Scenic: A Language for Scenario Specification and Scene Generation

Daniel J. Fremont
University of California, Berkeley
USA
dfremont@berkeley.edu

Tommaso Dreossi*
University of California, Berkeley
USA
tommasodreossi@berkeley.edu

Shromona Ghosh*
University of California, Berkeley
USA
shromona.ghosh@berkeley.edu

Xiangyu Yue*
University of California, Berkeley
USA
xyyue@berkeley.edu

Alberto L. Sangiovanni-Vincentelli
University of California, Berkeley
USA
alberto@berkeley.edu

Sanjit A. Seshia
University of California, Berkeley
USA
sseshia@berkeley.edu

Figure 1. Three scenes generated from a single ~20-line Scenic scenario representing bumper-to-bumper traffic.

Abstract

We propose a new probabilistic programming language for the design and analysis of perception systems, especially those based on machine learning. Specifically, we consider the problems of training a perception system to handle rare events, testing its performance under different conditions, and debugging failures. We show how a probabilistic programming language can help address these problems by specifying distributions encoding interesting types of inputs and sampling these to generate specialized training and test sets. More generally, such languages can be used for cyber-physical systems and robotics to write environment models, an essential prerequisite to any formal analysis. In this paper, we focus on systems like autonomous cars and robots, whose environment is a *scene*, a configuration of physical objects and agents. We design a domain-specific language, Scenic, for describing *scenarios* that are distributions over scenes. As a probabilistic programming language, Scenic allows assigning distributions to features of the scene, as well as declaratively imposing hard and soft constraints over the scene. We develop specialized techniques for sampling from the resulting distribution, taking advantage of the structure provided by Scenic's domain-specific syntax. Finally, we apply Scenic in a case study on a convolutional neural network designed to detect cars in road images, improving its performance beyond that achieved by state-of-the-art synthetic data generation methods.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; **Software testing and debugging**; *Specification languages*; • **Computing methodologies** → *Machine learning*; Computer vision.

Keywords scenario description language, synthetic data, deep learning, probabilistic programming, automatic test generation, fuzz testing

ACM Reference Format:
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*

*These authors contributed equally to the paper.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314633

Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314633

1 Introduction

Machine learning (ML) is increasingly used in safety-critical applications, thereby creating an acute need for techniques to gain higher assurance in ML-based systems [1, 39, 41]. ML has proved particularly effective at perceptual tasks such as speech and vision. Thus, there is a pressing need to tackle several important problems in the design of such ML-based perception systems, including:

- *training* the system so that it correctly responds to events that happen only rarely,
- *testing* the system under a variety of conditions, especially unusual ones, and
- *debugging* the system to understand the root cause of a failure and eliminate it.

The traditional ML approach to these problems is to gather more data from the environment, retraining the system until its performance is adequate. The major difficulty here is that collecting real-world data can be slow and expensive, since it must be preprocessed and correctly labeled before use. Furthermore, it may be difficult or impossible to collect data for corner cases that are rare but nonetheless necessary to train and test against: for example, a car accident. As a result, recent work has investigated training and testing systems with *synthetically generated data*, which can be produced in bulk with correct labels and giving the designer full control over the distribution of the data [21, 22, 24, 44].

A challenge to the use of synthetic data is that it can be highly non-trivial to generate *meaningful* data, since this usually requires modeling complex environments [41]. Suppose we wanted to train a network on images of cars on a road. If we simply sampled uniformly at random from all possible configurations of, say, 12 cars, we would get data that was at best unrealistic, with cars facing sideways or backward, and at worst physically impossible, with cars intersecting each other. Instead, we want scenes like those in Fig. 1, where the cars are laid out in a consistent and realistic way. Furthermore, we may want scenes that are not only realistic but represent particular *scenarios* of interest for training or testing, e.g., parked cars, cars passing across the field of view, or bumper-to-bumper traffic as in Fig. 1. In general, we need a way to *guide* data generation toward scenes that make sense for our application.

We argue that probabilistic programming languages (PPLs) provide a natural solution to this problem. Using a PPL, the designer of a system can construct distributions representing different input regimes of interest, and sample from these distributions to obtain concrete inputs for training and testing. More generally, the designer can model the system's environment, with the program becoming a specification of the distribution of environments under which the system is expected to operate correctly with high probability. Such environment models are essential for any formal analysis: in particular, composing the system with the model, we obtain a closed program which we could potentially prove properties about to establish the correctness of the system.

In this paper, we focus on designing and analyzing systems whose environment is a *scene*, a configuration of objects in space (including dynamic agents, such as vehicles). We develop a domain-specific *scenario description language*, Scenic, to specify such environments. Scenic is a probabilistic programming language, and a Scenic scenario defines a distribution over scenes. As we will see, the syntax of the language is designed to simplify the task of writing complex scenarios, and to enable the use of specialized sampling techniques. In particular, Scenic allows the user to both construct objects in a straightforward imperative style and impose hard and soft constraints declaratively. It also provides readable, concise syntax for common geometric relationships that would otherwise require complex non-linear expressions and constraints. In addition, Scenic provides a notion of classes allowing properties of objects to be given default values depending on other properties: for example, we can define a `Car` so that by default it faces in the direction of the road at its position. More broadly, Scenic uses a novel approach to object construction which factors the process into syntactically-independent *specifiers* which can be combined in arbitrary ways, mirroring the flexibility of natural language. Finally, Scenic provides an easy way to generalize a concrete scene by automatically adding noise.

Generating scenes from a Scenic scenario requires sampling from the probability distribution it implicitly defines. This task is closely related to the inference problem for imperative PPLs with observations [20]. While Scenic could be implemented as a library on top of such a language, we found that clarity and concision could be significantly improved with new syntax (specifiers in particular) difficult to implement as a library. Furthermore, while Scenic could be translated into existing PPLs, using a new language allows us to impose restrictions enabling domain-specific sampling techniques not possible with general-purpose PPLs. In particular, we develop algorithms which take advantage of the particular structure of distributions arising from Scenic programs to dramatically prune the sample space.

Finally, we demonstrate the utility of Scenic in training, testing, and debugging perception systems with a case study on SqueezeDet [48], a convolutional neural network for object detection in autonomous cars. For this task, it has been shown [24] that good performance on real images can be achieved with networks trained purely on synthetic images from the video game Grand Theft Auto V (GTAV [14]). We implemented a sampler for Scenic scenarios, using it to generate scenes which were rendered into images by GTAV. Our experiments demonstrate using Scenic to:

- evaluate the accuracy of the ML system under particular conditions, e.g. in good or bad weather,
- improve performance in corner cases by emphasizing them during training: we use Scenic to both identify a deficiency in a state-of-the-art car detection data set [24] and generate a new training set of equal size but yielding significantly better performance, and
- debug a known failure case by generalizing it in many directions, exploring sensitivity to different features and developing a more general scenario for retraining: we use Scenic to find an image the network misclassifies, discover the root cause, and fix the bug, in the process improving the network's performance on its original test set (again, without increasing training set size).

These experiments show that Scenic can be a very useful tool for understanding and improving perception systems.

While our main case study is performed in the domain of visual perception for autonomous driving, and uses one particular simulator (GTAV), we stress that Scenic is not specific to either. In Sec. 3 we give an example of a different domain (robotic motion planning) and simulator (Webots [30]), and we are currently also using Scenic with the CARLA driving simulator [5] and the X-Plane flight simulator [35] (see Sec. 8). Generally, Scenic *can produce data of any desired type* (e.g. RGB images, LIDAR point clouds, or trajectories from dynamical simulations) by interfacing it to an appropriate simulator. This requires only two steps: (1) writing a small Scenic library defining the types of objects supported by the simulator, as well as the geometry of the workspace; (2) writing an interface layer converting the configurations output by Scenic into the simulator's input format. While the current version of Scenic is primarily concerned with geometry, leaving the details of rendering up to the simulator, the language allows putting distributions on any parameters the simulator exposes: for example, in GTAV the meshes of the various car models are fixed but we can control their overall color. We have also used Scenic to specify distributions over parameters on system dynamics.

In summary, the main contributions of this work are:

- Scenic, a domain-specific probabilistic programming language for describing *scenarios*: distributions over configurations of physical objects and agents;
- a methodology for using PPLs to design and analyze perception systems, especially those based on ML;
- domain-specific algorithms for sampling from the distribution defined by a Scenic program;
- a case study using Scenic to analyze and improve the accuracy of a practical deep neural network for autonomous driving beyond what is achieved by state-of-the-art synthetic data generation methods.

The paper is structured as follows: we begin with an overview of our approach in Sec. 2. Section 3 gives examples highlighting the major features of Scenic and motivating various choices in its design. In Sec. 4 we describe the Scenic language in detail, and in Sec. 5 we discuss its formal semantics and our sampling algorithms. Section 6 describes the experimental setup and results of our car detection case study. Finally, we discuss related work in Sec. 7 and conclude in Sec. 8 with a summary and directions for future work.

An early version of this paper appeared as [11]. For the Appendices and our implementation code, see [13].

2 Using PPLs to Design and Analyze Perception Systems

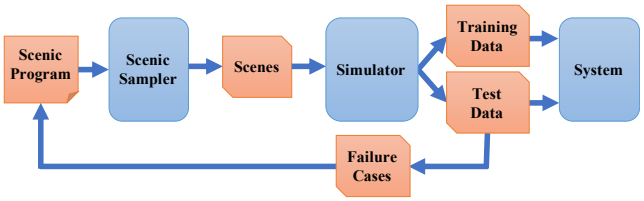

Figure 2. Tool flow using Scenic to train, test, and debug a perception system.

We propose a methodology for training, testing, and debugging perception systems using probabilistic programming languages. The core idea is to use PPLs to formalize general operation scenarios, then sample from these distributions to generate concrete environment configurations. Putting these configurations into a simulator, we obtain images or other sensor data which can be used to test and train the perception system. The general procedure is outlined in Fig. 2. Note that the training/testing datasets need not be purely synthetic: we can generate data to supplement existing real-world data (possibly mitigating a deficiency in the latter, while avoiding overfitting). Furthermore, even for models trained purely on real data, synthetic data can still be useful for testing and debugging, as we will see below. Now we discuss the three design problems from the Introduction in more detail.

Testing under Different Conditions. The most straightforward problem is that of assessing system performance under different conditions. We can simply write scenarios capturing each condition, generate a test set from each one, and evaluate the performance of the system on these. Note that conditions which occur rarely in the real world present no additional problems: as long as the PPL we use can encode the condition, we can generate as many instances as desired.

Training on Rare Events. Extending the previous application, we can use this procedure to help ensure the system performs adequately even in unusual circumstances or particularly difficult cases. Writing a scenario capturing these rare events, we can generate instances of them to augment or replace part of the original training set. Emphasizing these

instances in the training set can improve the system's performance in the hard case without impacting performance in the typical case. In Sec. 6.3 we will demonstrate this for car detection, where a hard case is when one car partially overlaps another in the image. We wrote a Scenic program to generate a set of these overlapping images. Training the car-detection network on a state-of-the-art synthetic dataset obtained by randomly driving around inside the simulated world of GTAV and capturing images periodically [24], we find its performance is significantly worse on the overlapping images. However, if we keep the training set size fixed but increase the proportion of overlapping images, performance on such images dramatically improves *without harming performance on the original generic dataset.*

Debugging Failures. Finally, we can use the same procedure to help understand and fix bugs in the system. If we find an environment configuration where the system fails, we can write a scenario reproducing that particular configuration. Having the configuration encoded as a program then makes it possible to explore the neighborhood around it in a variety of different directions, leaving some aspects of the scene fixed while varying others. This can give insight into which features of the scene are relevant to the failure, and eventually identify the root cause. The root cause can then itself be encoded into a scenario which generalizes the original failure, allowing retraining without overfitting to the particular counterexample. We will demonstrate this approach in Sec. 6.4, starting from a single misclassification, identifying a general deficiency in the training set, replacing part of the training data to fix the gap, and ultimately achieving higher performance on the original test set.

For all of these applications we need a PPL which can encode a wide range of general and specific environment scenarios. In the next section, we describe the design of a language suited to this purpose.

3 The Scenic Language

We use Scenic scenarios from our autonomous car case study to motivate and illustrate the main features of the language, focusing on features that make Scenic particularly well-suited for the domain of generating data for perception systems.

Basics: Classes, Objects, Geometry, and Distributions. To start, suppose we want scenes of one car viewed from another on the road. We can simply write:

```
1 import gtaLib
2 ego = Car
3 Car
```

First, we import a library `gtaLib` containing everything specific to our case study: the class `Car` and information about the locations of roads (from now on we suppress this line). Only general geometric concepts are built into Scenic.

The second line creates a `Car` and assigns it to the special variable `ego` specifying the *ego object* which is the reference point for the scenario. In particular, rendered images from the scenario are from the perspective of the ego object (it is a syntax error to leave `ego` undefined). Finally, the third line creates an additional `Car`. Note that we have not specified the position or any other properties of the two cars: this means they are inherited from the *default values* defined in the *class* `Car`. Object-orientation is valuable in Scenic since it provides a natural organizational principle for scenarios involving different types of physical objects. It also improves compositionality, since we can define a generic `Car` model in a library like `gtaLib` and use it in different scenarios. Our definition of `Car` begins as follows (slightly simplified):

```
1 class Car:
2     position: Point on road
3     heading: roadDirection at self.position
```

Here `road` is a *region* (one of Scenic's primitive types) defined in `gtaLib` to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *vector field* specifying the prevailing traffic direction at such points. The operator *F* `at` *X* simply gets the direction of the field *F* at point *X*, so the default value for a car's `heading` is the road direction at its `position`. The default `position`, in turn, is a `Point on road` (we will explain this syntax shortly), which means a *uniformly random* point on the road.

The ability to make random choices like this is a key aspect of Scenic. Scenic's probabilistic nature allows it to model real-world stochasticity, for example encoding a distribution for the distance between two cars learned from data. This in turn is essential for our application of PPLs to training perception systems: using randomness, a PPL can generate training data matching the distribution the system will be used under. Scenic provides several basic distributions (and allows more to be defined). For example, we can write

```
1 Car offset by (-10, 10) @ (20, 40)
```

to create a car that is 20–40 m ahead of the camera. The interval notation `(`*X*`,` *Y*`)` creates a uniform distribution on the interval, and *X* `@` *Y* creates a vector from *xy* coordinates (as in Smalltalk [15]).

Local Coordinate Systems. Using `offset by` as above overrides the default position of the `Car`, leaving the default orientation (along the road) unchanged. Suppose for greater realism we don't want to require the car to be *exactly* aligned with the road, but to be within say 5°. We could try:

```
1 Car offset by (-10, 10) @ (20, 40), \
2     facing (-5, 5) deg
```

but this is not quite what we want, since this sets the orientation of the `Car` in *global* coordinates (i.e. within 5° of North). Instead we can use Scenic's general operator *X* `relative to` *Y*, which can interpret vectors and headings as being in a variety of local coordinate systems:

```
Car offset by (-10, 10) @ (20, 40), \
  facing (-5, 5) deg relative to roadDirection
```

If we want the heading to be relative to the ego car's orientation, we simply write `(-5, 5) deg relative to ego`.

Notice that since `roadDirection` is a vector field, it defines a coordinate system at each point, and an expression like `15 deg relative to field` does not define a unique heading. The example above works because Scenic knows that `(-5, 5) deg relative to roadDirection` depends on a reference position, and automatically uses the `position` of the `Car` being defined. This is a feature of Scenic's system of *specifiers*, which we explain next.

Readable, Flexible Specifiers. The syntax `offset by` *X* and `facing` *Y* for specifying positions and orientations may seem unusual compared to typical constructors in object-oriented languages. There are two reasons why Scenic uses this kind of syntax: first, readability. The second is more subtle and based on the fact that in natural language there are many ways to specify positions and other properties, some of which interact with each other. Consider the following ways one might describe the location of an object:

1. "is at position *X*" (absolute position);
2. "is just left of position *X*" (pos. based on orientation);
3. "is 3 m left of the taxi" (a local coordinate system);
4. "is one lane left of the taxi" (another local system);
5. "appears to be 10 m behind the taxi" (relative to the line of sight);

These are all fundamentally different from each other: e.g., (3) and (4) differ if the taxi is not parallel to the lane.

Furthermore, these specifications combine other properties of the object in different ways: to place the object "just left of" a position, we must first know the object's `heading`; whereas if we wanted to face the object "towards" a location, we must instead know its `position`. There can be chains of such *dependencies*: "the car is 0.5 m left of the curb" means that the *right edge* of the car is 0.5 m away from the curb, not the car's `position`, which is its center. So the car's `position` depends on its `width`, which in turn depends on its `model`. In a typical object-oriented language, this might be handled by computing values for `position` and other properties and passing them to a constructor. For "a car is 0.5 m left of the curb" we might write:

```
m = Car.defaultModelDistribution.sample()
pos = curb.offsetLeft(0.5 + m.width / 2)
car = Car(pos, model=m)
```

Notice how `m` must be used twice, because `m` determines both the model of the car and (indirectly) its position. This is inelegant and breaks encapsulation because the default model distribution is used outside of the `Car` constructor. The latter problem could be fixed by having a specialized constructor or factory function,

```
car = CarLeftOfBy(curb, 0.5)
```

but these would proliferate since we would need to handle all possible combinations of ways to specify different properties (e.g. do we want to require a specific model? Are we overriding the width provided by the model for this specific car?). Instead of having a multitude of such monolithic constructors, Scenic factors the definition of objects into potentially-interacting but syntactically-independent parts:

```
Car left of spot by 0.5, with model BUS
```

Here `left of` *X* `by` *D* and `with model` *M* are *specifiers* which do not have an order, but which *together* specify the properties of the car. Scenic works out the dependencies between properties (here, `position` is provided by `left of`, which depends on `width`, whose default value depends on `model`) and evaluates them in the correct order. To use the default model distribution we would simply leave off `with model BUS`; keeping it affects the `position` appropriately without having to specify `BUS` more than once.

Specifying Multiple Properties Together. Recall that we defined the default `position` for a `Car` to be a `Point on road`: this is an example of another specifier, `on` *region*, which specifies `position` to be a uniformly random point in the given region. This specifier illustrates another feature of Scenic, namely that specifiers can specify multiple properties simultaneously. Consider the following scenario, which creates a parked car given a region `curb` defined in `gtaLib`:

```
spot = OrientedPoint on visible curb
Car left of spot by 0.25
```

The function `visible` *region* returns the part of the region that is visible from the ego object. The specifier `on visible curb` will then set `position` to be a uniformly random visible point on the curb. We create `spot` as an `OrientedPoint`, which is a built-in class that defines a local coordinate system by having both a `position` and a `heading`. The `on` *region* specifier can also specify `heading` if the region has a preferred orientation (a vector field) associated with it: in our example, `curb` is oriented by `roadDirection`. So `spot` is, in fact, a uniformly random visible point on the curb, oriented along the road. That orientation then causes the car to be placed 0.25 m left of `spot` in `spot`'s local coordinate system, i.e. away from the curb, as desired.

In fact, Scenic makes it easy to elaborate the scenario without needing to alter the code above. Most simply, we could specify a particular model or non-default distribution over models by just adding `with model` *M* to the definition of the `Car`. More interestingly, we could produce a scenario for *badly*-parked cars by adding two lines:

```
spot = OrientedPoint on visible curb
badAngle = Uniform(1.0, -1.0) * (10, 20) deg
Car left of spot by 0.5, \
    facing badAngle relative to roadDirection
```

Figure 3. A scene of a badly-parked car.

This will yield cars parked 10-20° off from the direction of the curb, as seen in Fig. 3. This illustrates how specifiers greatly enhance Scenic's flexibility and modularity.

Declarative Specifications of Hard and Soft Constraints. Notice that in the scenarios above we never explicitly ensured that the two cars will not intersect each other. Despite this, Scenic will never generate such scenes. This is because Scenic enforces several *default requirements*: all objects must be contained in the workspace, must not intersect each other, and must be visible from the ego object.[1] Scenic also allows the user to define custom requirements checking arbitrary conditions built from various geometric predicates. For example, the following scenario produces a car headed roughly towards us, while still facing the nominal road direction:

```
car2 = Car offset by (-10, 10) @ (20, 40), \
           with viewAngle 30 deg
require car2 can see ego
```

Here we have used the *X* `can see` *Y* predicate, which in this case is checking that the ego car is inside the 30° view cone of the second car. If we only need this constraint to hold part of the time, we can use a *soft requirement* specifying the minimum probability with which it must hold:

```
require[0.5] car2 can see ego
```

Hard requirements, called "observations" in other PPLs (see, e.g., [20]), are very convenient in our setting because they make it easy to restrict attention to particular cases of interest. They also improve encapsulation, since we can restrict an existing scenario without altering it. Finally, soft requirements are useful in ensuring adequate representation of a particular condition when generating a training set: for example, we could require that at least 90% of the images have a car driving on the right side of the road.

Mutations. Scenic provides a simple *mutation* system that improves compositionality by providing a mechanism to add variety to a scenario without changing its code. This is useful, for example, if we have a scenario encoding a single concrete scene obtained from real-world data and want to quickly generate variations. For instance:

```
taxi = Car at 120 @ 300, facing 37 deg, ...
...
mutate taxi
```

This will add Gaussian noise to the `position` and `heading` of `taxi`, while still enforcing all built-in and custom requirements. The standard deviation of the noise can be scaled by writing, for example, `mutate taxi by 2` (which adds twice as much noise), and we will see later that it can be controlled separately for `position` and `heading`.

[1] The last requirement ensures that the object will affect the rendered image. It can be disabled, if for example generating non-visual data.

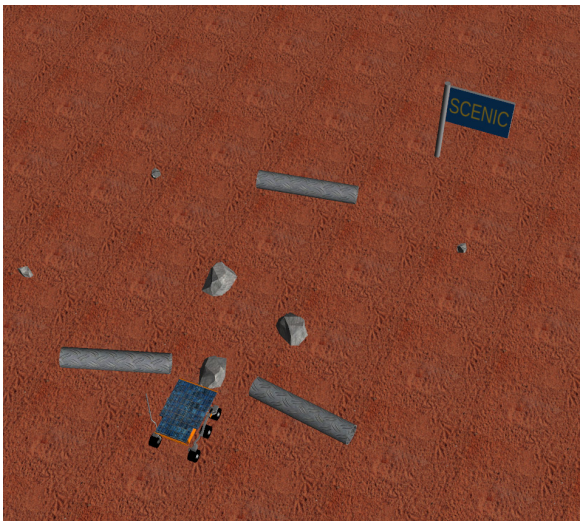

Figure 4. Webots scene of Mars rover in debris field.

Multiple Domains and Simulators. We conclude this section with an example illustrating a second application domain, namely generating workspaces to test motion planning algorithms, and Scenic's ability to work with different simulators. A robot like a Mars rover able to climb over rocks can have very complex dynamics, with the feasibility of a motion plan depending on exact details of the robot's hardware and the geometry of the terrain. We can use Scenic to write a scenario generating challenging cases for a planner to solve. Figure 4 shows a scene, visualized using an interface we wrote between Scenic and the Webots robotics simulator [30], with a bottleneck between the robot and its goal that forces the planner to consider climbing over a rock.

This example, the badly-parked car scenario of Fig. 3, and the bumper-to-bumper traffic scenario of Fig. 1 illustrate the versatility of Scenic in constructing a wide range of interesting scenarios. Complete Scenic code for the bumper-to-bumper scenario as well as other scenarios used as examples in this section or in our experiments, along with images of generated scenes, can be found in Appendix A of [13].

```
scenario := (import file)* (statement)*
boolean := True | False | booleanOperator
scalar := number | distrib | scalarOperator
distrib := baseDist | resample(distrib)
vector := scalar @ scalar | Point | vectorOperator
heading := scalar | OrientedPoint | headingOperator
direction := heading | vectorField
value := boolean | scalar | vector | direction
       | region | instance | instance.property
classDefn := class class[(superclass)]:
               (property: defaultValueExpr)*
instance := class specifier, ...
specifier := with property value | posSpec | headSpec
```

Figure 5. Simplified Scenic grammar. *Point* and *OrientedPoint* are instances of the corresponding classes. See Tab. 5 for statements, Fig. 7 for operators, Tab. 1 for *baseDist*, and Tables 3 and 4 for *posSpec* and *headSpec*.

4 Syntax of Scenic

Scenic is a simple object-oriented PPL, with programs consisting of sequences of statements built with standard imperative constructs including conditionals, loops, functions, and methods (which we do not describe further, focusing on the new elements). Compared to other imperative PPLs, the major restriction of Scenic, made in order to allow more efficient sampling, is that conditional branching may not depend on random variables. The novel syntax, outlined above, is largely devoted to expressing geometric relationships in a concise and flexible manner. Figure 5 gives a formal grammar for Scenic, which we now describe in detail.

4.1 Data Types

Scenic provides several primitive data types:

Booleans expressing truth values.

Scalars floating-point numbers, which can be sampled from various distributions (see Table 1).

Vectors representing positions and offsets in space, constructed from coordinates in meters with the syntax `X @ Y` (inspired by Smalltalk [15]).

Headings representing orientations in space. Conveniently, in 2D these are a single angle (in radians, anticlockwise from North). By convention the heading of a local coordinate system is the heading of its y-axis, so, for example, `-2 @ 3` means 2 meters left and 3 ahead.

Vector Fields associating an orientation to each point in space. For example, the shortest paths to a destination or (in our case study) the nominal traffic direction.

Regions representing sets of points in space. These can have an associated vector field giving points in the region preferred orientations.

Table 1. Distributions. All parameters *scalar*s except *value*.

Syntax	Distribution
`(`*low*`,` *high*`)`	uniform on interval
`Uniform(`*value*`, ...)`	uniform over values
`Discrete({`*value*`:` *wt*`, ...})`	discrete with weights
`Normal(`*mean*`,` *stdDev*`)`	normal with given μ, σ

Table 2. Properties of `Point`, `OrientedPoint`, and `Object`.

Property	Default	Meaning
`position`	`0@0`	position in global coords.
`viewDistance`	50	distance for 'can see'
`mutationScale`	0	overall scale of mutations
`positionStdDev`	1	mutation σ for `position`
`heading`	0	heading in global coords.
`viewAngle`	360°	angle for 'can see'
`headingStdDev`	5°	mutation σ for `heading`
`width`	1	width of bounding box
`height`	1	height of bounding box
`allowCollisions`	false	collisions allowed
`requireVisible`	true	must be visible from ego

In addition, Scenic provides *objects*, organized into single-inheritance *classes* specifying a set of properties their instances must have, together with corresponding default values (see Fig. 5). Default value expressions are evaluated each time an object is created. Thus if we write `weight: (1, 5)` when defining a class then each instance will have a `weight` drawn *independently* from `(1, 5)`. Default values may use the special syntax `self.`*property* to refer to one of the other properties of the object, which is then a *dependency* of this default value. In our case study, for example, the `width` and `height` of a `Car` are by default derived from its `model`.

Physical objects in a scene are instances of `Object`, which is the default superclass when none is specified. `Object` descends from the two other built-in classes: its superclass is `OrientedPoint`, which in turn subclasses `Point`. These represent locations in space, with and without an orientation respectively, and so provide the fundamental properties `heading` and `position`. `Object` extends them by defining a bounding box with the properties `width` and `height`. Table 2 lists the properties of these classes and their default values.

To allow cleaner notation, `Point` and `OrientedPoint` are automatically interpreted as vectors or headings in contexts expecting these (as shown in Fig. 5). For example, we can write `taxi offset by 1 @ 2` and `30 deg relative to taxi` instead of `taxi.position offset by 1 @ 2` and `30 deg relative to taxi.heading`. Ambiguous cases, e.g. `taxi relative to limo`, are illegal (caught by a simple type system); the more verbose syntax must be used instead.

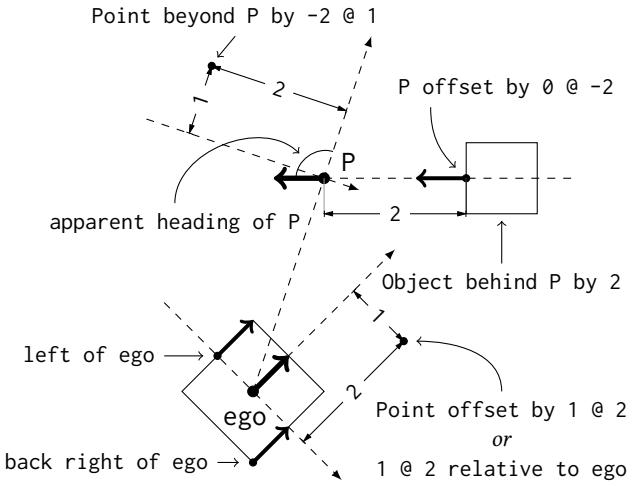

Figure 6. Various Scenic operators and specifiers applied to the ego object and an `OrientedPoint` P. Instances of `OrientedPoint` are shown as bold arrows.

4.2 Expressions

Scenic's expressions are mostly straightforward, largely consisting of the arithmetic, boolean, and geometric operators in Fig. 7. The meanings of these operators are largely clear from their syntax, so we defer complete definitions of their semantics to Appendix C of [13]. Figure 6 illustrates several of the geometric operators (as well as some specifiers, which we will discuss in the next section). Various points to note:

- *X* `can see` *Y* uses a simple model where a `Point` can see a certain distance, and an `OrientedPoint` restricts this to the sector along its `heading` with a certain angle (see Table 2). An `Object` is visible iff its bounding box is.
- *X* `relative to` *Y* interprets *X* as an offset in a local coordinate system defined by *Y*. Thus `-3 @ 0 relative to` *Y* yields 3 m West of *Y* if *Y* is a vector, and 3 m *left* of *Y* if *Y* is an `OrientedPoint`. If defining a heading inside a specifier, either *X* or *Y* can be a vector field, interpreted as a heading by evaluating it at the `position` of the object being specified. So we can write for example `Car at 120 @ 70, facing 30 deg relative to roadDirection`.
- `visible` *region* yields the part of the region visible from the ego, e.g. `Car on visible road`. The form *region* `visible from` *X* uses *X* instead of ego.

Two types of Scenic expressions are more complex: distributions and object definitions. As in a typical imperative probabilistic programming language, a distribution evaluates to a *sample* from the distribution. Thus the program

```
1 x = (0, 1)
2 y = x @ x
```

does not make y uniform over the unit box, but rather over its diagonal. For convenience in sampling multiple times from a primitive distribution, Scenic provides a `resample(`*D*`)` function returning an independent[2] sample from *D*, one of the distributions in Tab. 1. Scenic also allows defining custom distributions beyond those in the Table.

scalarOperator := `max(`*scalar*`, ...)` | `min(`*scalar*`, ...)`
| `-`*scalar* | `abs(`*scalar*`)` | *scalar* (`+` | `*`) *scalar*
| `relative heading of` *heading* [`from` *heading*]
| `apparent heading of` *OrientedPoint* [`from` *vector*]
| `distance` [`from` *vector*] `to` *vector*
| `angle` [`from` *vector*] `to` *vector*

booleanOperator := `not` *boolean*
| *boolean* (`and` | `or`) *boolean*
| *scalar* (`==` | `!=` | `<` | `>` | `<=` | `>=`) *scalar*
| (*Point* | *OrientedPoint*) `can see` (*vector* | *Object*)
| (*vector* | *Object*) `is in` *region*

headingOperator := *scalar* `deg`
| *vectorField* `at` *vector*
| *direction* `relative to` *direction*

vectorOperator := *vector* `relative to` *vector*
| *vector* `offset by` *vector*
| *vector* `offset along` *direction* `by` *vector*

regionOperator := `visible` *region*
| *region* `visible from` (*Point* | *OrientedPoint*)

orientedPointOperator :=
vector `relative to` *OrientedPoint*
| *OrientedPoint* `offset by` *vector*
| `follow` *vectorField* [`from` *vector*] `for` *scalar*
| (`front` | `back` | `left` | `right`) `of` *Object*
| (`front` | `back`) (`left` | `right`) `of` *Object*

Figure 7. Operators by result type.

The second type of complex Scenic expressions are object definitions. These are the only expressions with a side effect, namely creating an object in the generated scene. More interestingly, properties of objects are specified using the system of *specifiers* discussed above, which we now detail.

4.3 Specifiers

As shown in the grammar in Fig. 5, an object is created by writing the class name followed by a (possibly empty) comma-separated list of specifiers. The specifiers are combined, possibly adding default specifiers from the class definition, to form a complete specification of all properties of the object. Arbitrary properties (including user-defined properties with no meaning in Scenic) can be specified with the generic specifier `with` *property value*, while Scenic provides many more specifiers for the built-in properties `position` and `heading`, shown in Tables 3 and 4 respectively.

In general, a specifier is a function taking in values for zero or more properties, its *dependencies*, and returning values for one or more other properties, some of which can be specified

[2] Conditioned on the values of the distribution's parameters (e.g. *low* and *high* for a uniform interval), which are not resampled.

Table 3. Specifiers for `position`. Those in the second group also optionally specify `heading`.

Specifier	Dependencies
`at` *vector*	—
`offset by` *vector*	—
`offset along` *direction* `by` *vector*	—
(`left` | `right`) `of` *vector* [`by` *scalar*]	`heading`, `width`
(`ahead of` | `behind`) *vector* [`by` *scalar*]	`heading`, `height`
`beyond` *vector* `by` *vector* [`from` *vector*]	—
`visible` [`from` (*Point* | *OrientedPoint*)]	—
(`in` | `on`) *region*	—
(`left` | `right`) `of` (*OrientedPoint* | *Object*) [`by` *scalar*]	`width`
(`ahead of` | `behind`) (*OrientedPoint* | *Object*) [`by` *scalar*]	`height`
`following` *vectorField* [`from` *vector*] `for` *scalar*	—

Table 4. Specifiers for `heading`.

Specifier	Deps.
`facing` *heading*	—
`facing` *vectorField*	`position`
`facing` (`toward` | `away from`) *vector*	`position`
`apparently facing` *heading* [`from` *vector*]	`position`

Table 5. Statements.

Syntax	Meaning
identifier `=` *value*	var. assignment
`param` *identifier* `=` *value*, ...	param. assign.
classDefn	class definition
instance	object definition
`require` *boolean*	hard requirement
`require[`*number*`]` *boolean*	soft requirement
`mutate` *identifier*, ... [`by` *number*]	enable mutation

optionally, meaning that other specifiers will override them. For example, `on` *region* specifies `position` and optionally specifies `heading` if the given region has a preferred orientation. If `road` is such a region, as in our case study, then `Object on road` will create an object at a position uniformly random in `road` and with the preferred orientation there. But since `heading` is only specified optionally, we can override it by writing `Object on road, facing 20 deg`.

Specifiers are combined to determine the properties of an object by evaluating them in an order ensuring that their dependencies are always already assigned. If there is no such order or a single property is specified twice, the scenario is ill-formed. The procedure by which the order is found, taking into account properties that are optionally specified and default values, will be described in the next section.

As the semantics of the specifiers in Tables 3 and 4 are largely evident from their syntax, we defer exact definitions to Appendix C of [13]. We briefly discuss some of the more complex specifiers, referring to the examples in Fig. 6:

- `behind` *vector* means the object is placed with the midpoint of its front edge at the given vector, and similarly for `ahead/left/right of` *vector*.
- `beyond` *A* `by` *O* `from` *B* means the position obtained by treating *O* as an offset in the local coordinate system at *A* oriented along the line of sight from *B*. In this and other specifiers, if the `from` *B* is omitted, the ego object is used by default. So for example `beyond taxi by 0 @ 3` means 3 m directly behind the taxi as viewed by the camera (see Fig. 6 for another example).
- The `heading` optionally specified by `left of` *OrientedPoint*, etc. is that of the `OrientedPoint` (thus in Fig. 6, `P offset by 0 @ -2` yields an `OrientedPoint` facing the same way as `P`). Similarly, the `heading` optionally specified by `following` *vectorField* is that of the vector field at the specified `position`.
- `apparently facing` *H* means the object has heading *H* with respect to the line of sight from `ego`. For example, `apparently facing 90 deg` would orient the object so that the camera views its left side head-on.

4.4 Statements

Finally, we discuss Scenic's statements, listed in Table 5. Class and object definitions have been discussed above, and variable assignment behaves in the standard way.

The statement `param` *identifer* `=` *value* assigns values to global parameters of the scenario. These have no semantics in Scenic but provide a general-purpose way to encode arbitrary global information. For example, in our case study we used parameters `time` and `weather` to put distributions on the time of day and the weather conditions during the scene.

The `require` *boolean* statement requires that the given condition hold in all instantiations of the scenario (equivalently to *observe* statements in other probabilistic programming languages; see e.g. [4, 31]). The variant statement `require[`*p*`]` *boolean* adds a *soft* requirement that need only hold with some probability *p* (which must be a constant). We will discuss the semantics of these in the next section.

Lastly, the `mutate` *instance*`, ...` `by` *number* statement adds Gaussian noise with the given standard deviation (default 1) to the `position` and `heading` properties of the listed objects (or every `Object`, if no list is given). For example, `mutate taxi by 2` would add twice as much noise as `mutate taxi`. The noise can be controlled separately for `position` and `heading`, as we discuss in the next section.

5 Semantics and Scene Generation

5.1 Semantics of Scenic

The output of a Scenic program is a *scene* consisting of the assignment to all the properties of each `Object` defined in the scenario, plus any global parameters defined with `param`. Since Scenic is a probabilistic programming language, the semantics of a program is actually a *distribution* over possible outputs, here scenes. As for other imperative PPLs, the semantics can be defined operationally as a typical interpreter for an imperative language but with two differences. First, the interpreter makes random choices when evaluating distributions [40]. For example, the Scenic statement `x = (0, 1)` updates the state of the interpreter by assigning a value to `x` drawn from the uniform distribution on the interval $(0, 1)$. In this way every possible run of the interpreter has a probability associated with it. Second, every run where a `require` statement (the equivalent of an "observation" in other PPLs) is violated gets discarded, and the run probabilities appropriately normalized (see, e.g., [20]). For example, adding the statement `require x > 0.5` above would yield a uniform distribution for `x` over the interval $(0.5, 1)$.

Scenic uses the standard semantics for assignments, arithmetic, loops, functions, and so forth. Below, we define the semantics of the main constructs unique to Scenic. See Appendix B of [13] for a more formal treatment.

Soft Requirements. The statement `require[p] B` is interpreted as `require B` with probability `p` and as a no-op otherwise: that is, it is interpreted as a hard requirement that is only checked with probability `p`. This ensures that the condition `B` will hold with probability at least `p` in the induced distribution of the Scenic program, as desired.

Specifiers and Object Definitions. As we saw above, each specifier defines a function mapping values for its dependencies to values for the properties it specifies. When an object of class C is constructed using a set of specifiers S, the object is defined as follows (see Appendix B of [13] for details):

1. If a property is specified (non-optionally) by multiple specifiers in S, an ambiguity error is raised.
2. The set of properties P for the new object is found by combining the properties specified by all specifiers in S with the properties inherited from the class C.
3. Default value specifiers from C are added to S as needed so that each property in P is paired with a unique specifier in S specifying it, with precedence order: non-optional specifier, optional specifier, then default value.
4. The dependency graph of the specifiers S is constructed. If it is cyclic, an error is raised.
5. The graph is topologically sorted and the specifiers are evaluated in this order to determine the values of all properties P of the new object.

Mutation. The `mutate X by N` statement sets the special `mutationScale` property to `N` (the `mutate X` form sets it to 1). At the end of evaluation of the Scenic program, but before requirements are checked, Gaussian noise is added to the `position` and `heading` properties of objects with nonzero `mutationScale`. The standard deviation of the noise is the value of the `positionStdDev` and `headingStdDev` property respectively (see Table 2), multiplied by `mutationScale`.

The problem of sampling scenes from the distribution defined by a Scenic program is essentially a special case of the sampling problem for imperative PPLs with observations (since soft requirements can also be encoded as observations). While we could apply general techniques for such problems, the domain-specific design of Scenic enables specialized sampling methods, which we discuss below. We also note that the scene generation problem is closely related to *control improvisation*, an abstract framework capturing various problems requiring synthesis under hard, soft, and randomness constraints [12]. *Scene improvisation* from a Scenic program can be viewed as an extension with a more detailed randomness constraint given by the imperative part of the program.

5.2 Domain-Specific Sampling Techniques

The geometric nature of the constraints in Scenic programs, together with Scenic's lack of conditional control flow, enable domain-specific sampling techniques inspired by robotic path planning methods. Specifically, we can use ideas for constructing configuration spaces to prune parts of the sample space where the objects being positioned do not fit into the workspace. We describe three such techniques below, deferring formal statements of the algorithms to Appendix B of [13].

Pruning Based on Containment. The simplest technique applies to any object X whose position is uniform in a region R and which must be contained in a region C (e.g. the road in our case study). If *minRadius* is a lower bound on the distance from the center of X to its bounding box, then we can restrict R to $R \cap \text{erode}(C, \textit{minRadius})$. This is sound, since

if X is centered anywhere not in the restriction, then some point of its bounding box must lie outside of C.

Pruning Based on Orientation. The next technique applies to scenarios placing constraints on the relative heading and the maximum distance M between objects X and Y, which are oriented with respect to a vector field that is constant within polygonal regions (such as our roads). For each polygon P, we find all polygons Q_i satisfying the relative heading constraints with respect to P (up to a perturbation if X and Y need not be exactly aligned to the field), and restrict P to $P \cap \text{dilate}(\cup Q_i, M)$. This is also sound: suppose X can be positioned at x in polygon P. Then Y must lie at some y in a polygon Q satisfying the constraints, and since the distance from x to y is at most M, we have $x \in \text{dilate}(Q, M)$.

Pruning Based on Size. Finally, in the setting above of objects X and Y aligned to a polygonal vector field (with maximum distance M), we can also prune the space using a lower bound on the width of the configuration. For example, in our bumper-to-bumper scenario we can infer such a bound from the `offset by` specifiers in the program. We first find all polygons that are not wide enough to fit the configuration according to the bound: call these "narrow". Then we restrict each narrow polygon P to $P \cap \text{dilate}(\cup Q_i, M)$ where Q_i runs over all polygons except P. To see that this is sound, suppose object X can lie at x in polygon P. If P is not narrow, we do not restrict it; otherwise, object Y must lie at y in some other polygon Q. Since the distance from x to y is at most M, as above we have $x \in \text{dilate}(Q, M)$.

After pruning the space as described above, our implementation uses rejection sampling, generating scenes from the imperative part of the scenario until all requirements are satisfied. While this samples from exactly the desired distribution, it has the drawback that a huge number of samples may be required to yield a single valid scene (in the worst case, when the requirements have probability zero of being satisfied, the algorithm will not even terminate). However, we found in our experiments that all reasonable scenarios we tried required only several hundred iterations at most, yielding a sample within a few seconds. Furthermore, the pruning methods above could reduce the number of samples needed by a factor of 3 or more (see Appendix D of [13] for details of our experiments). In future work it would be interesting to see whether Markov chain Monte Carlo methods previously used for probabilistic programming (see, e.g., [31, 34, 47]) could be made effective in the case of Scenic.

6 Experiments

We demonstrate the three applications of Scenic discussed in Sec. 2: testing a system under particular conditions (6.2), training the system to improve accuracy in hard cases (6.3), and debugging failures (6.4).

6.1 Experimental Setup

We generated scenes in the virtual world of the video game Grand Theft Auto V (GTAV) [14]. We wrote a Scenic library `gtaLib` defining `Regions` representing the roads and curbs in (part of) this world, as well as a type of object `Car` providing two additional properties[3]: `model`, representing the type of car, with a uniform distribution over 13 diverse models provided by GTAV, and `color`, representing the car color, with a default distribution based on real-world car color statistics [7]. In addition, we implemented two global scene parameters: `time`, representing the time of day, and `weather`, representing the weather as one of 14 discrete types supported by GTAV (e.g. "clear" or "snow").

GTAV is closed-source and does not expose any kind of scene description language. Therefore, to import scenes generated by Scenic into GTAV, we wrote a plugin based on DeepGTAV[4]. The plugin calls internal functions of GTAV to create cars with the desired positions, colors, etc., as well as to set the camera position, time of day, and weather.

Our experiments used squeezeDet [48], a convolutional neural network real-time object detector for autonomous driving[5]. We used a batch size of 20 and trained all models for 10,000 iterations unless otherwise noted. Images captured from GTAV with resolution 1920×1200 were resized to 1248×384, the resolution used by squeezeDet. All models were trained and evaluated on NVIDIA TITAN Xp GPUs.

We used standard metrics *precision* and *recall* to measure the accuracy of detection on a particular image set. The accuracy is computed based on how well the network predicts the correct bounding box, score, and category of objects in the image set. Details are in Appendix D of [13], but in brief, precision is defined as $tp/(tp + fp)$ and recall as $tp/(tp + fn)$, where *true positives* tp is the number of correct detections, *false positives* fp is the number of predicted boxes that do not match any ground truth box, and *false negatives* fn is the number of ground truth boxes that are not detected.

6.2 Testing under Different Conditions

When testing a model, one may be interested in a particular operation regime. For instance, an autonomous car manufacturer may be more interested in certain road conditions (e.g. desert vs. forest roads) depending on where its cars will be mainly used. Scenic provides a systematic way to describe scenarios of interest and construct corresponding test sets.

To demonstrate this, we first wrote very general scenarios describing scenes of 1–4 cars (not counting the camera), specifying only that the cars face within 10° of the road direction. We generated 1,000 images from each scenario, yielding a training set X_{generic} of 4,000 images, and used

[3] For the full definition of `Car`, see Appendix A of [13]; the definitions of `road`, `curb`, etc. are a few lines loading the corresponding sets of points from a file storing the GTAV map (see Appendix D for how this was generated).
[4] https://github.com/aitorzip/DeepGTAV
[5] Used industrially, for example by DeepScale (http://deepscale.ai/).

```
wiggle = (-10 deg, 10 deg)
ego = Car with roadDeviation wiggle
c = Car visible, \
        with roadDeviation resample(wiggle)
leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
Car beyond c by leftRight @ (4, 10), \
    with roadDeviation resample(wiggle)
```

Figure 8. A scenario where one car partially occludes another. The property `roadDeviation` is defined in `Car` to mean its `heading` relative to the `roadDirection`.

these to train a model M_{generic} as described in Sec. 6.1. We also generated an additional 50 images from each scenario to obtain a generic test set T_{generic} of 200 images.

Next, we specialized the general scenarios in opposite directions: scenarios for good/bad road conditions fixing the time to noon/midnight and the weather to sunny/rainy respectively, generating specialized test sets T_{good} and T_{bad}.

Evaluating M_{generic} on T_{generic}, T_{good}, and T_{bad}, we obtained precisions of 83.1%, 85.7%, and 72.8%, respectively, and recalls of 92.6%, 94.3%, and 92.8%. This shows that, as might be expected, the model performs better on bright days than on rainy nights. This suggests there might not be enough examples of rainy nights in the training set, and indeed under our default weather distribution rain is less likely than shine. This illustrates how specialized test sets can highlight the weaknesses and strengths of a particular model. In the next section, we go one step further and use Scenic to redesign the training set and improve model performance.

6.3 Training on Rare Events

In the synthetic data setting, we are limited not by data availability but by the cost of training. The natural question is then how to generate a synthetic data set that as effective as possible given a fixed size. In this section we show that *over-representing* a type of input that may occur rarely but is difficult for the model can improve performance on the hard case without compromising performance in the typical case. Scenic makes this possible by allowing the user to write a scenario capturing the hard case specifically.

For our car detection task, an obvious hard case is when one car substantially occludes another. We wrote a simple scenario, shown in Fig. 8, which generates such scenes by placing one car behind the other as viewed from the camera, offset left or right so that it is at least partially visible (sample images are in Appendix A of [13]). Generating images from this scenario we obtained a training set X_{overlap} of 250 images and a test set T_{overlap} of 200 images.

For a baseline training set we used the "Driving in the Matrix" synthetic data set [24], which has been shown to yield good car detection performance even on real-world images[6]. Like our images, the "Matrix" images were rendered in GTAV; however, rather than using a PPL to guide generation, they were produced by allowing the game's AI to drive around randomly while periodically taking screenshots. We randomly selected 5,000 of these images to form a training set X_{matrix}, and 200 for a test set T_{matrix}. We trained squeezeDet for 5,000 iterations on X_{matrix}, evaluating it on T_{matrix} and T_{overlap}. To reduce the effect of jitter during training we used a standard technique [2], saving the last 10 models in steps of 10 iterations and picking the one achieving the best total precision and recall. This yielded the results in the first row of Tab. 6. Although X_{matrix} contains many images of overlapping cars, the precision on T_{overlap} is significantly lower than for T_{matrix}, indicating that the network is predicting lower-quality bounding boxes for such cars[7].

Table 6. Performance of models trained on 5,000 images from X_{matrix} or a mixture with X_{overlap}, averaged over 8 training runs with random selections of images from X_{matrix}.

Mixture %	T_{matrix} Precision	T_{matrix} Recall	T_{overlap} Precision	T_{overlap} Recall
100 / 0	72.9 ± 3.7	37.1 ± 2.1	62.8 ± 6.1	65.7 ± 4.0
95 / 5	73.1 ± 2.3	37.0 ± 1.6	68.9 ± 3.2	67.3 ± 2.4

Next we attempted to improve the effectiveness of the training set by mixing in the difficult images produced with Scenic. Specifically, we replaced a random 5% of X_{matrix} (250 images) with images from X_{overlap}, keeping the overall training set size constant. We then retrained the network on the new training set and evaluated it as above. To reduce the dependence on which images were replaced, we averaged over 8 training runs with different random selections of the 250 images to replace. The results are shown in the second row of Tab. 6. Even altering only 5% of the training set, performance on T_{overlap} significantly improves. Critically, the improvement on T_{overlap} is not paid for by a corresponding decrease on T_{matrix}: performance on the original data set remains the same. Thus, by allowing us to specify and generate instances of a difficult case, Scenic enables the generation of more effective training sets than can be obtained through simpler approaches not based on PPLs.

[6] We use the "Matrix" data set since it is known to be effective for car detection and was not designed by us, making the fact that Scenic is able to improve it more striking. The results of this experiment also hold under the Average Precision (AP) metric used in [24], as well as in a similar experiment using the Scenic generic two-car scenario from the last section as the baseline. See Appendix D of [13] for details.

[7] Recall is much *higher* on T_{overlap}, meaning the false-negative rate is better; this is presumably because all the T_{overlap} images have exactly 2 cars and are in that sense easier than the T_{matrix} images, which can have many cars.

Table 7. Performance of M_{generic} on different scenarios representing variations of a single misclassified image.

Scenario	Precision	Recall
(1) varying model and color	**80.3**	100
(2) varying background	50.5	99.3
(3) varying local position, orientation	62.8	100
(4) varying position but staying close	53.1	99.3
(5) any position, same apparent angle	58.9	98.6
(6) any position and angle	67.5	100
(7) varying background, model, color	61.3	100
(8) staying close, same apparent angle	52.4	100
(9) staying close, varying model	58.6	100

6.4 Debugging Failures

In our final experiment, we show how Scenic can be used to generalize a single input on which a model fails, exploring its neighborhood in a variety of different directions and giving insight into which features of the scene are responsible for the failure. The original failure can then be generalized to a broader scenario describing a class of inputs on which the model misbehaves, which can in turn be used for retraining. We selected one scene from our first experiment, consisting of a single car viewed from behind at a slight angle, which M_{generic} wrongly classified as three cars (thus having 33.3% precision and 100% recall). We wrote several scenarios which left most of the features of the scene fixed but allowed others to vary. Specifically, scenario (1) varied the model and color of the car, (2) left the position and orientation of the car relative to the camera fixed but varied the absolute position, effectively changing the background of the scene, and (3) used the mutation feature of Scenic to add a small amount of noise to the car's position, heading, and color (see Appendix A of [13] for code and the original misclassified image). For each scenario we generated 150 images and evaluated M_{generic} on them. As seen in Tab. 7, changing the model and color improved performance the most, suggesting they were most relevant to the misclassification, while local position and orientation were less important and global position (i.e. the background) was least important.

To investigate these possibilities further, we wrote a second round of variant scenarios, also shown in Tab. 7. The results confirmed the importance of model and color (compare (2) to (7)), as well as angle (compare (5) to (6)), but also suggested that being close to the camera could be the relevant aspect of the car's local position. We confirmed this with a final round of scenarios (compare (5) and (8)), which also showed that the effect of car model is small among scenes where the car is close to the camera (compare (4) and (9)).

Having established that car model, closeness to the camera, and view angle all contribute to poor performance of the network, we wrote broader scenarios capturing these features. To avoid overfitting, and since our experiments indicated car model was not very relevant when the car is close to the camera, we decided not to fix the car model. Instead, we specialized the generic one-car scenario from our first experiment to produce only cars close to the camera. We also created a second scenario specializing this further by requiring that the car be viewed at a shallow angle.

Table 8. Performance of M_{generic} after retraining, replacing 10% of X_{generic} with different data.

Replacement Data	Precision	Recall
Original (no replacement)	82.9	92.7
Classical augmentation	78.7	92.1
Close car	87.4	91.6
Close car at shallow angle	84.0	92.1

Finally, we used these scenarios to retrain M_{generic}, hoping to improve performance on its original test set T_{generic} (to better distinguish small differences in performance, we increased the test set size to 400 images). To keep the size of the training set fixed as in the previous experiment, we replaced 400 one-car images in X_{generic} (10% of the whole training set) with images generated from our scenarios. As a baseline, we used images produced with classical image augmentation techniques implemented in `imgaug` [25]. Specifically, we modified the original misclassified image by randomly cropping 10%–20% on each side, flipping horizontally with probability 50%, and applying Gaussian blur with $\sigma \in [0.0, 3.0]$.

The results of retraining M_{generic} on the resulting data sets are shown in Tab. 8. Interestingly, classical augmentation actually *hurt* performance, presumably due to overfitting to relatively slight variants of a single image. On the other hand, replacing part of the data set with specialized images of cars close to the camera significantly reduced the number of false positives like the original misclassification (while the improvement for the "shallow angle" scenario was less, perhaps due to overfitting to the restricted angle range). This demonstrates how Scenic can be used to improve performance by generalizing individual failures into scenarios that capture the essence of the problem but are broad enough to prevent overfitting during retraining.

7 Related Work

Data Generation and Testing for ML. There has been a large amount of work on generating synthetic data for specific applications, including text recognition [22], text localization [21], robotic object grasping [44], and autonomous driving [9, 24]. Closely related is work on *domain adaptation*, which attempts to correct differences between synthetic and real-world input distributions. Domain adaptation has enabled synthetic data to successfully train models for several

other applications including 3D object detection [28, 42], pedestrian detection [45], and semantic image segmentation [38]. Such work provides important context for our paper, showing that models trained exclusively on synthetic data (possibly domain-adapted) can achieve acceptable performance on real-world data. The major difference in our work is that we provide, through Scenic, language-based systematic data generation for *any* perception system.

Some works have also explored the idea of using adversarial examples (i.e. misclassified examples) to retrain and improve ML models (e.g., [17, 46, 49]). In particular, Generative Adversarial Networks (GANs) [16], a particular kind of neural network able to generate synthetic data, have been used to augment training sets [27, 29]. The difference with Scenic is that GANs require an initial training set/pretrained model and do not easily incorporate declarative constraints, while Scenic produces synthetic data in an explainable, programmatic fashion requiring only a simulator.

Model-Based Test Generation. Techniques using a model to guide test generation have long existed [3]. A popular approach is to provide *example tests*, as in mutational fuzz testing [43] and example-based scene synthesis [10]. While these methods are easy to use, they do not provide fine-grained control over the generated data. Another approach is to give *rules* or a *grammar* specifying how the data can be generated, as in generative fuzz testing [43], procedural generation from shape grammars [32], and grammar-based scene synthesis [23]. While grammars allow much greater control, they do not easily allow enforcing global properties. This is also true when writing a *program* in a domain-specific language with nondeterminism [8]. Conversely, *constraints* as in constrained-random verification [33] allow global properties but can be difficult to write. Scenic improves on these methods by simultaneously providing fine-grained control, enforcement of global properties, specification of probability distributions, and simple imperative syntax.

Probabilistic Programming Languages. The semantics (and to some extent, the syntax) of Scenic are similar to that of other probabilistic programming languages such as Prob [20], Church [18], and BLOG [31]. In probabilistic programming the focus is usually on *inference* rather than *generation* (the main application in our case), and in particular to our knowledge probabilistic programming languages have not previously been used for test generation. However, the most popular inference techniques are based on sampling and so could be directly applied to generate scenes from Scenic programs, as we discussed in Sec. 5.

Several probabilistic programming languages have been used to define generative models of objects and scenes: both general-purpose languages such as WebPPL [19] (see, e.g., [37]) and languages specifically motivated by such applications, namely Quicksand [36] and Picture [26]. The latter are in some sense the most closely-related to Scenic, although neither provides specialized syntax or semantics for dealing with geometry (Picture also was used only for inverse rendering, not data generation). The main advantage of Scenic over these languages is that its domain-specific design permits concise representation of complex scenarios and enables specialized sampling techniques.

8 Conclusion

In this paper, we introduced Scenic, a probabilistic programming language for specifying distributions over configurations of physical objects and agents. We showed how Scenic can be used to generate synthetic data sets useful for deep learning tasks. Specifically, we used Scenic to generate specialized test sets, improve the effectiveness of training sets by emphasizing difficult cases, and generalize from individual failure cases to broader scenarios suitable for retraining. In particular, by training on hard cases generated by Scenic, we were able to boost the performance of a car detector neural network (given a fixed training set size) significantly beyond what could be achieved by prior synthetic data generation methods [24] not based on PPLs.

In future work we plan to conduct experiments applying Scenic to a variety of additional domains, applications, and simulators. For example, we have integrated Scenic as the environment modeling language for VerifAI, a tool for the design and analysis of AI-based systems [6], and used it to generate seed inputs for temporal-logic falsification of an automated collision-avoidance system. We have also interfaced Scenic to the X-Plane flight simulator [35] in order to test ML-based aircraft navigation systems, and to the CARLA driving simulator [5] for scenarios requiring more control than GTAV provides. Finally, we plan to extend the Scenic language itself in several directions: allowing user-defined specifiers, describing 3D scenes, and encoding dynamic scenarios to aid in the analysis of complex dynamic behaviors, including both control as well as perception.

Acknowledgments

The authors would like to thank Ankush Desai, Alastair Donaldson, Andrew Gordon, Jonathan Ragan-Kelley, Sriram Rajamani, and several anonymous reviewers for helpful discussions and feedback. This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, NSF grants CNS-1545126 (VeHICaL), CNS-1646208, CNS-1739816, and CCF-1837132, DARPA under agreement number FA8750-16-C0043, the DARPA Assured Autonomy program, Berkeley Deep Drive, the iCyPhy center, and TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. 2016. Concrete Problems in AI Safety. *CoRR* abs/1606.06565 (2016). arXiv:1606.06565

[2] Sylvain Arlot and Alain Celisse. 2010. A survey of cross-validation procedures for model selection. *Statist. Surv.* 4 (2010), 40–79. https://doi.org/10.1214/09-SS054

[3] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science).* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4] Guillaume Claret, Sriram K Rajamani, Aditya V Nori, Andrew D Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 92–102.

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

[6] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. 2019. VerifAI: A Toolkit for the Design and Analysis of Artificial Intelligence-Based Systems. arXiv:1902.04245 https://github.com/BerkeleyLearnVerify/VerifAI

[7] DuPont. 2012. Global Automotive Color Popularity Report. https://web.archive.org/web/20130818022236/http://www2.dupont.com/Media_Center/en_US/color_popularity/Images_2012/DuPont2012ColorPopularity.pdf

[8] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. 2013. CONCURRIT: a domain specific language for reproducing concurrency bugs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 153–164.

[9] Artur Filipowicz, Jeremiah Liu, and Alain Kornhauser. 2017. *Learning to recognize distance to stop signs using the virtual world of Grand Theft Auto 5.* Technical Report. Princeton University.

[10] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. 2012. Example-based Synthesis of 3D Object Arrangements. In *ACM SIGGRAPH 2012 (SIGGRAPH Asia '12)*.

[11] Daniel Fremont, Xiangyu Yue, Tommaso Dreossi, Shromona Ghosh, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. 2018. *Scenic: Language-Based Scene Generation.* Technical Report UCB/EECS-2018-8. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-8.html

[12] Daniel J. Fremont, Alexandre Donzé, Sanjit A. Seshia, and David Wessel. 2015. Control Improvisation. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS) (LIPIcs)*, Vol. 45. 463–474.

[13] 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. arXiv:1809.09310 https://github.com/BerkeleyLearnVerify/Scenic

[14] Rockstar Games. 2015. Grand Theft Auto V. Windows PC version. https://www.rockstargames.com/games/info/V

[15] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

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

[18] Noah Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A universal language for generative models. In *Uncertainty in Artificial Intelligence 24 (UAI)*. 220–229.

[19] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2018-7-11.

[20] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *FOSE 2014*. ACM, 167–181.

[21] Ankush Gupta, Andrea Vedaldi, and Andrew Zisserman. 2016. Synthetic Data for Text Localisation in Natural Images. In *Computer Vision and Pattern Recognition, CVPR*. 2315–2324. https://doi.org/10.1109/CVPR.2016.254

[22] Max Jaderberg, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Synthetic Data and Artificial Neural Networks for Natural Scene Text Recognition. *CoRR* abs/1406.2227 (2014). arXiv:1406.2227

[23] Chenfanfu Jiang, Siyuan Qi, Yixin Zhu, Siyuan Huang, Jenny Lin, Lap-Fai Yu, Demetri Terzopoulos, and Song-Chun Zhu. 2018. Configurable 3D Scene Synthesis and 2D Image Rendering with Per-pixel Ground Truth Using Stochastic Grammars. *International Journal of Computer Vision* (2018), 1–22.

[24] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, Karl Rosaen, and Ram Vasudevan. 2017. Driving in the Matrix: Can virtual worlds replace human-generated annotations for real world tasks?. In *International Conference on Robotics and Automation, ICRA*. 746–753. https://doi.org/10.1109/ICRA.2017.7989092

[25] Alexander Jung. 2018. imgaug. https://github.com/aleju/imgaug

[26] Tejas Kulkarni, Pushmeet Kohli, Joshua B. Tenenbaum, and Vikash K. Mansinghka. 2015. Picture: A probabilistic programming language for scene perception. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4390–4399.

[27] Xiaodan Liang, Zhiting Hu, Hao Zhang, Chuang Gan, and Eric P Xing. 2017. Recurrent Topic-Transition GAN for Visual Paragraph Generation. *arXiv preprint arXiv:1703.07022* (2017).

[28] Joerg Liebelt and Cordelia Schmid. 2010. Multi-view object class detection with a 3D geometric model. In *Computer Vision and Pattern Recognition, CVPR*. 1688–1695. https://doi.org/10.1109/CVPR.2010.5539836

[29] Marco Marchesi. 2017. Megapixel Size Image Creation using Generative Adversarial Networks. *arXiv preprint arXiv:1706.00082* (2017).

[30] Olivier Michel. 2004. Webots: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems* 1, 1 (2004), 39–42.

[31] Brian Milch, Bhaskara Marthi, and Stuart Russell. 2004. BLOG: Relational modeling with unknown objects. In *ICML 2004 workshop on statistical relational learning and its connections to other fields*. 67–73.

[32] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. In *ACM Transactions On Graphics*, Vol. 25. ACM, 614–623.

[33] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan Marcus, and Gil Shurek. 2006. Constraint-Based Random Stimuli Generation for Hardware Verification. In *Proc. of AAAI*. 1720–1727.

[34] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs.. In *AAAI*. 2476–2482.

[35] Laminar Research. 2019. X-Plane 11. https://www.x-plane.com/

[36] Daniel Ritchie. 2014. Quicksand: A Lightweight Embedding of Probabilistic Programming for Procedural Modeling and Design. In *3rd NIPS Workshop on Probabilistic Programming*. https://dritchie.github.io/pdf/qs.pdf

[37] Daniel Ritchie. 2016. *Probabilistic programming for procedural modeling and design.* Ph.D. Dissertation. Stanford University. https://purl.stanford.edu/vh730bw6700

[38] Germán Ros, Laura Sellart, Joanna Materzynska, David Vázquez, and Antonio M. López. 2016. The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. In *Computer Vision and Pattern Recognition, CVPR*. 3234–3243. https:

//doi.org/10.1109/CVPR.2016.352

[39] Stuart Russell, Tom Dietterich, Eric Horvitz, Bart Selman, Francesca Rossi, Demis Hassabis, Shane Legg, Mustafa Suleyman, Dileep George, and Scott Phoenix. 2015. Letter to the Editor: Research Priorities for Robust and Beneficial Artificial Intelligence: An Open Letter. *AI Magazine* 36, 4 (2015).

[40] Nasser Saheb-Djahromi. 1978. Probabilistic LCF. In *Mathematical Foundations of Computer Science*. Springer, 442–451.

[41] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. 2016. Towards Verified Artificial Intelligence. arXiv:1606.08514

[42] Michael Stark, Michael Goesele, and Bernt Schiele. 2010. Back to the Future: Learning Shape Models from 3D CAD Data. In *British Machine Vision Conference, BMVC*. 1–11. https://doi.org/10.5244/C.24.106

[43] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley.

[44] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. 2017. Domain randomization for transferring deep neural networks from simulation to the real world. In *International Conference on Intelligent Robots and Systems, IROS*. 23–30. https://doi.org/10.1109/IROS.2017.8202133

[45] David Vazquez, Antonio M Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. 2014. Virtual and real world adaptation for pedestrian detection. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2014), 797–809.

[46] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. 2016. Understanding data augmentation for classification: when to warp?. In *Digital Image Computing: Techniques and Applications (DICTA), 2016 International Conference on*. IEEE, 1–6.

[47] Frank Wood, Jan Willem Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*. 1024–1032.

[48] Bichen Wu, Forrest N. Iandola, Peter H. Jin, and Kurt Keutzer. 2017. SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving. In *Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops*. 446–454. https://doi.org/10.1109/CVPRW.2017.60

[49] Yan Xu, Ran Jia, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. 2016. Improved relation classification by deep recurrent neural networks with data augmentation. *arXiv preprint arXiv:1601.03651* (2016).