Scenic: A Language for Scenario Specification and Scene Generation

Daniel J. Fremont
Logic and the Methodology of Science
University of California, Berkeley
dfremont@berkeley.edu

Tommaso Dreossi∗
Electrical Engineering and Computer Sciences
University of California, Berkeley
tommasodreossi@berkeley.edu

Shromona Ghosh∗
Electrical Engineering and Computer Sciences
University of California, Berkeley
shromona.ghosh@berkeley.edu

Xiangyu Yue∗
Electrical Engineering and Computer Sciences
University of California, Berkeley
xyyue@berkeley.edu

Alberto L. Sangiovanni-Vincentelli
Electrical Engineering and Computer Sciences
University of California, Berkeley
alberto@berkeley.edu

Sanjit A. Seshia
Electrical Engineering and Computer Sciences
Logic and the Methodology of Science
University of California, Berkeley
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 whose environment is a scene, a configuration of physical objects. 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.

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

ACM Reference Format:

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, 33, 35]. ML has proved particularly effective at perceptual tasks such as speech and vision. Thus, there is a pressing need to tackle three important challenges in the design of such ML-based perception systems:
• 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 [16, 17, 19, 38].

However, there is a serious problem with the use of synthetic data, namely that it can be highly non-trivial to generate meaningful data. 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. 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 [15]. However, the domain-specific design of SCENIC enables specialized sampling techniques: 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 [42], a convolutional neural network designed to perform object detection in autonomous cars. We implemented a sampler for SCENIC scenarios and used it to generate scenes which were rendered into images by Grand Theft Auto V (GTAV [9]), a video game with high-fidelity graphics. 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 [19] 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.

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;

• 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.

2 Using PPLs to Design and Analyze Perception Systems

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. Now we discuss in more detail how this process applies to the three design problems we described in the Introduction.

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 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. If we train the car-detection network on a generic dataset obtained by randomly driving around inside the simulated world of GTAV and capturing images periodically [19], as we might expect, 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, 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.
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 (which can be modified if needed). For example, we can write

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

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:

```python
1   m = Car.defaultModelDistribution.sample()
2   pos = curb.offsetLeft(0.5 + m.width / 2)
3   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,

```python
1   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:

```python
1   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.
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. 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:

```plaintext
1 car2 = Car offset by (-10, 10) @ (20, 40), with viewAngle 30 deg
2 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:

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

Hard requirements, called “observations” in other PPLs (see, e.g., [15]), are very convenient in our setting because they make it easy to restrict attention to particular cases of interest. This also improves 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:

```plaintext
1 taxi = Car at 120 @ 300, facing 37 deg, ...
2 ...
3 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.

**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 [25], with a bottleneck between the robot and its goal.
that forces the planner to consider climbing over a rock. The
Scenic code for this scenario is given in Appendix A.

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 inter-
esting 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.

4 Syntax of Scenic

Scenic is a simple object-oriented PPL, with programs con-
sisting of sequences of statements built with standard im-
perative 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 de-
pend 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, con-
  structed from coordinates in meters with the syntax
  \( X \@ Y \) (inspired by Smalltalk [10]).
- **Headings** representing orientations in space. Conve-
  niently, in 2D these are a single angle (in radians, anti-
  clockwise 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.

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

**Figure 5.** Simplified Scenic grammar. Point and Oriented-
Point are instances of the corresponding classes. See Tab. 5
for statements, Fig. 29 for operators, Tab. 1 for baseDist, and
Tables 3 and 4 for posSpec and headSpec.

**Table 1.** Distributions. All parameters scalars except value.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>((low, high))</td>
<td>uniform on interval</td>
</tr>
<tr>
<td>Uniform(value, ...)</td>
<td>uniform over values</td>
</tr>
<tr>
<td>Discrete({value: wt, ...})</td>
<td>discrete with weights</td>
</tr>
<tr>
<td>Normal(mean, stdDev)</td>
<td>normal with given ( \mu, \sigma )</td>
</tr>
</tbody>
</table>

**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 (e.g. the surface of an ob-
ject could have normal vectors, so that objects placed
randomly on the surface face outward by default).

In addition, Scenic provides **objects**, organized into single-
inheritance **classes** specifying a set of properties their in-
stances must have, together with corresponding default val-
ues (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** de-
sceds from the two other built-in classes: its superclass
is **OrientedPoint**, which in turn subclasses **Point**. These
represent locations in space, without and with an orienta-
tion respectively, and so provide the fundamental properties
position and heading. **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.
Table 2. Properties of Point, OrientedPoint, and Object.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>(0, 0)</td>
<td>position in global coords.</td>
</tr>
<tr>
<td>viewDistance</td>
<td>50</td>
<td>distance for ‘can see’</td>
</tr>
<tr>
<td>mutationScale</td>
<td>0</td>
<td>overall scale of mutations</td>
</tr>
<tr>
<td>positionStdDev</td>
<td>1</td>
<td>mutation $\sigma$ for position</td>
</tr>
<tr>
<td>heading</td>
<td>0</td>
<td>heading in global coords.</td>
</tr>
<tr>
<td>viewAngle</td>
<td>360°</td>
<td>angle for ‘can see’</td>
</tr>
<tr>
<td>headingStdDev</td>
<td>5°</td>
<td>mutation $\sigma$ for heading</td>
</tr>
<tr>
<td>width</td>
<td>1</td>
<td>width of bounding box</td>
</tr>
<tr>
<td>height</td>
<td>1</td>
<td>height of bounding box</td>
</tr>
<tr>
<td>allowCollisions</td>
<td>false</td>
<td>collisions allowed</td>
</tr>
<tr>
<td>requireVisible</td>
<td>true</td>
<td>must be visible from ego</td>
</tr>
</tbody>
</table>

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.

4.2 Expressions

Scenic’s expressions are mostly straightforward, largely consisting of arithmetic, boolean, and geometric operators seen above. A complete list of Scenic’s operators, with definitions, is in Appendix C. 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 circular sector along its heading with a certain angle (see Table 2). An Object is visible if and only if part of 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

\[
\begin{align*}
1 & \quad x = (0, 1) \\
2 & \quad y = x \circ x
\end{align*}
\]

does not make y uniform over the unit box, but rather over its diagonal. For convenience in sampling multiple times from a distribution, Scenic provides a \texttt{resample(x)} function returning an independent sample from the distribution of x.

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 \texttt{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 \texttt{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 optionally, meaning that other specifiers will override them. For example, on \texttt{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. We briefly discuss some of the more complex specifiers, referring to the examples in Fig. 6:

- **behind** object means the object is placed with the mid-point of its front edge at the given vector, and similarly for **ahead/left/right of** object.
- **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 left is oriented as P). Similarly, the heading optionally specified by following **vectorField** is that of the vector field at the specified position.
- **apparently facing** means the object has heading with respect to the line of sight from ego. For example, **apparently facing 90°** would orient the object so that the camera views its left side head-on.

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

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>at vector</td>
<td>—</td>
</tr>
<tr>
<td>offset by vector</td>
<td>—</td>
</tr>
<tr>
<td>offset along direction by vector</td>
<td>—</td>
</tr>
<tr>
<td>(left</td>
<td>right) of vector [by scalar]</td>
</tr>
<tr>
<td>(ahead of</td>
<td>behind) vector [by scalar]</td>
</tr>
<tr>
<td>beyond vector by vector [from vector]</td>
<td>—</td>
</tr>
<tr>
<td>visible [from (Point</td>
<td>OrientedPoint)]</td>
</tr>
<tr>
<td>(in</td>
<td>on) region</td>
</tr>
<tr>
<td>(left</td>
<td>right) of (OrientedPoint</td>
</tr>
<tr>
<td>(ahead of</td>
<td>behind) (OrientedPoint</td>
</tr>
<tr>
<td>following vectorField [from vector] for scalar</td>
<td>—</td>
</tr>
</tbody>
</table>

### Table 4. Specifiers for heading.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Deps.</th>
</tr>
</thead>
<tbody>
<tr>
<td>facing heading</td>
<td>—</td>
</tr>
<tr>
<td>facing vectorField</td>
<td>position</td>
</tr>
<tr>
<td>facing (toward</td>
<td>away from) vector</td>
</tr>
<tr>
<td>apparently facing heading [from vector]</td>
<td>position</td>
</tr>
</tbody>
</table>

### Table 5. Statements.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier = value</td>
<td>var. assignment</td>
</tr>
<tr>
<td>param identifier = value, ...</td>
<td>param. assignment</td>
</tr>
<tr>
<td>classDefn</td>
<td>class definition</td>
</tr>
<tr>
<td>instance</td>
<td>object definition</td>
</tr>
<tr>
<td>require boolean</td>
<td>hard requirement</td>
</tr>
<tr>
<td>require[number] boolean</td>
<td>soft requirement</td>
</tr>
<tr>
<td>mutate identifier, ... [by number]</td>
<td>enable mutation</td>
</tr>
</tbody>
</table>

### 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 identifier = 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, 26]). 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 [34]. 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., [15]). 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 for a more formal treatment.

SoR 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 described in the previous section, each specifier defines a function mapping values for its dependencies to values for the properties it specifies. The functions for the built-in specifiers were described above, and are given precisely in Appendix C. When an object of class \( C \) is constructed using a set of specifiers \( S \), the object is defined according to the following procedure (stated formally in Appendix B):

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 next.

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 two such techniques below, for scenarios placing objects with respect to a vector field which is constant within polygonal regions (such as our roads). We defer formal statements of the algorithms to Appendix B.

Pruning Based on Orientation. The first technique applies to scenarios placing constraints on relative heading and maximum distance between objects \( X \) and \( Y \). Let \( map \) be a collection of polygons with directions, \( A \) be the range of allowed relative headings, \( maxDist \) be the allowed maximum distance, and \( \delta \) the maximum allowed heading perturbation from the polygon direction. For each polygon \( P \), the algorithm finds portions of polygons \( Q \) in the \( maxDist \)-neighborhood of \( P \). Then, it checks if the heading constraint is satisfied by verifying that the relative heading \( relHead(P, Q) \) plus the worst case perturbation \( 2\delta \) falls in the given range \( A \) (\( 2\delta \) occurs when \( \delta \) is applied to both objects being placed).

This procedure only prunes regions of \( map \) where it is impossible to satisfy the heading and distance constraints. Suppose it is possible to put object \( X \) at \( x \) in polygon \( P \). Then object \( Y \) must lie at some \( y \) in another polygon \( Q \) with relative heading and maximum perturbation \( \delta \) in \( A \). Since the distance from \( x \) to \( y \) is at most \( maxDist \), \( x \) is in the dilation of \( Q \) by \( maxDist \) and therefore \( x \in P \cap Q' \).

Pruning Based on Size. We can also prune the space for scenarios where we can compute a lower bound \( minWidth \) 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. An improved sampler for this case begins by extracting from the \( map \) the polygons that are not wide enough to fit the configuration: call these “narrow”. For each narrow polygon \( P \), it considers the union
of all the other polygons $Q$ dilated by $\text{maxDist}$ and adds to the pruned map the intersection of $P$ with $U$.

This procedure also only prunes impossible regions of the space. Suppose it is possible to put object $X$ at $x$ in polygon $P$. $P$ is either large enough or too narrow. In the latter case, object $Y$ must lie at $y$ in some another polygon $Q$. Since the distance from $x$ to $y$ is at most $\text{maxDist}$, $x$ is in the dilation of $Q$ by $\text{maxDist}$ and therefore $x \in P \cap U$.

After pruning the space as described above, our implementation uses a simple rejection sampling approach, generating scenes from the imperative part of the scenario until all requirements are satisfied. While these 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. In future work it would be interesting to see whether Markov chain Monte Carlo (MCMC) methods previously used for probabilistic programming (see, e.g., [26, 29, 41]) could be made effective in the case of SCENIC.

6 Experiments

We experimented with three different uses of SCENIC: testing the accuracy of an ML system under particular conditions, training the system to improve accuracy in hard cases, and debugging a known failure case.

6.1 Experimental Setup

We generated scenes in the virtual world of the video game Grand Theft Auto V (GTAV) [9]. We wrote a SCENIC library gta1.lib defining Regions representing the roads and curbs in (part of) this world, as well as a type of object Car providing two additional properties\(^2\): model, representing the type of car, with uniform distribution over 13 diverse models, and color, representing the car color, with a default distribution based on real-world car color statistics [5]. 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”).

Our experiments used squeezeDet [42], a convolutional neural network real-time object detector for autonomous driving. We used a batch size of 20 and trained all models for 10,000 iterations unless otherwise noted. Images captured from GTAV with resolution $1920 \times 1200$ were resized to $1248 \times 384$, the resolution used by squeezeDet. All models were trained and evaluated on NVIDIA TITAN Xp GPUs.

\(^2\)For the full definition of Car, see Appendix A; the definitions of road, curb, etc. are a few lines loading the corresponding sets of points from a map file generated as described below.

We used standard metrics precision and recall to measure the accuracy of a prediction on a particular image. The accuracy is computed based on how well the network predicts the correct bounding box, score, and category of objects in the image. Details are in the appendix, 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. We use average precision and recall to evaluate the performance of a model on a collection of images constituting a test set.

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 $10^\circ$ of the road direction. We generated 1,000 images from each scenario, yielding a training set $T_{\text{generic}}$ of 4,000 images, and used these to train a model $M_{\text{generic}}$ as described in Sec. 6.1. We also generated an additional 50 images from each scenario to obtain a generic test set $T_{\text{generic}}$ 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 $T_{\text{good}}$ and $T_{\text{bad}}$.

Evaluating $M_{\text{generic}}$ on $T_{\text{generic}}$, $T_{\text{good}}$, and $T_{\text{bad}}$, we obtained average precisions of 86.1%, 88.5%, and 78.9%, respectively, and average recalls of 94.5%, 96.1%, and 95.0%. 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. 7, which generates such scenes by
We trained squeezeDet for 5,000 iterations on Table 6, which yielded the same overall conclusion. See Appendix D for details.

4 are in that sense easier than the T this is presumably because all the Scenic detection and was not designed by us, making the fact that Scenic is able to improve it more striking. We also performed a similar experiment using the Scenic generic two-car scenario from the last section as the baseline, which yielded the same overall conclusion. See Appendix D for details.

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.8). Generating images from this scenario we obtained a training set Xoverlap of 250 images and a test set Toverlap of 200 images.

For a baseline training set we used the “Driving in the Matrix” synthetic data set [19], which has been used to train a car detection network achieving good performance even on real-world images3. Like our images, the “Matrix” images were rendered in GTA V; however, they were produced by allowing the game’s AI to drive around while periodically taking screenshots. We randomly selected 5,000 of these images to form a training set Xmatrix, and 200 for a test set Tmatrix. We trained squeezeDet for 5,000 iterations on Xmatrix and evaluated it on both Tmatrix and Toverlap. To reduce the effect of jitter during training we used a standard technique [2], saving the last 50 models obtained in steps of 10 iterations and finding the best precision and recall respectively obtained by any of these models. This yielded the results in the first row of Table 6. Although Xmatrix contains many images of overlapping cars, the precision on Toverlap is significantly lower than for Tmatrix, indicating that the network is predicting lower-quality bounding boxes for such cars4.

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 Xmatrix (250 images) with images from Xoverlap, 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 additionally averaged over 8 training runs with different random selections of the 250 images. The results are shown in the second row of Table 6. Even altering only 5% of the training set, performance on Toverlap dramatically improves. Critically, the improvement on Toverlap is not paid for by a corresponding decrease on Tmatrix; on the contrary, performance on the original data set actually becomes significantly better. Thus, by emphasizing difficult cases, we were able to improve the training set’s effectiveness not only for such cases but even for the “typical” distribution it was originally obtained from.

### 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 gaining insight into which features of the scene are responsible for the failure. We selected one scene from our first experiment, consisting of a single car viewed from behind at a slight angle, which Mgeneric 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.6 for code and the original misclassified image). For each scenario we generated 150 images and evaluated Mgeneric 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

1 wiggle = (−10 deg, 10 deg)
2 ego = Car with roadDeviation wiggle
3 c = Car visible, \n4 with roadDeviation resample(wiggle)
5 leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
6 Car beyond c by leftRight @ (4, 10), \n7 with roadDeviation resample(wiggle)

Figure 7. A scenario where one car partially occludes another. The property roadDeviation is defined in Car to mean its heading relative to the roadDirection.

### Table 6. Performance of models trained on 5,000 images from Xmatrix or a mixture with Xoverlap. The results for the mixture were averaged over 8 training runs with different random selections of which images in Xmatrix to replace.

<table>
<thead>
<tr>
<th>Training Data</th>
<th>Xmatrix / Xoverlap</th>
<th>Tmatrix Precision</th>
<th>Tmatrix Recall</th>
<th>Toverlap Precision</th>
<th>Toverlap Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% / 0%</td>
<td>66.2 / 60.9</td>
<td>62.6 / 76.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95% / 5%</td>
<td>72.2 / 62.0</td>
<td>78.8 / 78.8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 7. Performance of Mgeneric on different scenarios representing variations of a single misclassified image.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) varying model and color</td>
<td>88.4</td>
<td>100</td>
</tr>
<tr>
<td>(2) varying background</td>
<td>63.7</td>
<td>100</td>
</tr>
<tr>
<td>(3) varying local position, orientation</td>
<td>74.2</td>
<td>100</td>
</tr>
<tr>
<td>(4) varying position but staying close</td>
<td>68.8</td>
<td>99.3</td>
</tr>
<tr>
<td>(5) any position, same apparent angle</td>
<td>74.3</td>
<td>98.6</td>
</tr>
<tr>
<td>(6) any position and angle</td>
<td>81.2</td>
<td>100</td>
</tr>
<tr>
<td>(7) varying background, model, color</td>
<td>74.4</td>
<td>100</td>
</tr>
<tr>
<td>(8) staying close, same apparent angle</td>
<td>64.1</td>
<td>100</td>
</tr>
<tr>
<td>(9) staying close, varying model</td>
<td>71.7</td>
<td>100</td>
</tr>
</tbody>
</table>

3We 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. We also performed a similar experiment using the Scenic generic two-car scenario from the last section as the baseline, which yielded the same overall conclusion. See Appendix D for details.

4Recall is much higher on Toverlap, meaning the false-negative rate is better; this is presumably because all the Tmatrix images have exactly 2 cars and are in that sense easier than the Tmatrix images, which can have many cars.

PLODI’19, June 24–26, 2019, Phoenix, AZ, USA
we used images produced with classical image augmentation with images generated from our scenarios. As a baseline, we specialized the generic one-car scenario from our first experiment 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.

Finally, we used these scenarios to retrain $M_{\text{generic}}$, hoping to improve performance on its original test set $T_{\text{generic}}$ (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 $X_{\text{generic}}$ (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 [20]. Specifically, we modified the original missclassified 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 $M_{\text{generic}}$ 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 improved performance (while the improvement for the "shallow angle" scenario was less, perhaps due to overfitting to the restricted angle range). This demonstrates how $\text{Scenic}$ can be used to improve performance by generalizing individual misclassifications into scenarios that capture the essence of the problem but are broad enough to prevent overfitting during retraining.

<table>
<thead>
<tr>
<th>Replacement Data</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no replacement)</td>
<td>85.9</td>
<td>94.8</td>
</tr>
<tr>
<td>Classical augmentation</td>
<td>82.6</td>
<td>94.4</td>
</tr>
<tr>
<td>Close car</td>
<td>89.8</td>
<td>94.0</td>
</tr>
<tr>
<td>Close car at shallow angle</td>
<td>87.6</td>
<td>94.8</td>
</tr>
</tbody>
</table>

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

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 [17], text localization [16], robotic object grasping [38], and autonomous driving [7, 19]. 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 [23, 36], pedestrian detection [39], and semantic image segmentation [32]. 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 $\text{Scenic}$, 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., [12, 40, 43]). In particular, Generative Adversarial Networks (GANs) [11], a particular kind of neural network able to generate synthetic data, have been used to augment training sets [22, 24]. The difference with $\text{Scenic}$ is that GANs require an initial training set/pretrained model and do not easily incorporate declarative constraints, while $\text{Scenic}$ produces synthetic data in an explainable, programmatic fashion requiring only a simulator.

Model-Based Test Generation: Techniques to use a model to guide test generation have long existed [3]. A popular approach is to provide example outputs, as in mutational fuzz testing [37] and example-based scene synthesis [8]. 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 [37], procedural generation from shape grammars [27], and grammar-based scene synthesis [18]. 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 [6]. Conversely, constraints as in constrained-random verification [28] allow global properties but can be difficult to write. $\text{Scenic}$ 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 $\text{Scenic}$ are similar to that of other probabilistic programming languages such as Pton [15], Church [13], and BLOG [26]. 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 $\text{Scenic}$ 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 [14] (see, e.g.,
languages specifically motivated by such applications, namely Quicksand [30] and Picture [21]. The latter are in some sense the most closely-related to SCENIC, although neither provides specialized syntax or semantics for dealing with geometry. 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. 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 significantly beyond what could be achieved by prior synthetic data generation methods [19] given a fixed training set size. In future work, we plan more experiments as well as extending the SCENIC language in several directions: allowing user-defined specifiers, describing 3D scenes, and encoding dynamics to enable the analysis of controllers as well as perception systems.

Acknowledgments
The authors would like to thank Ankush Desai and Jonathan Ragan-Kelley for helpful discussions. 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 and CNS-1739816, DARPA under agreement number FA8750-16-c-0043, MARCO and DARPA. This work is supported by MARCO and DARPA.

References


A Gallery of Scenarios

This section presents Scenic code for a variety of scenarios from our autonomous car case study (and the robot motion planning example used in Sec. 3), along with images rendered from them. The scenarios range from simple examples used above to illustrate different aspects of the language, to those representing interesting road configurations like platoons and lanes of traffic.

Contents

A.1 The gtaLib Module 15
A.2 The Simplest Possible Scenario 16
A.3 A Single Car 17
A.4 A Badly-Parked Car 18
A.5 An Oncoming Car 19
A.6 Adding Noise to a Scene 20
A.7 Two Cars 21
A.8 Two Overlapping Cars 22
A.9 Four Cars, in Poor Driving Conditions 23
A.10 A Platoon, in Daytime 24
A.11 Bumper-to-Bumper Traffic 25
A.12 Robot Motion Planning with a Bottleneck 27

A.1 The gtaLib Module

All the scenarios below begin with a line (not shown here) importing the gtaLib module, which as explained above contains all definitions specific to our autonomous car case study. These include the definitions of the regions road and curb, as well as the vector field roadDirection giving the prevailing traffic direction at each point on the road. Most importantly, it also defines Car as a type of object:

```
1 class Car:
2     position: Point on road
3     heading: (roadDirection at self.position) \n4         + self.roadDeviation
5     roadDeviation: 0
6     width: self.model.width
7     height: self.model.height
8     viewAngle: 80 deg
9     visibleDistance: 30
10    model: CarModel.defaultModel()
11    color: CarColor.defaultColor()
```

Most of the properties are inherited from Object or are self-explanatory. The property roadDeviation, representing the heading of the car with respect to the local direction of the road, is purely a syntactic convenience; the following two lines are equivalent:

```
1 Car facing 10 deg relative to roadDirection
2 Car with roadDeviation 10 deg
```

The gtaLib library also defines a few convenience sub-classes of Car with different default properties. For example, EgoCar overrides model with the fixed car model we used for the ego car in our interface to GTA V.
A.2 The Simplest Possible Scenario
This scenario, creating a single car with no specified properties, was used as an example in Sec. 3.

1 ego = Car
2 Car

Figure 8. Scenes generated from a Scenic scenario representing a single car (with reasonable default properties).
A.3  A Single Car

This scenario is slightly more general than the previous, allowing the car (and the ego car) to deviate from the road direction by up to 10°. It also specifies that the car must be visible, which is in fact redundant since this constraint is built into Scenic, but helps guide the sampling procedure. This scenario was also used as an example in Sec. 3.

```plaintext
1 wiggle = (-10 deg, 10 deg)
2 ego = EgoCar with roadDeviation wiggle
3 Car visible, with roadDeviation resample(wiggle)
```

Figure 9. Scenes generated from a Scenic scenario representing a single car facing roughly the road direction.
A.4 A Badly-Parked Car

This scenario, creating a single car parked near the curb but not quite parallel to it, was used as an example in Sec. 3.

```
1 ego = Car
2 spot = OrientedPoint on visible curb
3 badAngle = Uniform(1.0, -1.0) * (10, 20) deg
4 Car left of spot by 0.5, facing badAngle relative to roadDirection
```
A.5 An Oncoming Car

This scenario, creating a car 20–40 m ahead and roughly facing towards the camera, was used as an example in Sec. 3. Note that since we do not specify the orientation of the car when creating it, the default heading is used and so it will face the road direction. The require statement then requires that this orientation is also within 15° of facing the camera (as the view cone is 30° wide).

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

Figure 11. Scenes generated from a SCENIC scenario representing a car facing roughly towards the camera.
A.6 Adding Noise to a Scene

This scenario, using Scenic’s mutation feature to automatically add noise to an otherwise completely-specified scenario, was used in the experiment in Sec. 6.4 (it is Scenario (3) in Table 7). The original scene, which is exactly reproduced by this scenario if the mutate statement is removed, is shown in Fig. 13.

```
1 param time = 12 * 60 # noon
2 param weather = 'EXTRASUNNY'
3
ego = EgoCar at -628.7878 @ -540.6067, \n   facing -359.1691 deg
4 Car at -625.4444 @ -530.7654, facing 8.2872 deg, \n   with model CarModel.models['DOMINATOR'], \n   with color CarColor.byteToReal([187, 162, 157])
5
6 mutate
```

Figure 13. The original misclassified image in Sec. 6.4.

Figure 12. Scenes generated from a Scenic scenario adding noise to the scene in Fig. 13.
A.7 Two Cars
This is the generic two-car scenario used in the experiments in Secs. 6.2 and 6.3.

1 wiggle = (-10 deg, 10 deg)
2 ego = EgoCar with roadDeviation wiggle
3 Car visible, with roadDeviation resample(wiggle)
4 Car visible, with roadDeviation resample(wiggle)
A.8 Two Overlapping Cars

This is the scenario used to produce images of two partially-overlapping cars for the experiment in Sec. 6.3.

```plaintext
1 wiggle = (-10 deg, 10 deg)
2 ego = EgoCar with roadDeviation wiggle
3 c = Car visible, with roadDeviation resample(wiggle)
4 leftRight = Uniform(1.0, -1.0) * (1.25, 2.75)
5 Car beyond c by leftRight @ (4, 10), with roadDeviation resample(wiggle)
```

![Figure 15. Scenes generated from a SCENIC scenario representing two cars, one partially occluding the other.](image-url)
A.9 Four Cars, in Poor Driving Conditions

This is the scenario used to produce images of four cars in poor driving conditions for the experiment in Sec. 6.2. Without the first two lines, it is the generic four-car scenario used in that experiment.

```plaintext
1 param weather = 'RAIN'
2 param time = 0 * 60  # midnight
3 wiggle = (-10 deg, 10 deg)
4 ego = EgoCar with roadDeviation wiggle
5 Car visible, with roadDeviation resample(wiggle)
6 Car visible, with roadDeviation resample(wiggle)
7 Car visible, with roadDeviation resample(wiggle)
8 Car visible, with roadDeviation resample(wiggle)
9 Car visible, with roadDeviation resample(wiggle)
```

Figure 16. Scenes generated from a Scenic scenario representing four cars in poor driving conditions.
A.10 A Platoon, in Daytime

This scenario illustrates how Scenic can construct structured object configurations, in this case a platoon of cars. It uses a helper function provided by gtaLib for creating platoons starting from a given car, shown in Fig. 17. If no argument model is provided, as in this case, all cars in the platoon have the same model as the starting car; otherwise, the given model distribution is sampled independently for each car. The syntax for functions and loops supported by our Scenic implementation is inherited from Python.

```
1 param time = (8, 20) * 60 # 8 am to 8 pm
2 ego = Car with visibleDistance 60
3 c2 = Car visible
4 platoon = createPlatoonAt(c2, 5, dist=(2, 8))
```

```
1 def createPlatoonAt(car, numCars, model=None, dist=(2, 8), shift=(-0.5, 0.5), wiggle=0):
2   lastCar = car
3   for i in range(numCars-1):
4     center = follow roadDirection from (front of lastCar) for resample(dist)
5     pos = OrientedPoint right of center by shift, \
6     facing resample(wiggle) relative to roadDirection
7     lastCar = Car ahead of pos, with model (car.model if model is None else resample(model))
```

**Figure 17.** Helper function for creating a platoon starting from a given car.

**Figure 18.** Scenes generated from a Scenic scenario representing a platoon of cars during daytime.
A.11 Bumper-to-Bumper Traffic

This scenario creates an even more complex type of object structure, namely three lanes of traffic. It uses the helper function `createPlatoonAt` discussed above, plus another for placing a car ahead of a given car with a specified gap in between, shown in Fig. 19.

```python
1  depth = 4
2  laneGap = 3.5
3  carGap = (1, 3)
4  laneShift = (-2, 2)
5  wiggle = (-5 deg, 5 deg)
6  modelDist = CarModel.defaultModel()
7  def createLaneAt(car):
8      createPlatoonAt(car, depth, dist=carGap, wiggle=wiggle, model=modelDist)
9  ego = Car with visibleDistance 60
10  leftCar = carAheadOfCar(ego, laneShift + carGap, offsetX=-laneGap, wiggle=wiggle)
11  createLaneAt(leftCar)
12  midCar = carAheadOfCar(ego, resample(carGap), wiggle=wiggle)
13  createLaneAt(midCar)
14  rightCar = carAheadOfCar(ego, resample(laneShift) + resample(carGap), offsetX=laneGap, wiggle=wiggle)
15  createLaneAt(rightCar)

1  def carAheadOfCar(car, gap, offsetX=0, wiggle=0):
2      pos = OrientedPoint at (front of car) offset by (offsetX @ gap), \
3            facing resample(wiggle) relative to roadDirection
4      return Car ahead of pos
```

**Figure 19.** Helper function for placing a car ahead of a car, with a specified gap in between.
Figure 20. Scenes generated from a Scenic scenario representing bumper-to-bumper traffic.
### A.12 Robot Motion Planning with a Bottleneck

This scenario illustrates the use of Scenic in another domain (motion planning) and with another simulator (Webots [25]). Figure 21 encodes a scenario representing a rubble field of rocks and pipes with a bottleneck between a robot and its goal that forces the path planner to consider climbing over a rock. The code is broken into four parts: first, we import a small library defining the workspace and the types of objects, then create the robot at a fixed position and the goal (represented by a flag) at a random position on the other side of the workspace. Second, we pick a position for the bottleneck, requiring it to lie roughly on the way from the robot to its goal, and place a rock there. Third, we position two pipes of varying lengths which the robot cannot climb over on either side of the bottleneck, with their ends far enough apart for the robot to be able to pass between. Finally, to make the scenario slightly more interesting we add several additional obstacles, positioned either on the far side of the bottleneck or anywhere at random. Several resulting workspaces are shown in Fig. 22.

```python
import mars
ego = Rover at 0 @ -2
goal = Goal at (-2, 2) @ (2, 2.5)

halfGapWidth = (1.2 * ego.width) / 2
bottleneck = OrientedPoint offset by (-1.5, 1.5) @ (0.5, 1.5), facing (-30, 30) deg
require abs((angle to goal) - (angle to bottleneck)) <= 10 deg
BigRock at bottleneck

leftEnd = OrientedPoint left of bottleneck by halfGapWidth, facing (60, 120) deg relative to bottleneck
rightEnd = OrientedPoint right of bottleneck by halfGapWidth, facing (-120, -60) deg relative to bottleneck
Pipe ahead of leftEnd, with height (1, 2)
Pipe ahead of rightEnd, with height (1, 2)
BigRock beyond bottleneck by (-0.5, 0.5) @ (0.5, 1)
BigRock beyond bottleneck by (-0.5, 0.5) @ (0.5, 1)
Pipe
Rock
Rock
Rock
```

**Figure 21.** A Scenic representing rubble fields with a bottleneck so that the direct route to the goal requires climbing over rocks.
Figure 22. Workspaces generated from the scenario in Fig. 21, viewed in Webots from a fixed camera.
We will precisely define the meaning of Scenic. As explained in the previous section, e evaluates to the value (a map from variables to values), dependencies and the scenario is ill-formed (e.g. Car left of 0 @ 0, facing roadDirection). Construct a directed graph with vertices p and edges to p from each of the dependencies of sp (if a dependency is not in P, then a specifier references a nonexistent property and the scenario is ill-formed). If this graph has a cycle, there are cyclic dependencies and the scenario is ill-formed (e.g. Car left of 0 @ 0, facing roadDirection: the heading must be known.

B.2 Semantics of Expressions

As explained in the previous section, Scenic’s expressions are straightforward except for distributions and object definitions. As in a typical imperative probabilistic programming language, a distribution evaluates to a sample from the distribution, following the first rule in Fig. 23. For example, if baseDist is a uniform interval distribution and the parameters evaluate to low = 0 and high = 1, then the distribution can evaluate to any value in [0, 1] with probability density 1.

The semantics of object definitions are given by the second rule in Fig. 23. First note the side effect, namely adding the newly-defined object to the set O. The premises of the rule describe the procedure for combining the specifiers to obtain the overall set of properties for the object. The main step is working out the evaluation order for the specifiers so that all their dependencies are satisfied, as well as deciding for each specifier which properties it should specify (if it specifies a property optionally, another specifier could take precedence). This is done by the procedure resolveSpecifiers, shown formally as Alg. 1 and which essentially does the following:

Let P be the set of properties defined in the object’s class and superclasses, together with any properties specified by any of the specifiers. The object will have exactly these properties, and the value of each p ∈ P is determined as follows. If p is specified non-optionally by multiple specifiers the scenario is ill-formed. If p is only specified optionally, and by multiple specifiers, this is ambiguous and we also declare the scenario ill-formed. Otherwise, the value of p will be determined by its unique non-optional specifier, unique optional specifier, or the most-derived default value, in that order: call this specifier sp.

Figure 23. Semantics of expressions (excluding operators, defined in Appendix B). Here baseDist is viewed as a function mapping parameters θ to a distribution with density function Pθ, and newInstance(class, props) creates a new instance of a class with the given property values. 

B Semantics of Scenic

In this section we give a precise semantics for Scenic expressions and statements, building up to a semantics for a complete program as a distribution over scenes.

B.1 Notation for State and Semantics

We will precisely define the meaning of Scenic language constructs by giving a small-step operational semantics. We will focus on the aspects of Scenic that set it apart from ordinary imperative languages, skipping standard inference rules for sequential composition, arithmetic operations, etc. that we essentially use without change. In rules for statements, we will denote a state of a Scenic program by (s, σ, π, O), where s is the statement to be executed, σ is the current variable assignment (a map from variables to values), π is the current global parameter assignment (for param statements), and O is the set of all objects defined so far. In rules for expressions, we use the same notation, although we sometimes suppress the state on the right-hand side of rules for expressions without side effects: (e, σ, π, O) → v means that in the state (σ, π, O), the expression e evaluates to the value v without side effects.

Since Scenic is a probabilistic programming language, a single expression can be evaluated different ways with different probabilities. Following the notation of [4, 34], we write →p for a rewrite rule that fires with probability p (probability density p, in the case of continuous distributions). We will discuss the meaning of such rules in more detail below.

B.2 Semantics of Expressions

As explained in the previous section, Scenic’s expressions are straightforward except for distributions and object definitions. As in a typical imperative probabilistic programming language, a distribution evaluates to a sample from the distribution, following the first rule in Fig. 23. For example, if baseDist is a uniform interval distribution and the parameters evaluate to low = 0 and high = 1, then the distribution can evaluate to any value in [0, 1] with probability density 1.

The semantics of object definitions are given by the second rule in Fig. 23. First note the side effect, namely adding the newly-defined object to the set O. The premises of the rule describe the procedure for combining the specifiers to obtain the overall set of properties for the object. The main step is working out the evaluation order for the specifiers so that all their dependencies are satisfied, as well as deciding for each specifier which properties it should specify (if it specifies a property optionally, another specifier could take precedence). This is done by the procedure resolveSpecifiers, shown formally as Alg. 1 and which essentially does the following:

Let P be the set of properties defined in the object’s class and superclasses, together with any properties specified by any of the specifiers. The object will have exactly these properties, and the value of each p ∈ P is determined as follows. If p is specified non-optionally by multiple specifiers the scenario is ill-formed. If p is only specified optionally, and by multiple specifiers, this is ambiguous and we also declare the scenario ill-formed. Otherwise, the value of p will be determined by its unique non-optional specifier, unique optional specifier, or the most-derived default value, in that order: call this specifier sp.

Construct a directed graph with vertices P and edges to p from each of the dependencies of sp (if a dependency is not in P, then a specifier references a nonexistent property and the scenario is ill-formed). If this graph has a cycle, there are cyclic dependencies and the scenario is ill-formed (e.g. Car left of 0 @ 0, facing roadDirection: the heading must be known.
Algorithm 1 resolveSpecifiers (class, specifiers)

▷ gather all specified properties
1: specForProperty ← ∅
2: optionalSpecsForProperty ← ∅

for all specifiers $S$ in specifiers do
4: for all properties $P$ specified non-optionally by $S$ do
5: if $P ∈ \text{dom } specForProperty$ then
6: syntax error: property $P$ specified twice
7: specForProperty $(P) ← S$
8: for all properties $P$ specified optionally by $S$ do
9: optionalSpecsForProperty $(P).\text{append}(S)$

▷ filter optional specifications
10: for all properties $P ∈ \text{dom } optionalSpecsForProperty$ do
11: if $|\text{optionalSpecsForProperty}(P)| > 1$ then
12: syntax error: property $P$ specified twice
13: specForProperty $(P) ← \text{optionalSpecsForProperty}(P)[0]$

▷ add default specifiers as needed
14: defaults ← defaultValueExpressions (class)
15: for all properties $P ∈ \text{dom } defaults$ do
16: if $P ∉ \text{dom } specForProperty$ then
17: specForProperty $(P) ← defaults(P)$

▷ build dependency graph
18: $G ← \text{empty graph on } \text{dom } specForProperty$
19: for all specifiers $S ∈ \text{dom } specForProperty$ do
20: for all dependencies $D$ of $S$ do
21: if $D ∉ \text{dom } specForProperty$ then
22: syntax error: missing property $D$ required by $S$
23: add an edge in $G$ from $\text{specForProperty}(D)$ to $S$

if $G$ is cyclic then
24: syntax error: specifiers have cyclic dependencies

▷ construct specifier and property evaluation order
25: specsAndProps ← empty list
26: for all specifiers $S$ in $G$ in topological order do
27: specsAndProps.append(((S, {$P | \text{specForProperty}(P) = S$}))
30: return specsAndProps

B.3 Semantics of Statements

The semantics of class and object definitions have been discussed above, while rules for the other statements are given in Fig. 24. As can be seen from the first rule, variable assignment behaves in the standard way. Parameter assignment is nearly identical, simply updating the global parameter assignment $\pi$ instead of the variable assignment $\sigma$. 

As noted above the semantics of the individual specifiers are mostly straightforward, and exact definitions are given in Appendix B. To illustrate the pattern we precisely define two specifiers in Fig. 23: the with property value specifier, which has no dependencies but can specify any property, and the facing vectorField specifier, which depends on position and specifies heading. Both specifiers evaluate to maps assigning a value to each property they specify.
As we have just defined it, every time one runs a Scenic program its output is a scene consisting of an assignment to all the properties of each Object defined in the scenario, plus any global parameters defined with param. Since Scenic allows sampling from distributions, the imperative part of a scenario actually induces a distribution over scenes, resulting from the
Algorithm 2 pruneByHeading (map, A, maxDist, δ)

1: map' ← ∅
2: for all polygon P in map do
3:    for all polygon Q in map do
4:       Q' ← dilate(Q, maxDist)
5:       if P ∩ Q' ≠ ∅ ∧ relHead(P, Q) ± 2δ ∈ A then
6:          map' ← map' ∪ (Q' ∩ P)
7: return map'

Algorithm 3 pruneByWidth (map, maxDist, minWidth)

1: narP ← narrow(map, minWidth)
2: map' ← map \ narP
3: for all polygon P in narP do
4:    U ← ∪Q∈map\{P\} dilate(Q, maxDist)
5:    map' ← map' ∪ (P ∩ U)
6: return map'

probabilistic rules of the semantics described above. Specifically, for any execution trace the product of the probabilities of all rewrite rules yields a probability (density) for the trace (see e.g. [4]). The declarative part of a scenario, consisting of its require statements, modifies this distribution. As mentioned above, hard requirements are equivalent to “observations” in other probabilistic programming languages, conditioning the distribution on the requirement being satisfied. In particular, if we discard all traces which do not terminate (due to violating a requirement), then normalizing the probabilities of the remaining traces yields a distribution over traces, and therefore scenes, that satisfy all our requirements. This is the distribution defined by the Scenic scenario.

B.5 Sampling Algorithms

C Detailed Semantics of Specifiers and Operators

This section provides precise semantics for Scenic’s specifiers and operators, which were informally defined above.

Contents

- C.1 Notation 32
- C.2 Specifiers for position 34
- C.3 Specifiers for position and optionally heading 35
- C.4 Specifiers for heading 35
- C.5 Operators 36

C.1 Notation

Since none of the specifiers and operators have side effects, to simplify notation we write ⌊X⌋ for the value of the expression X in the current state (rather than giving inference rules). Throughout this section, S indicates a scalar, V a vector, H a heading, F a vectorField, R a region, P a Point, and OP an OrientedPoint. Figure 25 defines notation used in the rest of the semantics. In forwardEuler, N is an implementation-defined parameter specifying how many steps should be used for the forward Euler approximation when following a vector field (we used N = 4).
\langle x, y \rangle = \text{point with the given XY coordinates}

\text{rotate}(\langle x, y \rangle, \theta) = \langle x \cos \theta - y \sin \theta, x \sin \theta + y \cos \theta \rangle

\text{offsetLocal}(\text{OP}, v) = [\text{OP}.\text{position}] + \text{rotate}(v, [\text{OP}.\text{heading}])

\text{Disc}(c, r) = \text{set of points in the disc centered at } c \text{ and with radius } r

\text{Sector}(c, r, h, a) = \text{set of points in the sector of } \text{Disc}(c, r) \text{ centered along } h \text{ and with angle } a

\text{boundingBox}(O) = \text{set of points in the bounding box of object } O

\text{visibleRegion}(X) = \begin{cases} 
\text{Sector}([X.\text{position}], [X.\text{viewDistance}], [X.\text{heading}], [X.\text{viewAngle}]) & X \in \text{OrientedPoint} \\
\text{Disc}([X.\text{position}], [X.\text{viewDistance}]) & X \in \text{Point}
\end{cases}

\text{orientation}(R) = \text{preferred orientation of } R \text{ if any; otherwise } \bot

\text{uniformPointIn}(R) = \text{a uniformly random point in } R

\text{forwardEuler}(x, d, F) = \text{result of iterating the map } x \mapsto x + \text{rotate}(\langle 0, d/N \rangle, \|F\|(x)) \text{ a total of } N \text{ times on } x

\textbf{Figure 25.} Notation used to define the semantics.
C.2 Specifiers for position

Figure 26 gives the semantics of the position specifiers. The figure writes the semantics as a vector value; the semantics of the specifier itself is to assign the position property of the object being specified to that value. Several of the specifiers refer to properties of self: as explained in Sec. 4, this refers to the object being constructed, and the semantics of object construction are such that specifiers depending on other properties are only evaluated after those properties have been specified (or an error is raised, if there are cyclic dependencies).

\[
\begin{align*}
\text{[at } V]\text{]} &= \{V\} \\
\text{[offset by } V]\text{]} &= \{V\text{ relative to ego.position}\} \\
\text{[offset along } H\text{ by } V]\text{]} &= \{\text{ego.position offset along } H\text{ by } V\} \\
\text{[left of } V]\text{]} &= \{\text{left of } V\text{ by 0}\} \\
\text{[right of } V]\text{]} &= \{\text{right of } V\text{ by 0}\} \\
\text{[ahead of } V]\text{]} &= \{\text{ahead of } V\text{ by 0}\} \\
\text{[behind } V]\text{]} &= \{\text{behind } V\text{ by 0}\} \\
\normalsize\text{[left of } V\text{ by } S]\text{]} &= \{V\} + \text{rotate}(\small{-[\text{self.width}] / 2 - [S], 0}, [\text{self.heading}]) \\
\normalsize\text{[right of } V\text{ by } S]\text{]} &= \{V\} + \text{rotate}([\text{self.width}] / 2 + [S], 0), [\text{self.heading}]) \\
\normalsize\text{[ahead of } V\text{ by } S]\text{]} &= \{V\} + \text{rotate}(0, [\text{self.height}] / 2 + [S]), [\text{self.heading}]) \\
\normalsize\text{[behind } V\text{ by } S]\text{]} &= \{V\} + \text{rotate}(0, -[\text{self.height}] / 2 - [S]), [\text{self.heading}]) \\
\text{[beyond } V_1\text{ by } V_2\text{ from } V_3]\text{]} &= \{V_1\} + \text{rotate}([V_2], \text{arctan}([V_1] - [V_3])) \\
\text{[visible]} &= \{\text{visible from ego}\} \\
\text{[visible from } P]\text{]} &= \text{uniformPointIn}(\text{visibleRegion}(P))
\end{align*}
\]

**Figure 26.** Semantics of position specifiers, given as the value \( v \) such that the specifier evaluates to the map \( \text{position} \mapsto v \).
C.3 Specifiers for position and optionally heading

Figure 27 gives the semantics of the position specifiers that also optionally specify heading. The figure writes the semantics as an OrientedPoint value; if this is $\text{OP}$, the semantics of the specifier is to assign the position property of the object being constructed to $\text{OP}.\text{position}$, and the heading property of the object to $\text{OP}.\text{heading}$ if heading is not otherwise specified (see Sec. 4 for a discussion of optional specifiers).

\[
\text{in } R = \text{[on } R\text{]} = \begin{cases} 
\text{OrientedPoint}(x, \text{[orientation}(R)](x)) & \text{orientation}(R) \neq \perp \\
\text{OrientedPoint}(x, \perp) & \text{otherwise}
\end{cases}, \text{ with } x = \text{uniformPointIn}(\text{[}] R\text{])}
\]

- [ahead of $O$] = [ahead of (front of $O$)]
- [behind $O$] = [behind (back of $O$)]
- [left of $O$] = [left of (left of $O$)]
- [right of $O$] = [right of (right of $O$)]
- [ahead of $\text{OP}$] = [ahead of $\text{OP}$ by $\emptyset$]
- [behind $\text{OP}$] = [behind $\text{OP}$ by $\emptyset$]
- [left of $\text{OP}$] = [left of $\text{OP}$ by $\emptyset$]
- [right of $\text{OP}$] = [right of $\text{OP}$ by $\emptyset$]

- [ahead of $\text{OP}$ by $S$] = OrientedPoint($\text{offsetLocal}(\text{OP}, \langle 0, \text{[self.height]}/2 + \text{[S]} \rangle)$, $\text{OP}.\text{heading}$)
- [behind $\text{OP}$ by $S$] = OrientedPoint($\text{offsetLocal}(\text{OP}, \langle 0, -\text{[self.height]}/2 - \text{[S]} \rangle)$, $\text{OP}.\text{heading}$)
- [left of $\text{OP}$ by $S$] = OrientedPoint($\text{offsetLocal}(\text{OP}, \langle -\text{[self.width]}/2 - \text{[S]}, 0 \rangle)$, $\text{OP}.\text{heading}$)
- [right of $\text{OP}$ by $S$] = OrientedPoint($\text{offsetLocal}(\text{OP}, \langle \text{[self.width]}/2 + \text{[S]}, 0 \rangle)$, $\text{OP}.\text{heading}$)

- [following $F$ for $S$] = [following $F$ from ego.position for $S$]
- [following $F$ from $V$ for $S$] = [follow $F$ from $V$ for $S$]

Figure 27. Semantics of position specifiers that optionally specify heading. If $o$ is the OrientedPoint given as the semantics above, the specifier evaluates to the map \{position \mapsto o.\text{position}, heading \mapsto o.\text{heading}\}.

C.4 Specifiers for heading

Figure 28 gives the semantics of the heading specifiers. As for the position specifiers above, the figure indicates the heading value assigned by each specifier.

\[
\text{[facing } H\text{]} = \langle H \rangle
\]

- [facing $F$] = $\langle F \rangle(\text{[self.\text{position}]})$
- [facing toward $V$] = $\text{arctan}((V) - \text{[self.\text{position}])}$
- [facing away from $V$] = $\text{arctan}((\text{self.\text{position}}) - (V))$
- [apparently facing $H$] = [apparently facing $H$ from ego.position]
- [apparently facing $H$ from $V$] = $\langle H \rangle + \text{arctan}(\text{[self.\text{position}] - (V)}$

Figure 28. Semantics of heading specifiers, given as the value $v$ such that the specifier evaluates to the map heading \mapsto v.
C.5 Operators

Finally, Fig. 29 shows the syntax of all of Scenic’s operators, and Figures 30–35 give their semantics. The Figures are broken down by the type of value the operator returns. We omit the semantics for ordinary numerical and Boolean operators (max, +, or, >=, etc.), which are standard.

![Figure 29. Operators by result type.](image)

$$\text{C.5 Operators}$$

Finally, Fig. 29 shows the syntax of all of Scenic’s operators, and Figures 30–35 give their semantics. The Figures are broken down by the type of value the operator returns. We omit the semantics for ordinary numerical and Boolean operators (max, +, or, >=, etc.), which are standard.

![Figure 30. Scalar operators.](image)

$$\text{Figure 30. Scalar operators.}$$

![Figure 31. Boolean operators.](image)

$$\text{Figure 31. Boolean operators.}$$
\[ [F \text{ at } V] = [F][[V]] \]
\[ [F_1 \text{ relative to } F_2] = [F_1][[\text{self.position}]] + [F_2][[\text{self.position}]] \]
\[ [H \text{ relative to } F] = [H] + [F][[\text{self.position}]] \]
\[ [F \text{ relative to } H] = [H] + [F][[\text{self.position}]] \]
\[ [H_1 \text{ relative to } H_2] = [H_1] + [H_2] \]

**Figure 32.** Heading operators.

\[ [V_1 \text{ offset by } V_2] = [V_1] + [V_2] \]
\[ [V_1 \text{ offset along } H \text{ by } V_2] = [V_1] + \text{rotate}([[V_2],[H]]) \]
\[ [V_1 \text{ offset along } F \text{ by } V_2] = [V_1] + \text{rotate}([[V_2],[F][[V_1]]) \]

**Figure 33.** Vector operators.

\[ [\text{visible } R] = [R \text{ visible from ego}] \]
\[ [R \text{ visible from } P] = [R] \cap \text{visibleRegion}([P]) \]

**Figure 34.** Region operators.

\[ [OP \text{ offset by } V] = [V \text{ relative to } OP] \]
\[ [V \text{ relative to } OP] = \text{OrientedPoint}(\text{offsetLocal}(OP,[V]),[OP.\text{heading}]) \]
\[ [\text{follow } F \text{ for } S] = [\text{follow } F \text{ from ego.position for } S] \]
\[ [\text{follow } F \text{ from } V \text{ for } S] = \text{OrientedPoint}(y,[F](y)) \text{ where } y = \text{forwardEuler}([V],[S],[F]) \]
\[ [\text{front of } O] = [[0,0.\text{height}]/2] \text{ relative to } O \]
\[ [\text{back of } O] = [[0,-0.\text{height}]/2] \text{ relative to } O \]
\[ [\text{left of } O] = [[-0.\text{width}]/2,0] \text{ relative to } O \]
\[ [\text{right of } O] = [[0.\text{width}]/2,0] \text{ relative to } O \]
\[ [\text{front left of } O] = [[-0.\text{width}]/2,0.\text{height}]/2] \text{ relative to } O \]
\[ [\text{back left of } O] = [[-0.\text{width}]/2,-0.\text{height}]/2] \text{ relative to } O \]
\[ [\text{front right of } O] = [[0.\text{width}]/2,0.\text{height}]/2] \text{ relative to } O \]
\[ [\text{back right of } O] = [[0.\text{width}]/2,-0.\text{height}]/2] \text{ relative to } O \]

**Figure 35.** OrientedPoint operators.
D Additional Experiments

This section gives additional details on the experiments and describes an experiment analogous to that of Sec. 6.3 but using the generic two-car Scenic scenario as a baseline.

Additional Details

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:

- `model`, representing the type of car. We used a set of 13 diverse models supported by GTAV, with the default distribution over these being uniform.
- `color`, representing the car color. The default distribution was based on real-world car color statistics [5].

In addition, we implemented two global scene parameters:

- `time`, representing the time of day. The default distribution was uniform over all 24 hours.
- `weather`, representing the weather as one of 14 discrete types supported by GTAV (e.g. "clear" or "snow"). The default distribution gave all types positive probability, biased towards less extreme weather.

Unfortunately, GTAV does not provide an explicit representation of its map. We obtained an approximate map by processing a bird’s-eye schematic view of the game world. To identify points on a road, we converted the image to black and white, effectively turning roads white and everything else black. We then used edge detection to find curbs, and computed the nominal traffic direction by finding for each curb point `X` the nearest curb point `Y` on the other side of the road, and assuming traffic flows perpendicular to the segment `XY` (this was more robust than using the directions of the edges in the image). Since the resulting road information was imperfect, some generated scenes placed cars in undesired places such as sidewalks or medians, and we had to manually filter the generated images to remove these. With a real simulator, e.g. Webots, this is not necessary.

Our implementation’s interface to GTAV is based on DeepGTAV. To render a scene, we use a series of API calls to create the cars and set the time of day and weather.

We now define in detail the metrics used to measure the performance of our models. Let `ŷ = f(x)` be the prediction of the model `f` for input `x`. For our task, `ŷ` encodes bounding boxes, scores, and categories predicted by `f` for the image `x`. Let `B_ŷ` be a ground truth box (i.e. a bounding box from the label of a training sample that indicates the position of a particular object) and `B` be a box predicted by the model. The Intersection over Union (IoU) is defined as `IoU(B_ŷ, B) = area(B_ŷ ∩ B) / area(B_ŷ ∪ B)`, where `area(X)` is the area of a set `X`. IoU is a common evaluation metric used to measure how well predicted bounding boxes match ground truth boxes. We adopt the common practice of considering `B` a detection for `B_ŷ` if `IoU(B_ŷ, B) > 0.5`. Precision and recall are metrics used to measure the accuracy of a prediction on a particular image. Intuitively, precision is the fraction of predicted boxes that are correct, while recall is the fraction of objects actually detected. Formally, 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. We use average precision and recall to evaluate the performance of a model on a collection of images constituting a test set.

Overlapping Scenario Experiment

Specifically, we generated 1,000 images from that scenario, obtaining a training set `X_{twocar}`. We also generated 1,000 images from the overlapping scenario to get a training set `X_{overlap}`.

Note that `X_{twocar}` did contain images of overlapping cars, since the generic two-car scenario does not constrain the cars’ locations. However, the average overlap was much lower than that of `X_{overlap}`, as seen in Fig. 36 (note the log scale): thus the overlapping car images are highly “untypical” of generic two-car images. We would like to ensure the network performs well on these difficult images by emphasizing them in the training set. Therefore we constructed various mixtures of the two training sets, fixing the total number of images but using different ratios of images from `X_{twocar}` and `X_{overlap}`. We trained the network on each of these mixtures and evaluated their performance on 400-image test sets `T_{twocar}` and `T_{overlap}` from the two-car and overlapping scenarios respectively.

To reduce the effect of randomness in training, we used the maximum precision and recall obtained when training for 4,000 through 5,000 steps in increments of 250 steps. Additionally, we repeated each training 8 times, using a random mixture each time: for example, for the 90/10 mixture of `X_{twocar}` and `X_{overlap}`, each training used an independent random choice of which 90% of `X_{twocar}` to use and which 10% of `X_{overlap}`.

As the results in Tab. 9 show, the model trained purely on generic two-car images has high precision and recall on `T_{twocar}`, but has drastically worse recall on `T_{overlap}`: essentially, the network has difficulty detecting the partially-occluded car. However, devoting 20% of the training set to overlapping cars gives a large 8% improvement to recall on `T_{overlap}` while leaving performance on `T_{twocar}` essentially the same. This again demonstrates that we can improve the performance of a network on difficult corner cases by using Scenic to increase the representation of such cases in the training set.
Figure 36. Intersection Over Union (IOU) distribution for two-car and overlapping training sets (log scale).

Table 9. Performance of models trained on mixtures of $X_{twocar}$ and $X_{overlap}$ and tested on both, averaged over 8 training runs. 90/10 indicates a 9:1 mixture of $T_{twocar}/T_{overlap}$.

<table>
<thead>
<tr>
<th>Mixture</th>
<th>$T_{twocar}$ Precision</th>
<th>$T_{twocar}$ Recall</th>
<th>$T_{overlap}$ Precision</th>
<th>$T_{overlap}$ Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>100/0</td>
<td>96.5 ± 1.0</td>
<td>95.7 ± 0.5</td>
<td>94.6 ± 1.1</td>
<td>82.1 ± 1.4</td>
</tr>
<tr>
<td>90/10</td>
<td>95.3 ± 2.1</td>
<td>96.2 ± 0.5</td>
<td>93.9 ± 2.5</td>
<td>86.9 ± 1.7</td>
</tr>
<tr>
<td>80/20</td>
<td>96.5 ± 0.7</td>
<td>96.0 ± 0.6</td>
<td>96.2 ± 0.5</td>
<td>89.7 ± 1.4</td>
</tr>
<tr>
<td>70/30</td>
<td>96.5 ± 0.9</td>
<td>96.5 ± 0.6</td>
<td>96.0 ± 1.6</td>
<td>90.1 ± 1.8</td>
</tr>
</tbody>
</table>