames.

[7]YOLO and HOG code from https://github.com/JunshengFu/vehicle-detection.

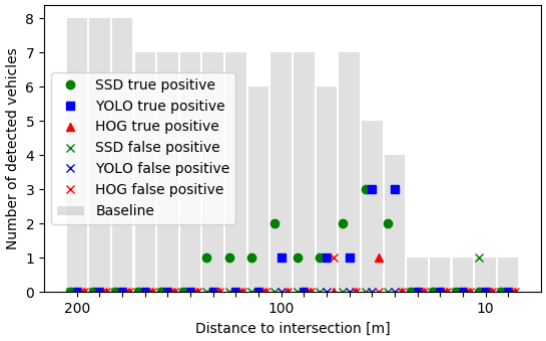


Figure 8. Vehicle detection rates for SSD, YOLO, and HOG. The light grey bar shows ground truth.

However, it may also be possible that a small variation pushes the system outside of it normal operating zone. Running longer tests with closed-loop behavior allows us to test these effects.

For this case study, we had to train our own system because we could not find existing systems that performed well enough in our tests. We used NVIDIA's behavioral cloning framework [7] due to its popularity and ease-of-use. It uses a deep convolutional neural network (CNN) to solve a supervised regression problem. Given an RGB image, it returns the corresponding steering or throttle control to be applied at that instant.[8] The goal of the case study is not to claim that we have trained a world-class autonomous driving system, or to belittle a specific implementation, but to show the variety of closed-loop evaluations possible in Paracosm.

Training Procedure. The CNN needs to be trained using annotated data which consists of RGB images and ideal steering/control for each image. We obtain this by driving around in 3D environments generated by Paracosm programs. Images and steering/throttle control values in the run are recorded and used for training.

Dynamic Pedestrian Behavior The autonomous car is trained to stop for pedestrians crossing the road at a set walking speed (1m/s). For the experiment, we parameterize both, the instant at which a pedestrian crosses the road and their walking speed to look for interesting failure cases. We then sample from the parameter state space using both random and low-dispersion (Halton) sampling. The automated testing procedure reveals that certain perturbations lead to collisions. Unsurprisingly, we observe that the faster the pedestrian is, the more collisions there are. However, there is no linear boundary that forms a "quick explanation". The results are summarized in Figure 9 and demonstrates the advantage of low-dispersion sampling over random sampling. The samples are more spread out for the Halton sequence (low-dispersion). This difference is more pronounced for smaller number of samples.

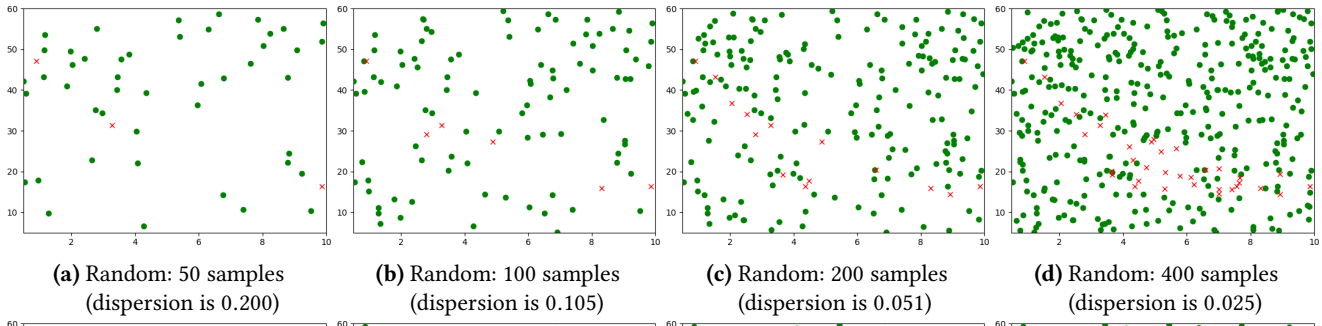
Behavior under Changing Environmental Settings. As discussed in Section 2, reactive variables can be used to parameterize environment settings so as to describe a large class of configurations. To demonstrate this, we train a model at a fixed light intensity and no fog. In this scenario, the car is behind a slow car and, therefore, may need to brake to avoid a collision. We analyse the car's performance when the environment condition varies by reporting the distance covered, and whether a collision happened. Each test lasts 15 seconds. We generate test parameters using the Halton sequence. The results are aggregated in Figure 10a. The car performs best around the parameter values used during training. The distance that the car covers drops-off fairly quickly as fog density increases and perturbations to light intensity often lead to scenarios with collision.

Next, we demonstrate an experiment with test scenarios over a longer time horizon. We created a loop by connecting equally sized straight road segments with T-intersections and trained the car to drive around in the loop indefinitely (always making a right turn). The training data corresponds to around 100 seconds of driving with the front facing camera producing 10 images per second. We parameterize fog density and measure the time to failure, i.e., time until the first collision. A time-out of 200 seconds is set. We observe that the steering angle is affected by the fog. Surprisingly, even though we train the car with no fog, it understeered in low fog conditions, then steered properly for moderate fog. Finally, in high fog conditions, the car first understeered then oversteered.

Behavior with Different 3D Components. We now discuss some observations made when training for one of the previous experiments (Figure 10a). For these experiments, the autonomous car is trained to follow a red lead car driving in front on a two-lane road. Under tests with ideal conditions, we observe that the autonomous car is able to follow the lead car while maintaining a safe distance in-between. We do two tests to try to understand what the neural network "learns" from the training images.

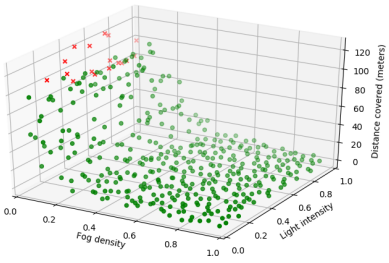
For the first experiment, we have another red car, similar to the lead car come from the other direction. If the system learned to follow the lead car, this new car should have no effect on the behavior of our autonomous vehicle. However, we observe that the autonomous car picks-up on the car coming from the other direction and applies brakes when it is close enough, essentially getting confused between the lead car and this car coming from the opposite direction. The speed over time is plotted for both, the ideal and problematic case in Figure 10c. For the second experiment, we change the body color of the lead car from red to white. As the autonomous car was only trained on a red lead car, it fails to recognize the lead car and crashes into it. These two

[8] We use the implementation at https://github.com/naokishibuya/car-behavioral-cloning.

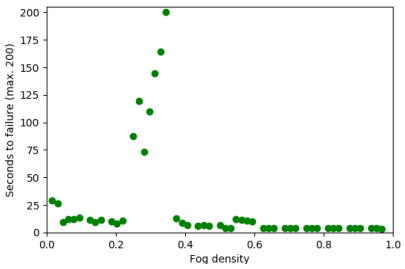


(a) Random: 50 samples (dispersion is 0.200)
(b) Random: 100 samples (dispersion is 0.105)
(c) Random: 200 samples (dispersion is 0.051)
(d) Random: 400 samples (dispersion is 0.025)
(e) Halton: 50 samples (dispersion is 0.083)
(f) Halton: 100 samples (dispersion is 0.041)
(g) Halton: 200 samples (dispersion is 0.029)
(h) Halton: 400 samples (dispersion is 0.011)

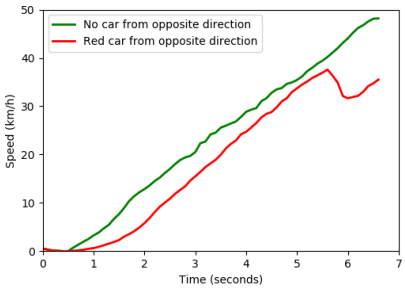
Figure 9. Comparison of random and Halton sampling. The X axis is the walking speed of the pedestrian (0.5 to 10 meters per second). The Y axis is the distance from the car when the pedestrian starts crossing (5 to 60 metres). When the car collides with the pedestrian, we mark the scenarios with a red cross.



(a) Distance covered in changing fog and light conditions tested with 400 iterations of the Halton sequence. Green dots and red crosses denote the absence or presence of a collision. The car is trained with a fog density of 0 and light intensity of 0.5.



(b) Fog density vs time to collision when driving in a loop. The car is trained at 0 fog. A value of 200 seconds means no collision until timeout.



(c) Speed over time of car trained to follow a red car in the presence and absence of another red car coming from the opposite direction. When the incoming car is close, the autonomous car brakes. Speed is picked up again once this car passes.

Figure 10. Experiences on the effect of environment conditions.

experiments point to the network learning that the throttle is related to the amount of red pixels in the image rather that recognizing a car in front.

5 Related Work

Reactive Programming Models. PARACOSM's programming model follows a synchronous reactive style for dataflows, similar to functional reactive programming (FRP) [57] or to synchronous dataflow languages like Lustre [8]. FRP has been applied to control robotic and embedded systems [28], including car [21]. In PARACOSM, rather than using reactive programming to control the SUT, we use reactive elements to build and control the world around the system under test. Our programming model shares elements found in automata-based models like reactive modules [2]. In contrast to these languages, PARACOSM natively supports geometric and physical properties of components and integrates with a game engine. Reactive programming has gained popularity in recent years as a way to build asynchronous distributed applications [31]. Our model is synchronous because we wish to test physical systems with a global time.

Traditionally, test-driven software development paradigms [5] have advocated testing and mocking frameworks to test software early and often. Mocking frameworks and mock objects [33, 36] allow the programmer to test a piece of code against an API specification, even when the implementation of the API is not available. Unfortunately, current mocking frameworks provide rudimentary support for complex environment interactions over time. Typically, mock objects are stubs providing outputs to explicitly provided lists of inputs of simple types, with little functionality of the actual code. Thus, they fall short of providing a rich environment for autonomous driving. Paracosm can be seen as a mocking framework for reactive, physical systems embedded in the 3D world. Our notion of constraining streams is inspired by work on declarative mocking [47].

Testing for Cyber-Physical Systems. There is a large body of work in automated test generation tools for cyber-physical systems through heuristic search of a high-dimensional continuous state space. While much of this work has focused on low-level controller interfaces [3, 13, 15, 16, 20, 48] rather than at the system level, specification and test generation techniques arising from this work—for example, the use of metric and signal temporal logics or search heuristics—can be adapted to our setting. Surprisingly, the notion of dispersion as a coverage criterion has not been stated before.

More recently, test generation tools have started to target autonomous systems with machine learning components, under a simulation-based semantic testing framework similar to ours [18, 19, 53]. In most of this work, an underlying visual scenario is fixed by hand, and test generation explores plausible perturbations to the nominal scenario to detect faulty system behavior. Such an adversarial analysis is shown to be preferable to the application of random noise on the input vector. Moreover, a simulation-based approach can filter benign misclassifications of the machine learning component from misclasifications that actually lead to bad system behavior. Our work extends this line of work by providing an expressive language to write parameterized tests. For example, instead of manually specifying the time and speed at which a pedestrian crosses a road, it is possible to have these variables dependent on the behavior of the autonomous car.

There is some recent work on interchange formats for test scenarios for autonomous driving [56]. The testing toolkit includes a set of visual building blocks for road elements similar to the building blocks we use to make the environment. These tools are aimed at exchanging information between humans. In contrast, Paracosm provides parametric test scenarios readily executable in a simulated environement.

To address the problem of high time and infrastructure costs of testing autonomous systems, several robotics simulators have been developed. The most popular is Gazebo [22] for the ROS [41] robotics framework. It offers a modular and extensible architecture, however falls behind on visual realism and complexity of environments that can be generated with it. To counter this, game engines have been used. A popular example is TORCS [59] and the Grand Theft Auto V environment, modified for training or testing [46]. Simulators such as CARLA [17] and AirSim [49] use modern game engines and support creation of realistic urban environments. Though these simulators enable visually realistic simulations and enable detection of infractions such as collisions, the environments themselves are rather difficult to design. Designing a custom environment involves manual placement of road segments, buildings, and actors (as well as their properties). Performing many tests is therefore time-consuming and difficult. While these systems and Paracosm share the same aims and much of the same infrastructure, Paracosm focuses on procedural generation and testing of complex scenarios.

Adversarial Examples. Adversarial examples for neural networks [25, 51] introduce perturbations to inputs that cause a classifier to classify "perceptually identical" inputs differently. Much work has focused on finding adversarial examples in the context of autonomous driving as well as on training a network to be robust to perturbations [6, 24, 35, 38, 58]. Tools such as DeepXplore [39] and DeepTest [52] define a notion of coverage for neural networks based on the number of neurons activated during the tests over the total number of neurons in the network. They use gradient descent and random test generation to find unsafe corner cases and attain high coverage. However, the evaluations of these techniques focus mostly on individual classification tasks and apply 2D transformations on images. Paracosm also considers the closed loop behavior of the system and our parameters directly change the world rather than applying transformations post facto. For example, we could observe, over time, that certain vehicles were never detected.

Alternately, there are recent techniques to verify controllers implemented as neural networks through constraint solving or abstract interpretation [24, 27, 45, 58]. While these tools do not focus on the problem of autonomous driving, their underlying techniques could be combined in the test generation phase for Paracosm. For example, some of our case studies in Section 4 track the performance of the controller against errors in the perceptual system over time.

6 Future Work and Conclusion

Deploying autonomous systems like self-driving cars in urban environments raises many challenges about making sure these systems are safe. The complex software stack includes the processing of sensor data, building a semantic model of the surrounding world, decision making, trajectory planning, and control of the car. The end-to-end testing of such systems requires the creation and simulation of a whole world. Furthermore, each test may require a sightly different world. Paracosm tackles these problems by (1) enabling

the construction of a wide range of scenarios, with precise control over elements like layout of the road, physical and visual properties of objects, and the behaviors of actors in the system, and (2) using quasi-random testing to get a good coverage over the large parametric test space.

In our evaluation, we showed that Paracosm can efficiently find erroneous behaviors in existing systems implemented using neural networks. While finding errors in sensing can be done with only a few static images, we show that Paracosm also enables the creation of longer test scenarios which exercise the controller's feedback on the environment, e.g., physics of the car.

In the future, we plan to extend Paracosm's testing infrastructure to also help training deep neural network based systems. The amount and quality of training data is critical in this step. For instance, we showed that a small variation in the camera model results in widely different results for road segmentation. However, generating data is a time consuming and expensive task. We plan to record data when a user manually drives within a parameterized Paracosm environement. The data produced during this initial run can serve as the basis for generating a large amount of new data by varying the parameters which should not impact the car's behavior. For instance, we can vary the color of other cars, positions of pedetrians who are not crossing, or even the light conditions and sensor properties (within reasonable limits). The new data can be generated completely automatically to augment the learning data set.

References

[1] Noga Alon and Joel H. Spencer. 2004. *The Probabilistic Method.* Wiley.

[2] Rajeev Alur and Thomas A. Henzinger. 1999. Reactive Modules. *Formal Methods in System Design* 15, 1 (1999), 7–48. https://doi.org/10.1023/A:1008739929481

[3] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. 2011. S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems. In *TACAS 11 (Lecture Notes in Computer Science)*, Vol. 6605. Springer, 254–257.

[4] Jon Arróspide, Luis Salgado, and Marcos Nieto. 2012. Video analysis-based vehicle detection and tracking using an MCMC sampling framework. *EURASIP Journal on Advances in Signal Processing* 2012, 1 (06 Jan 2012), 2. https://doi.org/10.1186/1687-6180-2012-2

[5] Kent L. Beck. 2002. *Test Driven Development: By Example.* Addison-Wesley Professional.

[6] Arjun Nitin Bhagoji, Warren He, Bo Li, and Dawn Song. 2017. Exploring the Space of Black-box Attacks on Deep Neural Networks. *CoRR* abs/1712.09491 (2017). arXiv:1712.09491 http://arxiv.org/abs/1712.09491

[7] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016).

[8] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. 178–188.

[9] Charles J. Colbourn. 2004. Combinatorial aspects of covering arrays. *Le Matematiche* 59, 1,2 (2004), 125–172. https://lematematiche.dmi.unict.it/index.php/lematematiche/article/view/166

[10] comma.ai. 2016. openpilot: open source driving agent. (2016). https://github.com/commaai/openpilot Accessed: 2018-11-13.

[11] NVIDIA Coporation. 2008. PhysX. (2008). https://developer.nvidia.com/gameworks-physx-overview Accessed: 2018-11-13.

[12] N. Dalal and B. Triggs. 2005. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, Vol. 1. 886–893 vol. 1. https://doi.org/10.1109/CVPR.2005.177

[13] Jyotirmoy Deshmukh, Xiaoqing Jin, James Kapinski, and Oded Maler. 2015. Stochastic local search for falsification of hybrid systems. In *ATVA*. Springer, 500–517.

[14] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods in System Design* 51, 1 (2017), 5–30. https://doi.org/10.1007/s10703-017-0286-7

[15] Jyotirmoy V. Deshmukh, Marko Horvat, Xiaoqing Jin, Rupak Majumdar, and Vinayak S. Prabhu. 2017. Testing Cyber-Physical Systems through Bayesian Optimization. *ACM Trans. Embedded Comput. Syst.* 16, 5 (2017), 170:1–170:18. https://doi.org/10.1145/3126521

[16] Alexandre Donzé. 2010. *Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems.* Springer, 167–170.

[17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[18] Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. 2017. Compositional Falsification of Cyber-Physical Systems with Machine Learning Components. In *NASA Formal Methods - 9th International Symposium, NFM 2017 (Lecture Notes in Computer Science)*, Vol. 10227. Springer, 357–372.

[19] Tommaso Dreossi, Somesh Jha, and Sanjit A. Seshia. 2018. Semantic Adversarial Deep Learning. 10981 (2018), 3–26. https://doi.org/10.1007/978-3-319-96145-3_1

[20] G.E. Fainekos. 2015. Automotive control design bug-finding with the S-TaLiRo tool. In *ACC 2015*. 4096.

[21] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. 2017. Vehicle Platooning Simulations with Functional Reactive Programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017*. ACM, 43–47. https://doi.org/10.1145/3055378.3055385

[22] Open Source Robotics Foundation. 2017. Vehicle simulation in Gazebo. (2017). http://gazebosim.org/blog/vehicle%20simulation Accessed: 2018-10-18.

[23] Jannik Fritsch, Tobias Kuehnl, and Andreas Geiger. 2013. A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms. In *International Conference on Intelligent Transportation Systems (ITSC)*.

[24] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy, S&P 2018*. IEEE, 3–18.

[25] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. *CoRR* abs/1412.6572 (2014). arXiv:1412.6572 http://arxiv.org/abs/1412.6572

[26] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. 2014. Online Monitoring of Metric Temporal Logic. In *Runtime Verification RV 2014 (Lecture Notes in Computer Science)*, Vol. 8734. Springer, 178–192.

[27] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*,

Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10426. Springer, 3–29. https://doi.org/10.1007/978-3-319-63387-9_1

[28] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2002. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures (Lecture Notes in Computer Science)*, Vol. 2638. Springer, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[29] Yoshifumi Kawai. 2014. UniRx: Reactive Extensions for Unity. (2014). https://github.com/neuecc/UniRx Accessed: 2018-11-13.

[30] D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. 2010. Combinatorial Testing. In *Encyclopedia of Software Engineering*, Phillip A. Laplante (Ed.). CRC Press, 1–12.

[31] Jesse Liberty and Paul Betts. 2011. *Programming Reactive Extensions and LINQ*. Apress.

[32] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2015. SSD: Single Shot MultiBox Detector. *CoRR* abs/1512.02325 (2015). arXiv:1512.02325 http://arxiv.org/abs/1512.02325

[33] Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-Testing: Unit Testing with Mock Objects. In *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*.

[34] Rupak Majumdar and Filip Niksic. 2018. Why is random testing effective for partition tolerance bugs? *PACMPL* 2, POPL (2018), 46:1–46:24.

[35] Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning (ICML)*. https://www.icml.cc/Conferences/2018/Schedule?showEvent=2477

[36] Mockito. [n. d.]. Tasty mocking framework for unit tests in Java. ([n. d.]). http://site.mockito.org

[37] H. Niederreiter. 1992. *Random number generation and quasi-Monte Carlo methods*. SIAM.

[38] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS 17*. ACM. https://doi.org/10.1145/3052973.3053009

[39] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 1–18. https://doi.org/10.1145/3132747.3132785

[40] D. Pomerleau. 1988. ALVINN: An autonomous land vehicle in a neural network. In *NIPS 88: Neural Information Processing Systems*.

[41] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*.

[42] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), 779–788.

[43] S.R. Richter, Z. Hayder, and V. Koltun. 2017. Playing for benchmarks. (2017).

[44] G. Rote and R.F. Tichy. 1996. Quasi-Monte-Carlo methods and the dispersion of point sequences. *Mathematical and Computer Modelling* 23 (1996), 9–23.

[45] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden.*, Jérôme Lang (Ed.). ijcai.org, 2651–2659. https://doi.org/10.24963/ijcai.2018/368

[46] Aitor Ruano. 2017. DeepGTAV: A plugin for GTAV that transforms it into a vision-based self-driving car research environment. https://github.com/aitorzip/DeepGTAV. (2017).

[47] Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. 2013. Declarative Mocking. In *ISSTA 2013*. ACM, 246–256.

[48] S. Sankaranarayanan and G. Fainekos. 2012. Falsification of Temporal Properties of Hybrid Systems Using the Cross-entropy Method. In *HSCC 12*. ACM, 125–134.

[49] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Field and Service Robotics*. arXiv:arXiv:1705.05065 https://arxiv.org/abs/1705.05065

[50] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).

[51] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2013. Intriguing Properties of Neural Networks. *CoRR* abs/1312.6199 (2013).

[52] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

[53] Cumhur Erkan Tuncali, Georgios E. Fainekos, Hisahiro Ito, and James Kapinski. 2018. Sim-ATAV: Simulation-Based Adversarial Testing Framework for Autonomous Vehicles. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC 2018, Porto, Portugal, April 11-13, 2018*. ACM, 283–284. https://doi.org/10.1145/3178126.3187004

[54] Udacity. 2017. Self-Driving Car Simulator. (2017). https://github.com/udacity/self-driving-car-sim Accessed: 2018-10-11.

[55] Unity3D. 2018. Unity Game Engine. (2018). https://unity3d.com/ Accessed: 2018-11-03.

[56] Voyage. 2018. Open Autonomous Safety. (2018). https://oas.voyage.auto/ Accessed: 2018-11-03.

[57] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. ACM, 242–252. https://doi.org/10.1145/349299.349331

[58] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science)*, Dirk Beyer and Marieke Huisman (Eds.), Vol. 10805. Springer, 408–426. https://doi.org/10.1007/978-3-319-89960-2_22

[59] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. 2014. TORCS, The Open Racing Car Simulator. http://www.torcs.org. (2014).