Machine Learning (2023) 112:3805–3849
https://doi.org/10.1007/s10994-021-06120-5

Scenic: a language for scenario specification and data generation

Daniel J. Fremont[1] · Edward Kim[2] · Tommaso Dreossi[3] · Shromona Ghosh[4] · Xiangyu Yue[2] · Alberto L. Sangiovanni-Vincentelli[2] · Sanjit A. Seshia[2]

Received: 22 September 2020 / Revised: 13 October 2021 / Accepted: 27 October 2021 /
Published online: 2 February 2022

Abstract

We propose a new probabilistic programming language for the design and analysis of cyber-physical systems, especially those based on machine learning. We consider several problems arising in the design process, including training a system to be robust to 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, then sampling these to generate specialized training and test data. More generally, such languages can be used to write environment models, an essential prerequisite to any formal analysis. In this paper, we focus on systems such as autonomous cars and robots, whose environment at any point in time is a *scene*, a configuration of physical objects and agents. We design a domain-specific language, Scenic, for describing *scenarios* that are distributions over scenes and the behaviors of their agents over time. Scenic combines concise, readable syntax for spatiotemporal relationships with the ability to declaratively impose hard and soft constraints over the scenario. 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 multiple case studies for training, testing, and debugging neural networks for perception both as standalone components and within the context of a full cyber-physical system.

Editors: Mykel Kochenderfer, Alessio Lomuscio, Dragos Margineantu, Cheng Soon-Ong.

Preliminary versions of this article appeared in the *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)* (Fremont et al. 2019) and as an April 2018 UC Berkeley technical report (Fremont et al. 2018). D. J. Fremont began this work while affiliated with UC Berkeley; T. Dreossi and S. Ghosh's contributions were also made there.

✉ Daniel J. Fremont
dfremont@ucsc.edu

1 University of California, Santa Cruz, CA, USA

2 University of California, Berkeley, CA, USA

3 insitro, San Francisco, CA, USA

4 Waymo LLC, Mountain View, CA, USA

Keywords Scenario description language · Synthetic data · Deep learning · Probabilistic programming · Debugging · Automatic test generation · Simulation

CR Subject Classification: D.2.5 · D.3.2 · I.2.6 · I.2.9

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 (Russell et al. 2015; Seshia et al. 2016; Amodei et al. 2016). ML has proved particularly effective at the difficult perceptual tasks (e.g., vision) arising in *cyber-physical systems* like autonomous vehicles which operate in heterogeneous, complex physical environments. Thus, there is a pressing need to tackle several important problems in the design of such ML-based cyber-physical systems, including:

- *training* the system to be robust, correctly responding 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 and even dangerous 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 (Jaderberg et al. 2014; Gupta et al. 2016; Tobin et al. 2017; Johnson-Roberson et al. 2017).

A challenge to the use of synthetic data is that it can be highly non-trivial to generate *meaningful* data, since this usually requires modeling complex environments (Seshia et al. 2016). Suppose we wanted to train a neural 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 scenarios that make sense for our application.

We argue that probabilistic programming languages (PPLs) (Gordon et al. 2014) 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 about which we could potentially prove properties to establish the correctness of the system.

In this paper, we focus on designing and analyzing ML-based cyber-physical systems. We refer to the environment of such a system at any point in time as a *scene*, a configuration of objects in space (including dynamic agents, such as vehicles) along with their features. 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 both scenes and the behaviors of the dynamic agents in them over time. 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 *spatial* and *temporal* relationships: constructs for common geometric relationships that would otherwise require complex non-linear expressions and constraints, as well as temporal constructs such as parallel and sequential composition and interrupts for building complex dynamic behaviors in a modular way. 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 constructs to *generalize* simple scenarios by adding noise or by *composing* multiple scenarios together.

The variety of constructs in SCENIC makes it possible to model scenarios anywhere on a spectrum from concrete scenes (i.e. individual test cases) to extremely broad classes of abstract scenarios (see Fig. 2). A scenario can be reached by moving along the spectrum from either end: the top-down approach is to progressively constrain a very general scenario, while the bottom-up approach is to generalize from a concrete example (such as a known failure case), for example by adding random noise. Probably most usefully, one can write a scenario in the middle which is far more general than simply adding noise to a single scene but has much more structure than a completely random scene: for example, the traffic

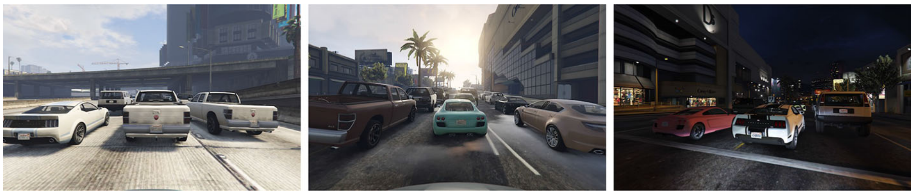
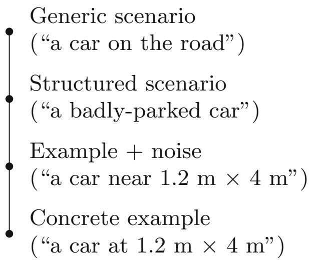


Fig. 1 Three scenes generated from a single ∼ 20-line SCENIC program representing bumper-to-bumper traffic

Fig. 2 Spectrum of scenarios, from general to specific



scenario depicted in Fig. 1. We will illustrate all three ways of developing a scenario, which as we will see are useful for different training, testing, and debugging tasks.

Generating concrete scenarios from a SCENIC program requires sampling from the probability distribution it implicitly defines. This task is closely related to the inference problem for imperative PPLs with observations (Gordon et al. 2014). While SCENIC could be implemented as a library on top of such a language, we found that clarity and concision could be significantly improved with new syntax (specifiers and interrupts in particular) difficult to implement as a library. Furthermore, while SCENIC could be translated into existing PPLs, using a new language allows us to impose restrictions that enable domain-specific sampling techniques which are not applicable to general-purpose PPLs. In particular, we develop algorithms which take advantage of the particular structure of distributions arising from SCENIC programs to dramatically prune the sample space. We refer to the random generation of concrete scenarios as *scenario improvisation*, as it is inspired by and closely related to a class of problems known as *control improvisation* (Fremont et al. 2015; Fremont 2019).

We also integrate SCENIC as the environment modeling language for VERIFAI, a tool for the formal design and analysis of AI-based systems (Dreossi et al. 2019). VERIFAI allows writing system-level specifications in Metric Temporal Logic (Koymans 1990) or as objective functions, and performing falsification, running simulations and monitoring for violations of the specifications. VERIFAI provides several search techniques, including active samplers that use feedback from earlier simulations to try to drive the system towards violations. To support these active samplers, each sampled concrete scenario and the corresponding performance of the system with respect to its given specifications are logged in a table. This data can be analyzed (by clustering, principal component analysis, etc.) to determine promising parts of the environment space; an active sampler can intelligently select an unexplored concrete scenario that is likely to induce a violation of a specification. We make these techniques available from SCENIC using syntax to define *external parameters* which are sampled by VERIFAI or another external tool. Such parameters need not have a fixed distribution of values: for instance, we can define a *prior* distribution, but then use cross-entropy optimization (Rubinstein and Kroese 2004) to drive the distribution towards one that is concentrated on values that tend to lead to system failures (Fremont et al. 2020).

We demonstrate the utility of SCENIC in training, testing, and debugging ML-based cyber-physical systems, both at the ML component level and at the full system level. Our first case study is on SqueezeDet (Wu et al. 2017), a convolutional neural network for object detection in autonomous cars. For this task, it has been shown (Johnson-Roberson et al. 2017) that good performance on real images can be achieved with networks trained purely on synthetic images from the video game Grand Theft Auto V [GTAV (Rockstar Games 2015)]. We implemented a sampler for SCENIC scenarios, using it to generate scenes which were rendered into images by GTAV. Our experiments demonstrate using SCENIC to:

- evaluate the accuracy of the ML model 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 (Johnson-Roberson et al. 2017) 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

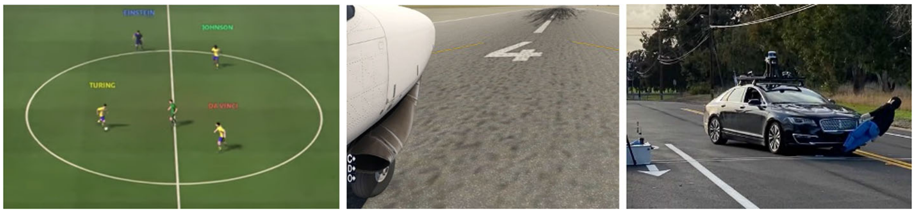


Fig. 3 Various domains where we have applied SCENIC: reinforcement learning agents for soccer (Azad et al. 2021), ML-based aircraft navigation (Fremont et al. 2020), and autonomous vehicle testing in the real world (Fremont et al. 2020b)

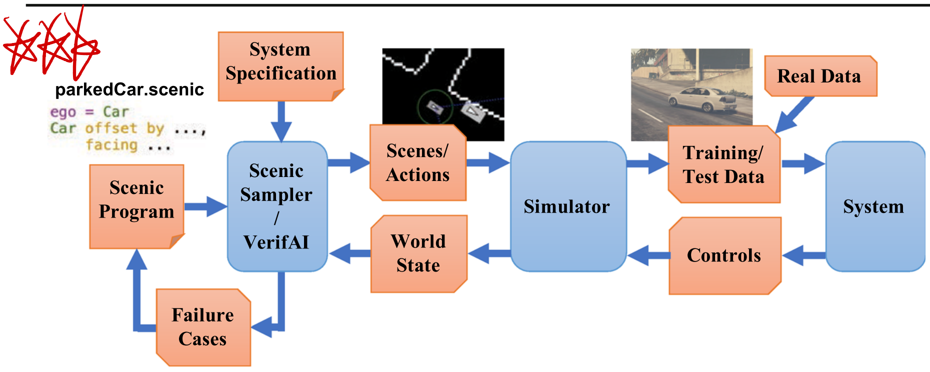
the process improving the network's performance on its original test set (again, without increasing training set size).

These experiments show that SCENIC can be a very useful tool for understanding and improving ML-based perception systems.

While this case study is performed in the domain of visual perception for autonomous driving, and uses one particular simulator (GTAV), we stress that SCENIC is not specific to either. Several other applications where we have successfully used SCENIC are shown in Fig. 3; see the cited papers for details. In this paper, we include two additional examples: in Sec. 3 we illustrate a different domain, namely robotic motion planning [using the Webots simulator (Michel 2004)], and in Sect. 7.2.2 we use SCENIC and VERIFAI to falsify an autonomous agent in the CARLA driving simulator (Dosovitskiy et al. 2017). The latter experiment demonstrates SCENIC's usefulness applied not only to ML-based perception components in isolation but to entire closed-loop cyber-physical systems. In fact, since the conference version of this paper we have successfully applied SCENIC in two industrial case studies on large ML-based systems (Fremont et al. 2020, b): an aircraft navigation system from Boeing [tested in the X-Plane flight simulator (Laminar Research 2019)] and the Apollo autonomous driving platform (Baidu 2020) [tested in the LGSVL driving simulator (Rong et al. 2020) and on an actual test track]. Generally, SCENIC *can produce data of any desired type* (e.g. RGB images, LIDAR point clouds, or trajectories from dynamical simulations) by interfacing it to an appropriate simulator. This requires only two steps: (1) writing a small SCENIC library defining the types of objects supported by the simulator, as well as the geometry of the workspace; (2) writing an interface layer converting the configurations output by SCENIC into the simulator's input format (and, for dynamic scenarios, transferring simulator state back into SCENIC). While the current version of SCENIC is primarily concerned with geometry, leaving the details of rendering up to the simulator, the language allows putting distributions on any parameters the simulator exposes: for example, in GTAV the meshes of the various car models are fixed but we can control their overall color. We have also used SCENIC to specify distributions over parameters on system dynamics, such as mass.

In summary, the main contributions of this work are:

- SCENIC, a domain-specific probabilistic programming language for describing *scenarios*: distributions over spatio-temporal configurations of physical objects and agents;
- a methodology for using PPLs to design and analyze cyber-physical systems, especially those based on ML;
- domain-specific algorithms for sampling from the distribution defined by a SCENIC program;



```
ego = Car
Car offset by ...,
    facing ...
```

Fig. 4 Tool flow using Scenic to train, test, and debug a cyber-physical system

– a case study using Scenic to analyze and improve the accuracy of a practical deep neural network used for perception in an autonomous driving context 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 Sect. 2. Section 3 gives examples highlighting the major features of Scenic for specifying spatial relationships and motivating various choices in its design. We continue in Sect. 4 with a discussion of Scenic's more advanced features for temporal modeling and scenario composition. In Sect. 5 we describe the syntax of the Scenic language in detail, and in Sect. 6 we discuss its formal semantics and our sampling algorithms. Section 7 describes the setup and results of our car detection case study and other experiments. Finally, we discuss related work in Sect. 8 before concluding in Sect. 9 with a summary and directions for future work.

An early version of this paper appeared as Fremont et al. (2018), extended and published as Fremont et al. (2019). This paper further extends Fremont et al. (2019) by generalizing Scenic to dynamic scenarios (including new spatiotemporal pruning techniques), adding constructs for composing scenarios, and integrating Scenic within the broader VerifAI toolkit. For the Appendices and our implementation code, see Fremont et al. (2020a).

2 Using PPLs to design and analyze ML-based cyber-physical systems

We propose a methodology for training, testing, and debugging ML-based cyber-physical 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 system. The general procedure is outlined in Fig. 4. For a demonstration of this paradigm on an industrial system, proceeding from falsification through failure analysis, retraining, and validation, see Fremont et al. (2020). Note that the training/testing datasets need not be purely synthetic: we can generate data to supplement existing real-world data (possibly mitigating a deficiency in the latter, while avoiding overfitting). Furthermore, even for models trained purely on real data, synthetic data can still be useful for testing and debugging, as we will see below. Now we discuss the three design problems from the Introduction in more detail.

Testing and falsification. 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. If we do not have particular conditions in mind, we can write a very general scenario describing the expected operation regime of the system [e.g., the "Operational Design Domain" (ODD) of an autonomous vehicle (Thorn et al. 2018)] and perform falsification, looking for violations of the system's specification. We can perform such analyses at the level of individual components or of the system as a whole: in Sect. 7.2.1 we test a car-detecting neural network's sensitivity to weather, while in Sect. 7.2.2, we use the VERIFAI toolkit (Dreossi et al. 2019) to falsify a closed-loop AV system, modeling a traffic scenario in SCENIC and specifying a safety specification for the AV in temporal logic.

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 Sect. 7.3 we will demonstrate this for car detection, where a hard case is when one car partially overlaps another in the image. We wrote a SCENIC program to generate a set of these overlapping images. Training the car-detection network on a state-of-the-art synthetic dataset obtained by randomly driving around inside the simulated world of GTAV and capturing images periodically (Johnson-Roberson et al. 2017), we find its performance is significantly worse on the overlapping images. However, if we keep the training set size fixed but increase the proportion of overlapping images, performance on such images dramatically improves *without harming performance on the original generic dataset*.

Debugging failures. Finally, we can use the same procedure to help understand and fix bugs in the system. If we find an environment configuration where the system fails, we can write a scenario reproducing that particular configuration. Having the configuration encoded as a program then makes it possible to explore the neighborhood around it in a variety of different directions, leaving some aspects of the scene fixed while varying others. This can give insight into which features of the scene are relevant to the failure, and eventually identify the root cause. The root cause can then itself be encoded into a scenario which generalizes the original failure, allowing retraining without overfitting to the particular counterexample. We will demonstrate this approach in Sect. 7.4, starting from a single misclassification, identifying a general deficiency in the training set, replacing part of the training data to fix the gap, and ultimately achieving higher performance on the original test set.
For all of these applications we need a PPL which can encode a wide range of general and specific environment scenarios. In the next section, we describe the design of a language suited to this purpose.

3 The basic 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 specifying scenarios for cyber-physical systems. We begin by describing how SCENIC can define *spatial* relationships between objects to model scenarios like "a badly-parked car”; in Sect. 4, we will cover Scenic’s more advanced constructs for temporal dynamics and scenario composition.

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

```
model scenic.simulators.gta.model
ego = Car
Car
```

First, we import Scenic’s *world model* for the GTAV simulator: a Scenic library containing everything specific to our case study, including 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 the GTAV world model and use it in different scenarios. Our definition of `Car` begins as follows (slightly simplified):

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

Here, `position` and `heading` are properties of a `Car` object. These properties may have distributions and constraints, both of which model realistic initial state of the object. `road` is a *region* (one of Scenic’s primitive types) defined in the GTAV world model to specify which points in the workspace are on a road. Similarly, `roadDirection` is a *vector field* specifying the prevailing traffic direction at such points. The operator F `at` X simply gets the direction of the field F at point X, so the default value for a car’s `heading` is the road direction at its `position`. The default `position`, in turn, is a `Point on road` (we will explain this syntax shortly), which means a *uniformly random* point on the road.

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

```
Car offset by (Range(-10, 10), Range(20, 40))
```

to create a car that is 20–40 m ahead of the camera. The notation `Range`(X, Y) creates a uniform distribution over the given continuous range, and (X,Y)creates a pair, interpreted here as a vector given by its xy coordinates.

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

```
Car offset by (Range(-10, 10), Range(20, 40)),
    facing Range(-5, 5) deg
```

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

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

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

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

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

1. "is at position *X*" (absolute position);
2. "is just left of position *X*" (position 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 coordinate system);
5. "appears to be 10 m behind the taxi" (relative to the line of sight);
6. "is 10 m along the road from the taxi" (following a curved vector field).

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:

```
# hypothetical Python-like language
model = Car.defaultModelDistribution.sample()
pos = curb.offsetLeft(0.5 + model.width / 2)
car = Car(pos, model=model)
```

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

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

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

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

Here left of *X* by *D* and with model*M* are *specifiers*, which are unordered and *together* specify the properties of the car. SCENIC works out the dependencies between properties (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 omit with model BUS; keeping it affects the position appropriately without having to specify BUS more than once.

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

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

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

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

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

This will yield cars parked 10°–20° off from the direction of the curb, as seen in Fig. 5. This illustrates how specifiers greatly enhance SCENIC's flexibility and modularity.

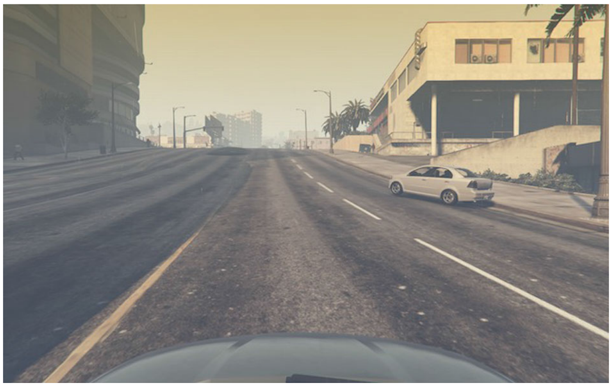


Fig. 5 A scene of a badly-parked car

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

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

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

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

Hard requirements, called "observations" in other PPLs (see, e.g., Gordon et al. (2014)), are very convenient in our setting because they make it easy to restrict attention to particular cases of interest. They also improve encapsulation, since we can restrict an existing scenario without altering it (we can simply import it in a new SCENIC program that includes additional `require` statements). 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:

[1] The last requirement ensures that the object will affect the rendered image. It can be disabled on a per-object basis, for example in dynamic scenarios where the object is initially out of sight but may interact with the ego object later on.

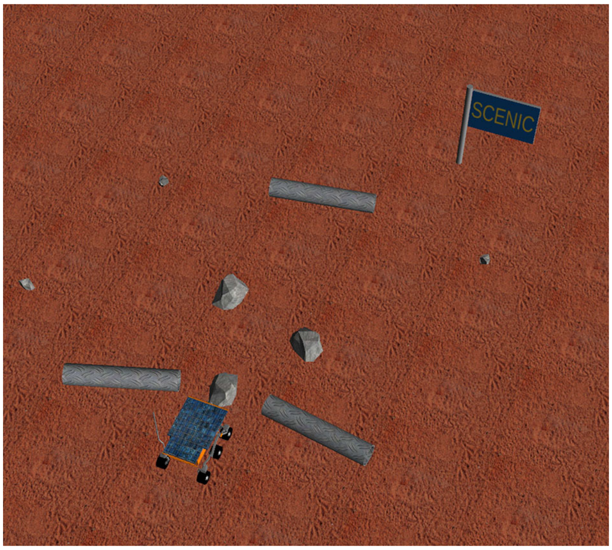


Fig. 6 Webots scene of a Mars rover in a debris field with a bottleneck

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

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

Multiple Domains and Simulators. We conclude this section by illustrating a second application domain, namely testing motion planning algorithms, and also 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 6 shows a scene, visualized using an interface we wrote between Scenic and the Webots robotics simulator (Michel 2004), with a bottleneck between the robot and its goal that forces the planner to consider climbing over a rock.

Even within a single application domain, such as autonomous driving, Scenic enables writing *cross-platform* scenarios that will work without change in multiple simulators. This is made possible by what we call *abstract application domains*: Scenic world models which define object classes and other world information like our GTAV world model, but which are abstract, simulator-agnostic protocols that can be implemented by models for particular simulators. For example, Scenic includes an abstract domain for autonomous driving, `scenic.domains.driving`, which loads road networks from standard formats, providing a uniform API for referring to lanes, maneuvers, and other aspects of road geometry. The driving domain also provides generic `Car` and `Pedestrian` classes, complete with implementations of common dynamic behaviors (covered in the next section) like lane following. These make it straightforward to implement complex driving scenarios, which are then guar-

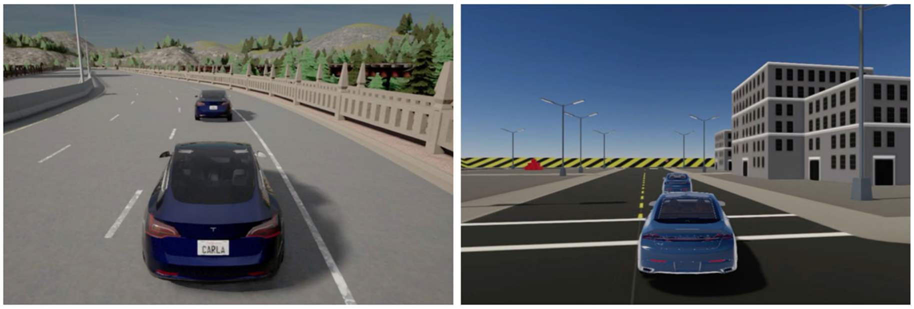


Fig. 7 Scenes sampled from the same SCENIC program in CARLA and LGSVL

anteed to work in any simulator supporting the driving domain. Figure 7 illustrates this, showing the exact same SCENIC code being used to generate scenarios in both the CARLA (Dosovitskiy et al. 2017) and LGSVL (Rong et al. 2020) simulators.
All of the examples we have seen above illustrate the versatility of SCENIC in modeling a wide range of interesting scenarios. Complete SCENIC code for the bumper-to-bumper scenario of Fig. 1, the Mars rover scenario of Fig. 6, 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 the Appendix (Fremont et al. 2020a).

4 Dynamic and compositional scenarios

In Sect. 3 we saw the basic constructs SCENIC provides for defining objects and their spatial relationships. These constructs suffice for expressing *static* scenarios like "a badly-parked car", where SCENIC need only define a configuration of objects at one point in time. However, for *dynamic* scenarios like "a badly-parked car, which pulls into the road as you approach", we need ways to express *temporal* properties of objects. In this section, we outline SCENIC's support for dynamic scenarios, as well as for *composing* multiple scenarios together to produce more complex ones.

4.1 Dynamic scenarios

Agents, Actions, and Behaviors. We call SCENIC objects which take actions over time *dynamic agents*, or simply *agents*. We can still use all of the syntax described above to define the initial positions, orientations, etc. of such objects. In addition, we specify their dynamic behavior using a built-in property called `behavior`. Using a behavior defined in SCENIC's driving library, we can write for example:

```
model scenic.domains.driving.model
Car with behavior FollowLaneBehavior
```

A behavior defines a sequence of *actions* for the agent to take, which need not be fixed but can be probabilistic and depend on the state of the agent or other objects. In SCENIC, an *action* is an instantaneous operation executed by an agent, like setting the steering angle of a car or turning on its headlights. Most actions are specific to particular application domains, and so different sets of actions are provided by different simulator interfaces. For example, the SCENIC driving domain defines a `SetThrottleAction` for cars.

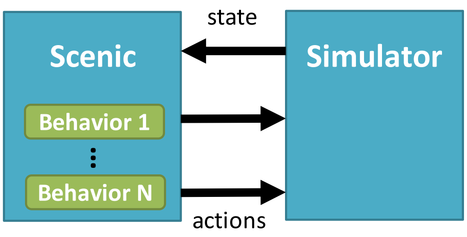


Fig. 8 Diagram showing interaction between SCENIC and a simulator during the execution of a dynamic scenario

To define a behavior, we write a function which runs over the course of the scenario, periodically issuing actions. SCENIC uses a discrete notion of time, so at each time step the function specifies zero or more actions for the agent to take. For example, here is a very simplified version of the `FollowLaneBehavior` above:

```
behavior FollowLaneBehavior():
    while True:
        throttle, steering = ...     # compute controls
        take SetThrottleAction(throttle), SetSteerAction(steering)
```

We intend this behavior to run for the entire scenario, so we use an infinite loop. In each step of the loop, we compute appropriate throttle and steering controls, then use the `take` statement to take the corresponding actions. When that statement is executed, SCENIC pauses the behavior until the next time step of the simulation, whereupon the function resumes and the loop repeats.

Execution of Behaviors. When there are multiple agents, their behaviors run in parallel, as seen in Fig. 8; each time step, SCENIC sends their selected actions to the simulator to be executed and runs the simulation for one step. It then reads back the state of the simulation, updating the `position`, `speed`, etc. of each object.

As behaviors run dynamically during simulations, they can access the current state of the world to decide what actions to take. Consider the following behavior:

```
behavior WaitUntilClose(threshold=15):
    while (distance from self to ego) > threshold:
        wait
    do FollowLaneBehavior()
```

Here, we repeatedly query the distance from the agent running the behavior (`self`) to the ego car; as long as it is above a threshold, we use the `wait` statement to take no action. Once the threshold is met, we start driving by using the `do` statement to invoke the `FollowLaneBehavior` we saw above.

Behavior Arguments and Random Parameters. The example above also shows how behaviors may take arguments, like any SCENIC function. Here, `threshold` has default value 15 but can be customized, so we could write for example:

```
ego = Car
carB = Car visible, with behavior WaitUntilClose
carC = Car visible, with behavior WaitUntilClose(20)
```

Both `carB` and `carC` will use the `WaitUntilClose` behavior, but independent copies of it with thresholds of 15 and 20 respectively.

Unlike ordinary SCENIC code, control flow constructs such as `if` and `while` are allowed to depend on random variables inside a behavior. Any distributions defined inside a behavior are sampled at simulation time, not during scene sampling. Consider the following behavior:

```
1 behavior AvoidPedestrian():
2     threshold = Range(4, 7)
3     while True:
4         if self.distanceToClosest(Pedestrian) < threshold:
5             strength = TruncatedNormal(0.8, 0.02, 0.5, 1)
6             take SetBrakeAction(strength), SetThrottleAction(0)
7         else:
8             take SetThrottleAction(0.5), SetBrakeAction(0)
```

Here, the value of `threshold` is sampled only once, at the beginning of the scenario when the behavior starts running. The value `strength`, on the other hand, is sampled every time control reaches line 5, so that every time step when the car is braking we use a slightly different braking strength.

Interrupts. It is frequently useful to take an existing behavior and add a complication to it; for example, suppose we want a car that follows a lane, stopping whenever it encounters an obstacle. Scenic provides a concept of *interrupts* which allows us to reuse the basic `FollowLaneBehavior` without having to modify it.

```
behavior FollowAvoidingObstacles():
    try:
        do FollowLaneBehavior()
    interrupt when self.distanceToClosest(Object) < 5:
        take SetBrakeAction(1)
```

This `try-interrupt` statement has the following semantics: at first, the code block after the `try` (the *body*) is executed. At the start of every time step during its execution, the condition from each `interrupt` clause is checked; if any are true, execution of the body is suspended and we instead begin to execute the corresponding *interrupt handler*. In the example above, there is only one interrupt, which fires when we come within 5 meters of any object. When that happens, `FollowLaneBehavior` is paused and we instead apply full braking for one time step. In the next step, we will resume `FollowLaneBehavior` wherever it left off, unless we are still within 5 meters of an object, in which case the interrupt will fire again.

Successive `interrupt` clauses take precedence over those which precede them, and such higher-priority interrupts can fire even during the execution of an earlier interrupt handler. This makes it easy to model a hierarchy of behaviors with different priorities; for example, we could implement a car which drives along a lane, passing slow cars and avoiding collisions, along the following lines:

```
behavior Drive():
    try:
        do FollowLaneBehavior()
    interrupt when self.distanceToNextObstacle() < 20:
        do PassingBehavior()
    interrupt when self.timeToCollision() < 5:
        do CollisionAvoidance()
```

Here, the car begins by lane following, switching to passing if there is a car or other obstacle too close ahead. During *either* of those two sub-behaviors, if the time to collision gets too low, we switch to collision avoidance. Once the `CollisionAvoidance` behavior completes, we will resume whichever behavior was interrupted earlier. If we were executing `PassingBehavior`, it will run to completion (possibly being interrupted again) before we finally resume `FollowLaneBehavior`.

When resuming the interrupted code after an interrupt completes is undesired, using the `abort` statement exits the entire try-interrupt statement. For example, to run a behavior until a condition is met without resuming it later, we can write:

```
behavior ApproachAndTurnLeft():
    try:
        do FollowLaneBehavior()
    interrupt when (distance to intersection) < 10:
        abort    # cancel lane following
    do WaitForTrafficLightBehavior()
    do TurnLeftBehavior()
```

This is a common enough use case of interrupts that Scenic provides a shorthand notation:

```
behavior ApproachAndTurnLeft():
    do FollowLaneBehavior() until (distance to intersection) < 10
    do WaitForTrafficLightBehavior()
    do TurnLeftBehavior()
```

Finally, note that when try-interrupt statements are nested, interrupts of the outer statement take precedence. This makes it easy to build up complex behaviors in a modular way. For example, the behavior `Drive` we wrote above is relatively complicated, using interrupts to switch between several different sub-behaviors. We would like to be able to put it in a library and reuse it in many different scenarios without modification. Interrupts make this straightforward; for example, if for a particular scenario we want a car that drives normally but suddenly brakes for 5 seconds when it reaches a certain area, we can write:

```
behavior DriveWithSuddenBrake():
    haveBraked = False
    try:
        do Drive()
    interrupt when self in targetRegion and not haveBraked:
        do StopBehavior() for 5 seconds
        haveBraked = True
```

With this behavior, `Drive` operates as it did before, interrupts firing as appropriate to switch between lane following, passing, and collision avoidance. But during any of these sub-behaviors, if the car enters the `targetRegion` it will immediately brake for 5 seconds, then pick up where it left off. This example also shows how behaviors can use local variables to maintain state, enabling the encoding of behaviors which make decisions based on actions taken in the past.

Requirements and Monitors. Just as you can declare spatial constraints on scenes using the `require` statement, you can also impose constraints on dynamic scenarios. For example, if we don't want to generate any simulations where `carA` and `carB` are simultaneously visible from the ego car, we could write:

```
require always not ((ego can see carA) and (ego can see carB))
```

The `require always` statement enforces that the given condition must hold at every time step of the scenario; if it is ever violated during a simulation, we reject that simulation and sample a new one. Similarly, we can require that a condition hold at *some* time during the scenario using the `require eventually` statement:

```
require eventually ego in intersection
```

To enforce more complex temporal properties, you can define a *monitor*. Like behaviors, monitors are functions which run in parallel with the scenario and can inspect world state. Here is a monitor for the property "`carA` and `carB` must both enter the intersection before `carC`":

```
1 monitor CarCEntersLast:
2     seenA, seenB = False, False
3     while not (seenA and seenB):
4         require carC not in intersection
5         if carA in intersection:
6             seenA = True
7         if carB in intersection:
8             seenB = True
9         wait
```

We use the variables `seenA` and `seenB` to remember whether we have seen `carA` and `carB` respectively enter the intersection. The loop will iterate as long as at least one of the cars has not yet entered the intersection, so if `carC` enters before either `carA` or `carB`, the requirement on line 4 will fail and we will reject the simulation. Note the necessity of the `wait` statement on line 9: if we omitted it, the loop could run forever without any time actually passing in the simulation.

Preconditions and Invariants. Even general behaviors designed to be used in multiple scenarios may not operate correctly from all possible starting states: for example, `FollowLaneBehavior` assumes that the agent is actually in a lane rather than, say, on a sidewalk. To model such assumptions, Scenic provides a notion of *guards* for behaviors. Most simply, we can specify one or more *preconditions*:

```
behavior MergeInto(newLane):
    precondition: self.lane is not newLane and self.road is newLane.road
    ...
```

Here, the precondition requires that whenever the `MergeInto` behavior is executed by an agent, the agent must not already be in the destination lane but should be on the same road. We can add any number of such preconditions; like ordinary requirements, violating any precondition causes the simulation to be rejected.

Since behaviors can be interrupted, it is possible for a behavior to resume execution in a state it doesn't expect: imagine a car which is lane following, but then swerves onto the shoulder to avoid an accident; naïvely resuming lane following, we find we are no longer in a lane. To catch such situations, Scenic allows us to define *invariants* which are checked at every time step during the execution of a behavior, not just when it begins running. These are written similarly to preconditions:

```
behavior FollowLaneBehavior():
    invariant: self in road
    ...
```

While by default guard violations cause the simulation to be rejected, in some cases it may be possible to recover by taking additional actions. To enable this kind of design, Scenic signals guard violations by raising a `GuardViolation` exception which can be caught like any other exception; the simulation is only rejected if the exception propagates out to the top level. So to model the lane-following-with-collision-avoidance behavior suggested above, we could write code like this:

```
behavior Drive():
    while True:
        try:
            do FollowLaneBehavior()
        interrupt when self.distanceToClosest(Object) < 5:
            do CollisionAvoidance()
        except InvariantViolation:   # FollowLaneBehavior has failed
            do GetBackOntoRoad()
```

When any object comes within 5 meters, we suspend lane following and switch to collision avoidance. When the latter completes, `FollowLaneBehavior` will be resumed; if its invariant fails because we are no longer on the road, we catch the resulting `InvariantViolation` exception and run a `GetBackOntoRoad` behavior to restore the invariant. The whole `try` statement then completes, so the outermost loop iterates and we begin lane following once again.

Terminating the Scenario. By default, scenarios run forever, unless a time limit is specified when running the Scenic tool. However, scenarios can also define termination criteria using the `terminate when` statement; for example, we could decide to end a scenario as soon as the ego car travels at least a certain distance:

```
start = Point on road
ego = Car at start
terminate when (distance to start) >= 50
```

Additionally, the `terminate` statement can be used inside behaviors and monitors: if it is ever executed, the scenario ends. For example, we can use a monitor to terminate the scenario once the ego spends 30 time steps in an intersection:

```
monitor StopAfterTimeInIntersection:
    totalTime = 0
    while totalTime < 30:
        if ego in intersection:
            totalTime += 1
        wait
    terminate
```

4.2 Compositional scenarios

Scenic provides facilities for defining multiple scenarios in a single program and *composing* them in various ways. This enables writing a library of scenarios which can be repeatedly used as building blocks to construct more complex scenarios.

Modular Scenarios. To define a named, reusable scenario, optionally with tunable parameters, Scenic provides the `scenario` statement. For example, here is a scenario which creates a parked car on the shoulder of the `ego`'s current lane (assuming there is one), using some APIs from the driving library:

```
scenario ParkedCar(gap=0.25):
    precondition: ego.laneGroup._shoulder != None
    setup:
        spot = OrientedPoint on visible ego.laneGroup.curb
        parkedCar = Car left of spot by gap
```

The `setup` block contains SCENIC code which executes when the scenario is instantiated, and which can define classes, create objects, declare requirements, etc. as in any of the example scenarios we saw above. Additionally, we can define preconditions and invariants, which operate in the same way as for dynamic behaviors. Having now defined the `ParkedCar` scenario, we can use it in a more complex scenario, potentially multiple times:

```
scenario Main():
    setup:
        ego = Car
    compose:
        do ParkedCar(), ParkedCar(0.5)
```

Here our `Main` scenario itself only creates the ego car; then its `compose` block orchestrates how to run other modular scenarios. In this case, we invoke two copies of the `ParkedCar` scenario in parallel, specifying in one case that the gap between the parked car and the curb should be 0.5 m instead of the default 0.25. So the scenario will involve three cars in total, and as usual SCENIC will automatically ensure that they are all on the road and do not intersect.

Parallel and Sequential Composition. The scenario above is an example of *parallel* composition, where we use the `do` statement to run two scenarios at the same time. We can also use *sequential* composition, where one scenario begins after another ends. This is done the same way as in behaviors: in fact, the `compose` block of a scenario is executed in the same way as a monitor, and allows all the same control-flow constructs. For example, we could write a `compose` block as follows:

```
while True:
    do ParkedCar(gap=0.25) for 30 seconds
    do ParkedCar(gap=0.5) for 30 seconds
```

Here, a new parked car is created every 30 s,[2] with the distance to the curb alternating between 0.25 and 0.5 m. Note that without the `for 30, s` qualifier, we would never get past line 2, since the `ParkedCar` scenario does not define any termination conditions using `terminate when` (or `terminate`) and so runs forever by default. If instead we want to create a new car only when the `ego` has passed the current one, we can use a `do-until` statement:

```
while True:
    subScenario = ParkedCar(gap=0.25)
    do subScenario until distance past subScenario.parkedCar > 10
```

Note how we can refer to the `parkedCar` variable created in the `ParkedCar` scenario as a property of the scenario. Combined with the ability to pass objects as parameters of scenarios, this is convenient for reusing objects across scenarios.

Interrupts, Overriding, and Initial Scenarios. The `try-interrupt` statement used in behaviors can also be used in `compose` blocks to switch between scenarios. For example, suppose we already have a scenario where the `ego` is following a `leadCar`, and want to elaborate it by adding a parked car which suddenly pulls in front of the lead car. We could write a `compose` block as follows:

[2] In a real implementation, we would probably want to require that the parked car is not initially visible from the `ego`, to avoid the sudden appearance of cars out of nowhere.

```
1  following = FollowingScenario()
2  try:
3      do following
4  interrupt when distance to following.leadCar < 10:
5      do ParkedCarPullingAheadOf(following.leadCar)
```

If the `ParkedCarPullingAheadOf` scenario is defined to end shortly after the parked car finishes entering the lane, the interrupt handler will complete and Scenic will resume executing `FollowingScenario` on line 3 (unless the `ego` is still within 10 m of the lead car).

Suppose that we want the lead car to behave differently while the parked car scenario is running; for example, perhaps the behavior for the lead car defined in `FollowingScenario` does not handle a parked car suddenly pulling in. To enable changing the `behavior` or other properties of an object in a sub-scenario, Scenic provides the `override` statement, which we can use as follows:

```
scenario ParkedCarPullingAheadOf(target):
    setup:
        override target with behavior FollowLaneAvoidingCollisions
        parkedCar = Car left of ...
```

Here we override the `behavior` property of `target` for the duration of the scenario, reverting it back to its original value (and thereby continuing to execute the old behavior) when the scenario terminates. The `override` *object specifier*, ... statement has the same syntax as an object definition, and can specify any properties of the object except for dynamic properties like `position` or `speed` which can only be indirectly controlled by taking actions.

In order to allow writing scenarios which can both stand on their own and be invoked during another scenario, Scenic provides a special conditional statement testing whether we are inside the *initial scenario*, i.e., the very first scenario to run.

```
scenario TwoLanePedestrianScenario():
    setup:
        if in initial scenario:   # create ego on random 2-lane road
            roads = filter(lambda r: len(r.lanes) == 2, network.roads)
            road = Uniform(*roads)   # pick uniformly from list
            ego = Car on road
        else:  # use existing ego car; require it is on a 2-lane road
            require len(ego.road.lanes) == 2
            road = ego.road
        Pedestrian on visible road.sidewalkRegion, with behavior ...
```

Random Selection of Scenarios. For very general scenarios, like "driving through a city, encountering typical human traffic", we may want a variety of different events and interactions to be possible. We saw above how we can write behaviors for individual agents which choose randomly between possible actions; Scenic allows us to do the same with entire scenarios. Most simply, since scenarios are first-class objects, we can write functions which operate on them, perhaps choosing a scenario from a list of options based on some complex criterion:

```
chosenScenario = pickNextScenario(ego.position, ...)
do chosenScenario
```

However, some scenarios may only make sense in certain contexts; for example, a red light runner scenario can take place only at an intersection. To facilitate modeling such situations, Scenic provides variants of the `do` statement which randomly choose scenarios to run amongst only those whose preconditions are satisfied:

```
1  do choose RedLightRunner, Jaywalker, ParkedCar(gap=0.5)
2  do shuffle RedLightRunner, Jaywalker, ParkedCar
```

Here, line 1 checks the preconditions of the three given scenarios, then executes one (and only one) of the enabled scenarios. If for example the current road has no shoulder, then `ParkedCar` will be disabled and we will have a 50/50 chance of executing either `RedLightRunner` or `Jaywalker` (assuming their preconditions are satisfied). If *none* of the three scenarios are enabled, Scenic will reject the simulation. Line 2 is a shuffled variant, where *all three* scenarios will be executed, but in random order.[3]

5 Syntax of Scenic

Scenic is an object-oriented PPL, with programs consisting of sequences of statements built with standard imperative constructs including conditionals, loops, functions, and methods (which we do not describe further, focusing on the new elements). Compared to other imperative PPLs, the major restriction of Scenic, made in order to allow more efficient sampling, is that conditional branching may not depend on random variables (except in behaviors). The novel syntax, outlined above, is largely devoted to expressing spatiotemporal relationships in a concise and flexible manner. Figure 9 gives a formal grammar for Scenic, which we now describe in detail.

5.1 Data types

Scenic provides several primitive data types:

Booleans	expressing truth values.
Scalars	as floating-point numbers, which can be sampled from various distributions (see Table 1).
Vectors	representing positions and offsets in space, constructed from coordinates in meters with the syntax `(X, Y)`.[4]
Headings	representing orientations in space. Conveniently, in 2D these are a single angle (in radians, anticlockwise from North). By convention the heading of a local coordinate system is the heading of its y-axis, so, for example, `(-2, 3)` means 2 meters left and 3 ahead.
Vector Fields	associating an orientation to each point in space. For example, the shortest paths to a destination or (in our case study) the nominal traffic direction.
Regions	representing sets of points in space. These can have an associated vector field giving points in the region preferred orientations (e.g. the surface of an object 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 instances must have, together with corresponding default values

[3] Respecting preconditions, so in particular the simulation will be rejected if at some point none of the remaining scenarios to execute are enabled.

[4] The Smalltalk-like (Goldberg and Robson 1983) syntax `X @ Y` used in earlier versions of Scenic is also legal.

```
  program := (statement)*
  boolean := True | False | booleanOp
   scalar := number | distrib | scalarOp
  distrib := baseDist | resample(distrib)
   vector := (scalar, scalar) | Point
               | vectorOp
  heading := scalar | OrientedPoint
               | headingOp
direction := heading | vectorField
    value := boolean | scalar | vector
               | direction | region
               | object | object.property
 classDef := class class[(superclass)]:
               (property: value)*
   object := class specifier, ...
specifier := with property value
               | posSpec | headSpec

behavior := behavior name(params):
              (precondition: boolean)*
              (invariant: boolean)*
              (statement)*
     try := try:
              (statement)*
            (interrupt when boolean:
              (statement)*)*
            (except exception:
              (statement)*)*
scenario := scenario name(params):
              (precondition: boolean)*
              (invariant: boolean)*
              [setup:
                (statement)*]
              [compose:
                (statement)*]
```

Fig. 9 Simplified SCENIC grammar. *Point* and *OrientedPoint* are instances of the corresponding classes. See Table 5 for statements, Fig. 11 for operators, Table 1 for *baseDist*, and Tables 3 and 4 for *posSpec* and *headSpec*

Table 1 Built-in distributions

Syntax	Distribution
`Range` (*low*, *high*)	Uniform on continuous interval
`Uniform` (*value*, ...)	Uniform over discrete values
`Discrete` ({*value*: *Weight*, ...})	Discrete with weights
`Normal` (*mean*, *stdDev*)	Normal (Gaussian)
`TruncatedNormal` (*mean*, *stdDev*, *low*, *high*)	Normal, truncated to the given window

All parameters are *scalars* except *value*

(see Fig. 9). Default value expressions are evaluated each time an object is created. Thus if we write `weight: Range(1, 5)` when defining a class then each instance will have a `weight` drawn *independently* from `Range(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 `length` of a `Car` are by default derived from its `model`.

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

To allow cleaner notation, `Point` and `OrientedPoint` are automatically interpreted as vectors or headings in contexts expecting these (as shown in Fig. 9). 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.

Table 2 Properties of the built-in classes `Point`, `OrientedPoint`, and `Object`

Property	Default	Meaning
`position`	(0, 0)	Position in global coordinates
`viewDistance`	50	Distance for '`can see`' predicate
`mutationScale`	0	Overall scale of mutations
`positionStdDev`	1	Mutation σ for `position`
`heading`	0	Heading in global coordinates
`viewAngle`	360°	Angle for '`can see`' predicate
`headingStdDev`	5°	Mutation σ for `heading`
`width`	1	Width of bounding box
`length`	1	Length of bounding box
`speed`	0	Speed of object
`velocity`	(0, 0)	Velocity (default from `speed`, `heading`)
`angularSpeed`	0	Angular speed (in rad/s)
`behavior`	`None`	Dynamic behavior, if any
`allowCollisions`	False	Collisions allowed
`requireVisible`	True	Must be visible from `ego`
`regionContainedIn`	`None`	Region object must be contained in

`taxi relative to limo`, are illegal (caught by a simple type system); the more verbose syntax must be used instead.

5.2 Expressions

SCENIC's expressions are mostly straightforward, largely consisting of the arithmetic, boolean, and geometric operators shown in Fig. 11. The meanings of these operators are largely clear from their syntax, so we defer complete definitions of their semantics to the Appendix (Fremont et al. 2020a). Figure 10 illustrates several of the geometric operators (as well as some specifiers, which we will discuss in the next section). Various points to note:

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

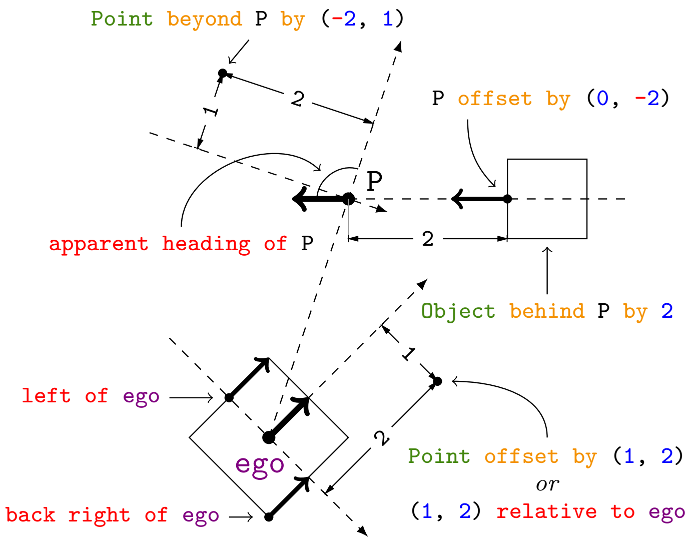


Fig. 10 Various SCENIC operators and specifiers applied to the `ego` object and an `OrientedPoint` `P`. Instances of `OrientedPoint` are shown as bold arrows

```
scalarOperator := max(scalar, ...) | min(scalar, ...)
  | -scalar | abs(scalar) | scalar (+ | *) scalar
  | relative heading of heading [from heading]
  | apparent heading of OrientedPoint [from vector]
  | distance to vector [from vector]
  | distance [of OrientedPoint] past vector
  | angle [from vector] to vector
booleanOperator := not boolean
  | boolean (and | or) boolean
  | scalar (== | != | < | > | <= | >=) scalar
  | (Point | OrientedPoint) can see (vector | Object)
  | (vector | Object) in region
headingOperator := scalar deg
  | vectorField at vector
  | direction relative to direction
vectorOperator := vector relative to vector
  | vector offset by vector
  | vector offset along direction by vector
regionOperator := visible region
  | region visible from (Point | OrientedPoint)
orientedPointOperator :=
    vector relative to OrientedPoint
  | OrientedPoint offset by vector
  | (front | back | left | right) of Object
  | (front | back) (left | right) of Object
```

Fig. 11 Operators by result type

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

```
x = Range(0, 1)
y = (x, x)
```

does not make `y` uniform over the unit box, but rather over its diagonal. For convenience in sampling multiple times from a primitive distribution, Scenic provides a `resample`(*D*) function returning an independent[5] sample from *D*, one of the distributions in Table 1. Scenic also allows defining custom distributions beyond those in the Table.

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

5.3 Specifiers

As shown in the grammar in Fig. 9, an object is created by writing the class name followed by a (possibly empty) comma-separated list of specifiers. The specifiers are combined, possibly adding default specifiers from the class definition, to form a complete specification of all properties of the object. Arbitrary properties (including user-defined properties with no meaning in Scenic) can be specified with the generic specifier `with` *property value*, while Scenic provides many more specifiers for the built-in properties `position` and `heading`, shown in Tables 3 and 4 respectively.

In general, a specifier is a function taking in values for zero or more properties, its *dependencies*, and returning values for one or more other properties, some of which can be specified *optionally*, meaning that other specifiers will override them. For example, `on` *region* specifies `position` and optionally specifies `heading` if the given region has a preferred orientation. If `road` is such a region, as in our case study, then `Object on road` will create an object at a position uniformly random in `road` and with the preferred orientation there. But since `heading` is only specified optionally, we can override it by writing `Object on road, facing 20 deg`.

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

As the semantics of the specifiers in Tables 3 and 4 are largely evident from their syntax, we defer exact definitions to the Appendix (Fremont et al. 2020a). We briefly discuss some of the more complex specifiers, referring to the examples in Fig. 10:

- `behind` *vector* means the object is placed with the midpoint of its front edge at the given vector, and similarly for `ahead`/`left`/`right of` *vector*.
- `beyond` *A* `by` *O* `from` *B* means the position obtained by treating *O* as an offset in the local coordinate system at *A* oriented along the line of sight from *B*. In this and other specifiers, if the `from` *B* is omitted, the ego object is used by default. So for example

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

Table 3 Specifiers for `position`

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

The specifiers in the second group also optionally specify `heading`

Table 4 Specifiers for `heading`

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

`beyond taxi by (0, 3)` means 3 m directly behind the taxi as viewed by the camera (see Fig. 10 for another example).

- The `heading` optionally specified by `left of` *OrientedPoint*, etc. is that of the `OrientedPoint` (thus in Fig. 10, `P offset by (0, -2)` yields an `OrientedPoint` facing the same way as `P`). Similarly, the `heading` optionally specified by the `following` *vectorField* specifier is that of the vector field at the specified `position`.
- `apparently facing` *H* means the object has heading *H* with respect to the line of sight from `ego`. For example, `apparently facing 90 deg` would orient the object so that the camera views its left side head-on.

5.4 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.

Selecting a world model. The `model` *name* statement specifies that the SCENIC program is written for the given SCENIC world model. It is equivalent to the statement `from` *name* `import *` (as in Python), importing everything from the given SCENIC module, but can be overridden from the command-line when running the SCENIC tool. This enables writing cross-platform scenarios using abstract domains like `scenic.domains.driving`, then

Table 5 Statements (excluding `if`, `while`, `def`, `import`, etc. from Python)

Syntax	Meaning
`model` *name*	Select world model
name = *value*	Variable assignment
`param` *name* = *value*, …	Global parameter assignment
classDefn (see Fig. 9)	Class definition
object	Object definition
behavior	Behavior definition
monitor	Monitor definition
scenario	Modular scenario definition
`require` *boolean*	Hard requirement
`require[`*number*`]` *boolean*	Soft requirement
`require always` *boolean*	Always dynamic requirement
`require eventually` *boolean*	Eventually dynamic requirement
`terminate when` *boolean*	Termination condition
`mutate` *name*, … [by *number*]	Enable mutation
`take` *action*, …	Invoke action(s)
`wait`	Invoke no actions this step
`terminate`	End scenario immediately
`do` *name*, …	Invoke sub-behavior(s)/sub-scenario(s)
`do` *name*, … `for` *scalar* (`seconds` | `steps`)	Invoke with time limit
`do` *name*, … `until` *boolean*	Invoke until condition
try (see Fig. 9)	Try-interrupt statement
`abort`	Abort try-interrupt statement
`override` *name specifier*, …	Override object properties dynamically

The statements in the second group are only legal inside behaviors, monitors, and `compose` blocks

executing them in particular simulators by overriding the model with a more specific module (e.g. `scenic.simulators.carla.model`).

Global parameters. The statement `param` *name* = *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.

Behaviors and monitors. The `behavior` statement (see Fig. 9) defines a dynamic behavior. A behavior definition has the same structure as a function definition, with two differences: (1) it may begin with any number of `precondition`: *boolean* and `invariant`: *boolean* lines defining preconditions and invariants; (2) it may use the statements in the second section of Table 5, which are not allowed in ordinary functions. The `monitor` statement has the same structure as a `behavior` statement but defines a monitor.

Modular scenarios. The `scenario` statement (see Fig. 9) defines a modular scenario which can be invoked from another scenario. Scenario definitions begin like behavior definitions, with a name, parameters, preconditions, and invariants. However, the body of a scenario consists of two parts, either of which can be omitted: a `setup` block and a `compose` block.

The setup block contains code that runs once when the scenario begins to execute, and is a list of statements like a top-level Scenic program.[6] The compose block orchestrates the execution of sub-scenarios during a dynamic scenario, and may use do and any of the other statements allowed inside behaviors (except take, which only makes sense for an individual agent).

Requirements. The require *boolean* statement requires that the given condition hold in all generated scenes (equivalently to *observe* statements in other probabilistic programming languages; see e.g. Milch et al. (2004); Claret et al. (2013)). The variant 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. The require always and require eventually variants define requirements that must hold in *every* and *some* time step of dynamic simulations respectively.

Mutation. 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.

Termination conditions. The terminate when *boolean* statement defines a condition which is monitored as in require eventually, but which when true causes the scenario to end. The terminate statement can be called inside a behavior, monitor, or compose block to end the scenario immediately.

Actions. The take *action*, ... statement can be used inside behaviors to select one or more actions[7] for the agent to take in the current time step. The wait statement means no actions are taken in this time step (which makes sense inside monitors and compose blocks). When either of these statements is executed, the behavior is suspended until one time step has elapsed; then its invariants are checked (raising an InvariantViolation exception if any are violated) and it is resumed.

Invoking other behaviors and scenarios. The do *name*, ... statement has the same structure as the take statement, but invokes one or more behaviors (if in a behavior) or scenarios (if in a compose block). It does not return until the sub-behavior/sub-scenario terminates, so multiple time steps may pass (unlike take). Early termination can be enabled by adding a for *scalar* seconds/steps clause, which enforces a maximum time limit, or an until *boolean* clause, which adds an arbitrary termination criterion. When the do statement returns, the invariants of the calling behavior/scenario are checked as above.

Interrupts. The try statement (see Fig. 9) consists of a try: block and one or more interrupt when *boolean*: and except *exception*: blocks, each containing arbitrary lists of statements. As described in Sect. 4.1, when a try statement executes, the conditions for each interrupt when block are checked at each time step. While none of them are true, the try block executes. When an interrupt condition becomes true, the body of the corresponding block is executed (with lower blocks preempting those above), suspending any behaviors/scenarios that were executing in the try block until the interrupt handler

[6] In fact, a top-level Scenic program is equivalent to an unnamed scenario definition with no parameters, preconditions, invariants, or compose block, and whose setup block consists of the whole program.

[7] The statement will accept lists and tuples of actions, in order to support taking a number of actions that is not fixed, i.e., if myActions is a list of actions, we can write take myActions.

completes (at which point the invariants of the suspended behavior/scenario are checked as usual). Any exceptions raised in the `try` block or any interrupt handler can be caught by `except` blocks as in the Python `try` statement. Additionally, any block may execute the `abort` statement to immediately terminate the entire `try` statement.

Overrides. The `override` *name specifier*, ... statement may be used inside a scenario definition to override properties of an object during a dynamic scenario. It has the same structure as an object definition, with `override` and the name of the object replacing the class, so for example given an object `taxi` we could write `override taxi with aggression 3` to set the `aggression` property of `taxi` to 3. Dynamic properties read back from the simulator at every time step, like `position`, cannot be overridden since they are controlled using actions and not direct assignments. Properties overridden by a scenario revert to their original values when the scenario terminates. When the `behavior` property is overridden, the original behavior is suspended, then resumed at the end of the scenario.

6 Semantics and concrete scenario generation

6.1 Semantics of Scenic

The output of a Scenic program has two parts: first, a *scene* consisting of an assignment to all the properties of each `Object` defined in the scenario, plus any global parameters defined with `param`. For dynamic scenarios, this scene forms the initial state of the scenario, which then changes after each time step according to the actions taken by the agents. Since actions and their effects are domain-specific (consider for example the different physics involved for aerial, ground, and underwater vehicles), dynamic Scenic scenarios do not directly define trajectories for objects. Instead, the second part of the output of a Scenic program is a *policy*, a function mapping the history of past scenes to the choice of actions for the agents in the current time step.[8] This pair of a scene and a policy is what we mean formally by the *concrete scenario* generated by a Scenic program.

Since Scenic is a probabilistic programming language, the semantics of a program is actually a *distribution* over possible outputs, here concrete scenarios. As for other imperative PPLs (with declarative constraints), the semantics can be defined operationally as a typical interpreter for an imperative language but with two differences to handle random sampling and constraints. First, the interpreter makes random choices when evaluating distributions (Saheb-Djahromi 1978). For example, the Scenic statement `x = Range(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., Gordon et al. (2014)). For example, adding the statement `require x > 0.5` above would yield a uniform distribution for `x` over the interval (0.5, 1).

In order to support efficient sampling, the Scenic tool does not directly implement an interpreter along the lines above; instead, it compiles a Scenic program into an intermediate representation, an *expression forest*, which preserves the structure of the distributions defined in the program. The expression forest is a directed acyclic graph where each vertex

[8] In fact the policy is a probabilistic function, since behaviors can make random choices, and it can also return special values indicating that the scenario should terminate or that it has violated a requirement and should be discarded, as we discuss below.

is a random-valued expression occurring in the program, and edges indicate dependencies between expressions.[9] For example, in the program `Car at x offset by y`, the forest would have a root node for the `position` property of the car; that node would have a single child representing `x offset by y`, which in turn would have children representing `x` and `y`. The SCENIC sampler works by traversing the expression forest in topological order from leaves to roots, sampling a value for each node after values for all of its dependencies have already been determined. This yields the same distribution that would be obtained by simply "running" the SCENIC code as usual in imperative PPLs; by rejecting any samples which violate `require` statements, we obtain a scene distribution conditioned on the requirements being satisfied, as desired. However, the structural information in the expression forest allows us to improve on this simplistic rejection sampling approach by performing transformations on the forest that reduce the probability of rejection while leaving the conditioned distribution the same. These transformations take advantage of the domain-specific syntax of SCENIC, using pattern matching to identify subtrees representing certain geometric relationships in the forest and replace them with "pruned" versions that exclude parts of the parameter space which would be guaranteed to violate built-in or user-defined requirements. We describe several such pruning techniques in Sect. 6.2. For clarity, since these techniques do not change the semantics of the program, in the rest of this section and in the Appendix we describe the semantics of SCENIC constructs in terms of a simple imperative interpreter.

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 the Appendix (Fremont et al. 2020a) for a more formal treatment.

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

Specifiers and object definitions. As we saw above, each specifier defines a function mapping values for its dependencies to values for the properties it specifies. When an object of class C is constructed using a set of specifiers S, the object is defined as follows (see the Appendix (Fremont et al. 2020a) for details):

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

Mutation. The `mutate X by N` statement sets the special `mutationScale` property to `N` (the `mutate X` form sets it to 1). At the end of evaluation of the SCENIC program, but

[9] Such forests are similar to the abstract syntax trees commonly used in compilers, except that in our case the forest tracks only the semantic relationships between distributions, not the syntax used to define those relationships. For example, the assignment `x = y` would not be saved in the forest at all: the node for `x` would simply be the same as the node for `y`. The lack of conditional control flow in SCENIC (outside behaviors) makes it possible to determine such relationships at compile time.

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`.

Dynamic constructs. As suggested in Sect. 5.4, behaviors and monitors are coroutines: they usually execute like ordinary functions, but are suspended when they `take` an action (or `wait`) until one time step has passed. Scenarios behave similarly: in their `compose` blocks, using `wait` causes them to wait for one step, and any sub-scenarios they invoke using `do` run recursively; scenarios without `compose` blocks do nothing in a time step other than check whether any of their `terminate when` conditions have been met or their `require always` conditions violated.

The output of the policy of a dynamic SCENIC program is defined according to the following procedure:

1. Run the `compose` blocks of all currently-running scenarios for one time step. If any `require` conditions fail, discard the simulation. If instead the top-level scenario finishes its `compose` block (if any), one of its `terminate when` conditions is true, or it executes `terminate`, set a flag to remember this (we use a flag rather than terminating immediately since we need to ensure that all requirements are satisfied before terminating).
2. Check all `require always` conditions of currently-running scenarios; if any fail, discard the simulation.
3. Run all monitors of currently-running scenarios for one time step. As above, discard the simulation if any `require` conditions fail, and set the terminate flag if the `terminate` statement is executed.
4. If the flag is set, check that all `require eventually` conditions were satisfied at some time step: if so, terminate the simulation; otherwise, discard it.
5. Run all the behaviors of dynamic agents for one time step, gathering their actions and discarding the simulation or setting the terminate flag as in (3).
6. Repeat (4) to check the terminate flag.
7. Return the choice of actions selected by the dynamic agents.

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,[10] the domain-specific design of SCENIC enables specialized sampling methods, which we discuss below. We also note that the scenario generation problem is closely related to *control improvisation*, an abstract framework capturing various problems requiring synthesis under hard, soft, and randomness constraints (Fremont et al. 2015; Fremont 2019). *Scenario improvisation* from a SCENIC program can be viewed as an extension with a more detailed randomness constraint given by the imperative part of the program.

6.2 Domain-specific sampling techniques

The geometric nature of the constraints in SCENIC programs, together with SCENIC's lack of conditional control flow outside behaviors, enable domain-specific sampling techniques

[10] Note however that the presence of dynamic agents complicates the use of standard PPL techniques, since the fact that the physics relating actions to their effects on the world is not modeled in SCENIC means that the program effectively contains an unknown, black-box function. For the same reason, SCENIC does not currently analyze dynamic behaviors using expression trees, as it does for the static part of the program.

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. Furthermore, by combining spatial and temporal constraints, we can prune some initial scenes by proving that they *force* a requirement to be violated at some future point during a dynamic scenario. We describe several pruning techniques below, deferring formal statements of the algorithms to the Appendix (Fremont et al. 2020a).

Pruning based on containment. The simplest technique applies to any object X whose position is uniform in a region R and which must be contained in a region C (e.g. the road in our case study). If *minRadius* is a lower bound on the distance from the center of X to its bounding box, then we can restrict R to $R \cap \mathit{erode}(C, \mathit{minRadius})$. This is sound, since if X is centered anywhere not in the restriction, then some point of its bounding box must lie outside of C.

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

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

Pruning based on reachability. Finally, we can prune initial positions for objects which make it impossible to reach a goal location within the duration of the scenario; for example, a car which travels down a road and then runs a red light must start sufficiently close to an intersection. Suppose an object is required to enter a region R within T time (either by an explicit `require eventually` statement or a precondition of a behavior or scenario guaranteed to eventually execute) and we have an upper bound S on the object’s speed. Then we can prune away all initial positions of the object which do not lie within a distance $D = ST$ of R, i.e., we can restrict its initial positions to $\mathit{dilate}(R, D)$. If the object is also required to stay within some containing region C (e.g., a road) for the entire duration of the scenario, we can compute a tighter value of D by considering only paths that lie within C.
After pruning the space as described above, our implementation uses rejection sampling, generating scenes from the imperative part of the scenario until all requirements are satisfied. While this samples from exactly the desired distribution, it has the drawback that a huge number of samples may be required to yield a single valid scene (in the worst case, when the requirements have probability zero of being satisfied, the algorithm will not even terminate). However, we found in our experiments that all reasonable scenarios we tried required at most several hundred iterations of rejection sampling, yielding a sample within a few seconds. Furthermore, the pruning methods above could reduce the number of samples needed by a factor of 3 or more (see the Appendix (Fremont et al. 2020a) for details of our experiments).

In future work it would be interesting to see whether Markov chain Monte Carlo methods previously used for probabilistic programming (see, e.g., Milch et al. (2004); Nori et al. (2014); Wood et al. (2014)) could be made effective in the case of Scenic.

7 Experiments

We demonstrate the three applications of Scenic discussed in Sect. 2: testing a system under particular conditions, either a perception component in isolation Sect. 7.2.1 or a dynamic closed-loop system Sect. 7.2.2, training a system to improve accuracy in hard cases Sect. 7.3, and debugging failures Sect. 7.4. We begin by describing the general experimental setup.

7.1 Experimental setup

For our main case study, we generated scenes in the virtual world of the video game Grand Theft Auto V (GTAV) (Rockstar Games 2015). We wrote a Scenic world model defining `Region`s representing the roads and curbs in (part of) this world, as well as a type of object `Car` providing two additional properties[11]: `model`, representing the type of car, with a uniform distribution over 13 diverse models provided by GTAV, and `color`, representing the car color, with a default distribution based on real-world car color statistics (DuPont 2012). In addition, we implemented two global scene parameters: `time`, representing the time of day, and `weather`, representing the weather as one of 14 discrete types supported by GTAV (e.g. “clear” or “snow”).

GTAV is closed-source and does not expose any kind of scene description language. Therefore, to import scenes generated by Scenic into GTAV, we wrote a plugin based on DeepGTAV.[12] The plugin calls internal functions of GTAV to create cars with the desired positions, colors, etc., as well as to set the camera position, time of day, and weather.

Our experiments used SqueezeDet (Wu et al. 2017), a convolutional neural network real-time object detector for autonomous driving.[13] We used a batch size of 20 and trained all models for 10,000 iterations unless otherwise noted. Images captured from GTAV with resolution 1920×1200 were resized to 1248×384, the resolution used by SqueezeDet and the standard KITTI benchmark (Geiger et al. 2012). All models were trained and evaluated on NVIDIA TITAN XP GPUs.

We used standard metrics *precision* and *recall* to measure the accuracy of detection on a particular image set. The accuracy is computed based on how well the network predicts the correct bounding box, score, and category of objects in the image set. Details are in the Appendix (Fremont et al. 2020a), 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.

[11] For the full definition of `Car`, see the Appendix (Fremont et al. 2020a); the definitions of `road`, `curb`, etc. are a few lines loading the corresponding sets of points from a file storing the GTAV map (see the Appendix for how this file was generated).

[12] https://github.com/aitorzip/DeepGTAV.

[13] Used industrially, for example by DeepScale (http://deepscale.ai/).

7.2 Testing and falsification

We begin with the most straightforward application of Scenic, namely generating specialized data to test a system under particular conditions. We demonstrate both using a static scenario to test a perception component, and using a dynamic scenario to falsify a closed-loop system.

7.2.1 Testing a perception module

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 static scenes of 1–4 cars (not counting the camera), specifying only that the cars face within 10° of the road direction: all other features had their default distributions, e.g. the cars were positioned uniformly at random over the road and the time of day was uniform over an entire 24 h period. We generated 1000 images from each scenario, yielding a training set X_{generic} of 4000 images, and used these to train a model M_{generic} as described in Sect. 7.1. We also generated an additional 50 images from each scenario to obtain a generic test set T_{generic} of 200 images. For all of our scenarios (including in our other experiments), sampling a single scene and rendering an image from it took at most several seconds.

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

Evaluating M_{generic} on T_{generic}, T_{good}, and T_{bad}, we obtained precisions of 83.1, 85.7, and 72.8%, respectively, and recalls of 92.6, 94.3, and 92.8%. This shows that, as might be expected, the model performs better on bright days than on rainy nights. This suggests there might not be enough examples of rainy nights in the training set, and indeed under our default weather distribution rain is less likely than shine. This illustrates how specialized test sets can highlight the weaknesses and strengths of a particular model. In Sect. 7.3, we go one step further and use Scenic to redesign the training set and improve model performance.

7.2.2 Falsifying a dynamic closed-loop system

Next, we demonstrate how we can use a dynamic Scenic scenario to test a closed-loop system, using VerifAI's falsification facilities to monitor and analyze counterexamples to a system-level specification. We tested an autonomous agent[14] in the CARLA (Dosovitskiy et al. 2017) driving simulator, for which we wrote a similar Scenic world model as we did for GTA V. This agent consists of a planner and controller (but no perception components) which implement basic driving behaviors including abiding by traffic lights, lane following, and collision avoidance.

We wrote a Scenic program describing a scenario where the ego vehicle (i.e. the autonomous agent) is performing a right turn at an intersection, yielding to the crossing traffic. As the ego approaches the intersection, the traffic light turns green, but a crossing car runs the red light. The ego vehicle has to decide either to yield or make a right turn. The crossing car executes a reactive behavior where it slows down to maintain a minimum distance with any car in front.

[14] https://github.com/carla-simulator/carla/blob/dev/PythonAPI/examples/automatic_control.py.

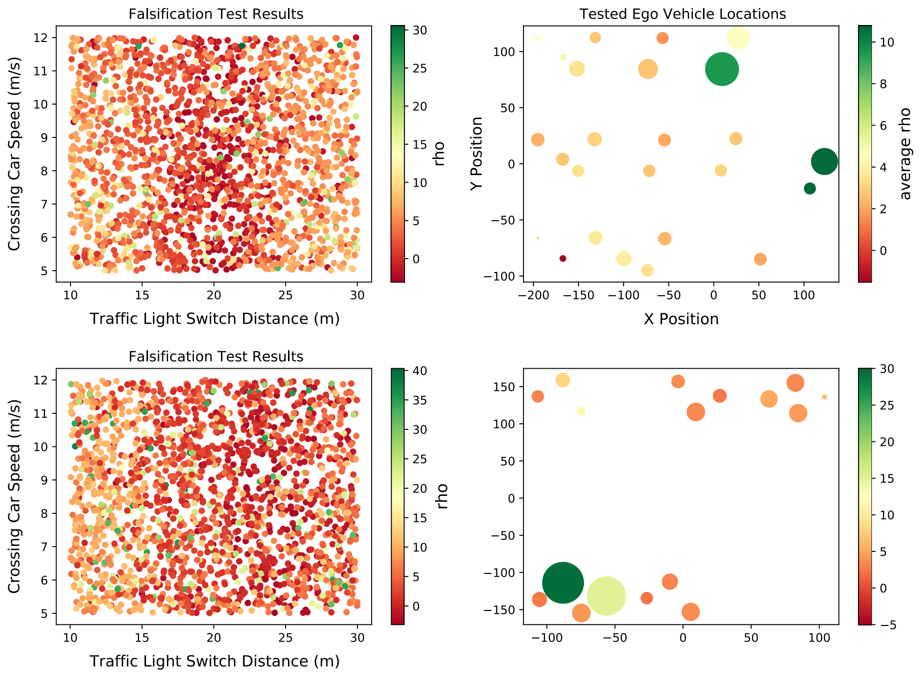


Fig. 12 Falsification results in CARLA. Top: Town05; bottom: Town03

We allowed three environment parameters to vary in this scenario (code for which can be found in the Appendix (Fremont et al. 2020a)) :

– The traffic light's transition from red to green is triggered when the distance between the ego and the crossing car reaches a threshold, which was uniformly random between 10–30 m.
– The crossing car's speed was uniformly random between 5–12 m/s.
– The scenario takes place at a random 4-way intersection in the CARLA map. To demonstrate how Scenic programs can be written in a generic, map-agnostic style, we used the same Scenic code on two different CARLA maps (Town05 and Town03).

We formulated a safety specification for the autonomous agent in Metric Temporal Logic, stating that the distance between the agent and the crossing car must be greater than 5 meters at all times. Giving this specification and the Scenic program to VerifAI, we generated 2,000 scenarios for each map. VerifAI monitored each simulation and computed the *robustness value* ρ of the MTL specification, which measures how strongly the specification was satisfied (Koymans 1990) (negative values meaning it was violated).

Our results are shown in Fig. 12. On the left, we plot ρ as a function of the traffic light trigger threshold and the speed of the crossing car. Each dot represents one simulation, with redder colors indicating smaller ρ, i.e., being closer to violating the safety specification. We found a significant number of violations, approximately 21% and 17% of tests on Town05 and Town03 respectively. From the plots we observe broadly similar behavior across the two maps, with the distance when the traffic light switch occurs being the dominant factor controlling failures of the autonomous agent (most failures occurring for values of 15–25 m).

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

Fig. 13 A scenario where one car partially occludes another. The property `roadDeviation` is defined in `Car` to mean its `heading` relative to the `roadDirection`

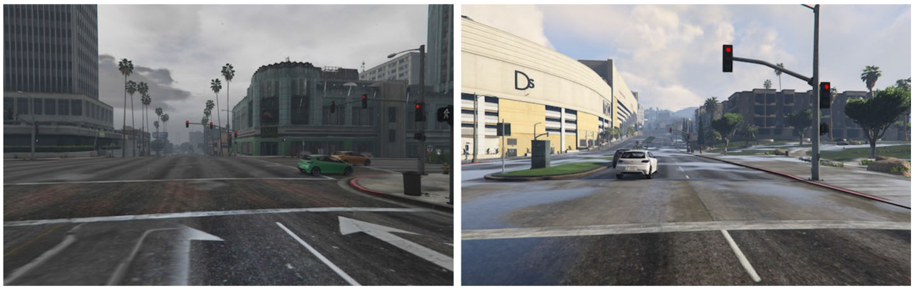


Fig. 14 Two scenes generated from the partial-occlusion scenario

On the right side of Fig. 12, we plot the average value of ρ at each intersection, with color again indicating the average value of ρ and the size of each dot being proportional to its variance. We can see that some intersections are much easier or harder for the autonomous agent to handle. Investigating some of the most extreme intersections, we observed that those with 4-lane legs and a turning radius of about 6.5 m caused the agent to fail most frequently. Re-testing the agent at such intersections, we found that this geometry often created a situation where the agent and the crossing car were merging into the same lane simultaneously, instead of one car completing its maneuver before the other.

These results show how we can use SCENIC to find scenarios where a closed-loop system violates its specification. In Sect. 7.4, we will further show how SCENIC can help us diagnose the root causes of failures and eliminate them through retraining.

7.3 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. 13, which generates such scenes by placing one car behind the other as viewed from the camera, offset left or right so that it is at least partially visible; Fig. 14 shows some of the resulting images. Generating images from this scenario we obtained a training set X_{overlap} of 250 images and a test set T_{overlap} of 200 images.

For a baseline training set we used the "Driving in the Matrix" synthetic data set (Johnson-Roberson et al. 2017), which has been shown to yield good car detection performance even

Table 6 Performance of models trained on 5000 images from X_{matrix} or a mixture with X_{overlap}, averaged over 8 training runs with random selections of images from X_{matrix}

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

on real-world images.[15] Like our images, the “Matrix” images were rendered in GTAV; however, rather than using a PPL to guide generation, they were produced by allowing the game’s AI to drive around randomly while periodically taking screenshots. We randomly selected 5000 of these images to form a training set X_{matrix}, and 200 for a test set T_{matrix}. We trained SqueezeDet for 5,000 iterations on X_{matrix}, evaluating it on T_{matrix} and T_{overlap}. To reduce the effect of jitter during training we used a standard technique (Arlot and Celisse 2010), saving the last 10 models in steps of 10 iterations and picking the one achieving the best total precision and recall. This yielded the results in the first row of Table 6. Although X_{matrix} contains many images of overlapping cars, the precision on T_{overlap} is significantly lower than for T_{matrix}, indicating that the network is predicting lower-quality bounding boxes for such cars.[16]

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

7.4 Debugging failures

In our final experiment, we show how Scenic can be used to generalize a single input on which a model fails, exploring its neighborhood in a variety of different directions and giving insight into which features of the scene are responsible for the failure. The original failure can then be generalized to a broader scenario describing a class of inputs on which the model misbehaves, which can in turn be used for retraining. We selected one scene from our first experiment, shown in Fig. 15, consisting of a single car viewed from behind at a slight angle, which M_{generic} wrongly classified as three cars (thus having 33.3% precision and 100% recall). We wrote several scenarios which left most of the features of the scene fixed but allowed others to vary. Specifically, scenario (1) varied the model and color of the

[15] We use the “Matrix” data set since it is known to be effective for car detection and was not designed by us, making the fact that Scenic is able to improve it more striking. The results of this experiment also hold under the Average Precision (AP) metric used in Johnson-Roberson et al. (2017), as well as in a similar experiment using the Scenic generic two-car scenario from the last section as the baseline. See Appendix (Fremont et al. 2020a) for details.

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

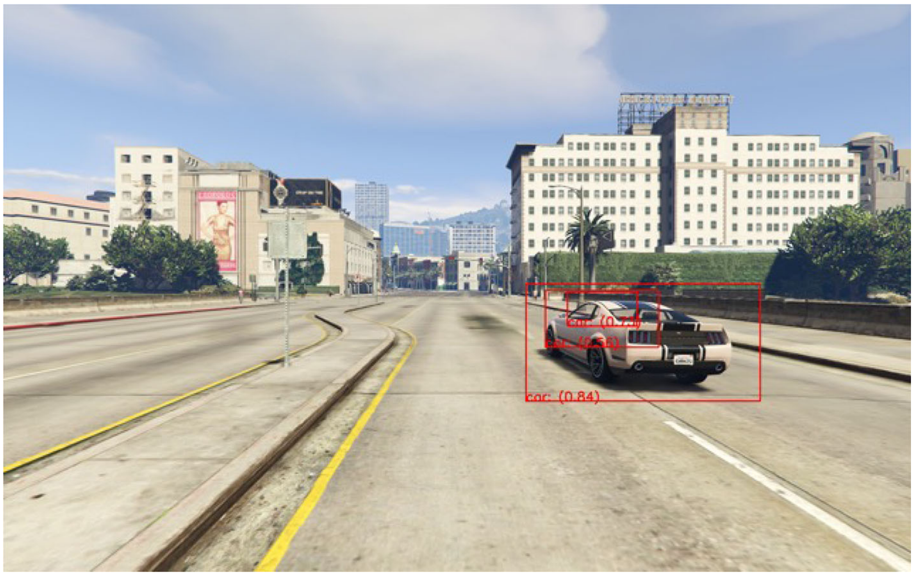


Fig. 15 The misclassified image, with the predicted bounding boxes

Table 7 Performance of M_{generic} on different scenarios representing variations of the image in Fig. 15

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

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. For each scenario we generated 150 images and evaluated M_{generic} on them. As seen in Table 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 Table 7. The results confirmed the importance of model and color [compare (2)–(7)], as well as angle [compare (5)–(6)], but also suggested that being close to the camera could be the relevant aspect of the car's local position. We confirmed this with a final round of scenarios [compare (5) and (8)], which also showed that the effect of car model is small among scenes where the car is close to the camera [compare (4) and (9)].

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

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

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_{generic}, hoping to improve performance on its original test set T_{generic} (to better distinguish small differences in performance, we increased the test set size to 400 images). To keep the size of the training set fixed as in the previous experiment, we replaced 400 one-car images in X_{generic} (10% of the whole training set) with images generated from our scenarios. As a baseline, we used images produced with classical image augmentation techniques implemented in `imgaug` (Jung 2018). Specifically, we modified the original misclassified image by randomly cropping 10–20% on each side, flipping horizontally with probability 50%, and applying Gaussian blur with $\sigma \in [0.0, 3.0]$.

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

8 Related work

Synthetic data generation. There has been a large amount of work on generating synthetic data for specific applications, including text recognition (Jaderberg et al. 2014), text localization (Gupta et al. 2016), robotic object grasping (Tobin et al. 2017), and autonomous driving (Johnson-Roberson et al. 2017; Filipowicz et al. 2017). 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 (Liebelt et al. 2010; Stark et al. 2010), pedestrian detection (Vazquez et al. 2014), and semantic image segmentation (Ros et al. 2016). Such work provides important context for our paper, showing that models trained exclusively on synthetic data (possibly domain-adapted) can achieve acceptable performance on real-world data. The major difference in our work is that we provide, through Scenic, language-based systematic data generation for *any* cyber-physical system.

A closely-related area is that of Generative Adversarial Networks (GANs) (Goodfellow et al. 2014a), a particular kind of neural network able to generate realistic synthetic data, which has been used to augment training sets (Liang et al. 2017; Marchesi 2017). The difference with Scenic is that GANs require an initial training set/pretrained model and do not easily incorporate declarative constraints, while Scenic produces synthetic data in an explainable, programmatic fashion requiring only a simulator. At present, achieving precise control over the contents of images generated by GANs is challenging. However, in future it would be interesting to explore using GANs in combination with Scenic, either to improve the realism of the generated data (as in domain adaptation), or more interestingly, using Scenic to generate some of the latent variables of the GAN, thereby providing some level of controllability.

Robustness checking and adversarial ML. Adversarial machine learning (Szegedy et al. 2014) is a field which focuses on the analysis of ML algorithms against adversarial attacks and the design of models robust to such attacks. Some of these methods generate misclassified examples by looking at the model gradient and by finding minimal input perturbations that lead to a misclassification (Szegedy et al. 2013; Goodfellow et al. 2014b; Moosavi-Dezfooli et al. 2016; Nguyen et al. 2015). Other techniques assume the model to be gray/black-box and focus on input modifications or high-level properties of the model (Pei et al. 2017; Dreossi et al. 2017, 2018). Based on these analyses, some works have explored the idea of using adversarial examples (i.e. misclassified examples) to retrain and improve ML models (e.g., Xu et al. 2016; Wong et al. 2016; Goodfellow et al. 2014b; Dreossi et al. 2018). Our work on Scenic is complementary to most prior work on adversarial ML, which usually considers attacks consisting of small pixel-level perturbations to the input images. By contrast, Scenic is part of a line of work on *semantic* adversarial ML (Dreossi et al. 2018), enabling search through a space of meaningful, high-level features rather than individual pixel values.

Model-based test generation. Techniques using a model to guide test generation have long existed Broy et al. 2005. A popular approach is to provide *example tests*, as in mutational fuzz testing (Sutton et al. 2007) and example-based scene synthesis (Fisher et al. 2012). 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 (Sutton et al. 2007), procedural generation from shape grammars (Müller et al. 2006), and grammar-based scene synthesis (Jiang et al. 2018). 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 (Elmas et al. 2013). Conversely, *constraints* as in constrained-random verification (Naveh et al. 2006) allow global properties but can be difficult to write. Scenic improves on these methods by simultaneously providing fine-grained control, enforcement of global properties, specification of probability distributions, and simple imperative syntax.

Probabilistic programming languages. The semantics (and to some extent, the syntax) of Scenic are similar to that of other probabilistic programming languages such as Prob (Gordon et al. 2014), Church (Goodman et al. 2008), and BLOG (Milch et al. 2004). In probabilistic programming the focus is usually on *inference* rather than *generation* (the main application in our case), and in particular to our knowledge probabilistic programming languages have not previously been used for test generation. However, the most popular inference techniques are based on sampling and so could be directly applied to generate scenes from Scenic programs, as we discussed in Sect. 6.

Several probabilistic programming languages have been used to define generative models of objects and scenes: both general-purpose languages such as WebPPL (Goodman and Stuhlmüller 2014) (see, e.g., Ritchie (2016)) and languages specifically motivated by such applications, namely Quicksand (Ritchie 2014) and Picture (Kulkarni et al. 2015). The latter are in some sense the most closely-related to Scenic, although neither provides specialized syntax or semantics for dealing with geometry or dynamic behaviors (Picture also was used only for inverse rendering, not data generation). The main advantage of Scenic over these languages is that its domain-specific design permits concise representation of complex scenarios and enables specialized sampling techniques.

Scenario description languages for autonomous driving. Recently, formal dynamic scenario description languages have been proposed for the domain of autonomous driving. The Paracosm language (Majumdar et al. 2019) is used to model dynamic scenarios with a reactive and synchronous model of computation. However, it is not a PPL, so it lacks probability distributions and declarative constraints; it also does not provide constructs like Scenic's interrupts which allow easy customization of generic behavior models. The Measurable Scenario Description Language (M-SDL) (Foretellix 2020) does provide declarative constraints, as well as compositional features similar to those we introduced in this paper. However, compared to both of these languages (which were introduced after the first version of this paper), Scenic has several distinguishing features: (1) it provides a much higher-level, declarative way of specifying geometric constraints; (2) it is fundamentally a probabilistic programming language (as opposed to M-SDL where distributions are optional), and (3) it is not specific to the autonomous driving domain (as demonstrated in Fremont et al. (2019, 2020)).

9 Conclusion

In this paper, we introduced Scenic, a probabilistic programming language for specifying distributions over configurations of physical objects and the behaviors of dynamic agents. We showed how Scenic can be used to generate synthetic data sets useful for a variety of tasks in the design of robust ML-based cyber-physical systems. Specifically, we used Scenic to generate specialized test sets and falsify a system, improve the robustness of a system by emphasizing difficult cases in its training set, and generalize from individual failure cases to broader scenarios suitable for retraining. In particular, by training on hard cases generated by Scenic, we were able to boost the performance of a car detector neural network (given a fixed training set size) significantly beyond what could be achieved by prior synthetic data generation methods (Johnson-Roberson et al. 2017) not based on PPLs.

In future work we plan to conduct experiments applying Scenic to a variety of additional domains, applications, and simulators. As we mentioned in the Introduction, we have already successfully applied Scenic to aircraft (Fremont et al. 2020), and we are currently investigating applications in further domains including underwater vehicles and indoor robots. We also plan to extend the Scenic language itself in several directions, including allowing user-defined specifiers and describing 3D scenes. Finally, we are exploring ways to combine Scenic with automated analyses: in particular, reducing the human burden of writing Scenic programs through algorithms for synthesizing or adapting such programs (e.g. Kim et al. (2020)), and improving the efficiency of falsification by performing white-box analyses of the system.

Acknowledgements The authors are grateful to Johnathan Chiu, Greg Crow, Francis Indaheng, Ellen Kalvan, Kevin Li, Matthew Rhea, Jay Shenoy, Kesav Viswanadha, and Wilson Wu for helping to develop and test Scenic libraries and simulator interfaces. The authors also thank Ankush Desai, Alastair Donaldson, Andrew Gordon, Steve Lemke, Shalin Mehta, Jonathan Ragan-Kelley, Sriram Rajamani, and several anonymous reviewers for helpful discussions and feedback. This work is supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400, NSF grants CNS-1545126 (VeHICaL), CNS-1646208, CNS-1739816, and CCF-1837132, by DARPA contracts FA8750-16-C0043 (Assured Autonomy) and FA8750-20-C-0156 (Symbiotic Design of Cyber-Physical Systems), by Berkeley Deep Drive, by Toyota through the iCyPhy center, and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Author Contributions DJF and SAS conceived of Scenic, scene/scenario improvisation, and the applications reported in this paper. DJF led the design of Scenic, guided by SAS and with input from the other authors. DJF implemented the language. SG and TD developed some of the pruning algorithms, and the library allowing Scenic to read the GTAV map. XY implemented the GTAV plugin to import Scenic scenes and capture the resulting data. TD and DJF designed the GTAV experiments and ran them with XY. EK designed and ran the CARLA experiments, and helped design and implement the library for dynamic driving scenarios. DJF, TD, XY, and SAS wrote the conference version of the paper (Fremont et al. 2019) in consultation with SG; the new material in this extended version was written by DJF and EK. SAS and ASV supervised the project and revised both versions of the paper.

Funding See Acknowledgements above.

Availability of data and materials Software and documentation for generating the datasets used in this paper can be found in the Scenic repository (see below).

Declarations

Conflict of interest DJF is a guest editor of the special issue in which this article appears; he was recused from all matters related to this paper. Otherwise, the authors have no conflicts of interest to declare that are relevant to the content of this article.

Code availability The Scenic implementation is open-source; its code, as well as code for all Scenic programs used in this paper, can be found at https://github.com/BerkeleyLearnVerify/Scenic.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P. F., Schulman, J., & Mané, D. (2016). Concrete problems in AI safety. CoRR. arXiv:abs/1606.06565

Arlot, S., & Celisse, A. (2010). A survey of cross-validation procedures for model selection. *Statistics Surveys, 4*, 40–79. https://doi.org/10.1214/09-SS054

Azad, A. S., Kim, E., Wu, Q., Lee, K., Stoica, I., Abbeel, P., Seshia, S. A. (2021). Scenic4rl: Programmatic modeling and generation of reinforcement learning environments. CoRR. arXiv:abs/2106.10365

Baidu. (2020). Apollo. https://apollo.auto/

Broy, M., Jonsson, B., Katoen, J. P., Leucker, M., & Pretschner, A. (2005). *Model-based testing of reactive systems: Advanced lectures (lecture notes in computer science)*. Springer.

Claret, G., Rajamani, S. K., Nori, A. V., Gordon, A. D., & Borgström, J. (2013). Bayesian inference using data flow analysis. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering* (pp. 92–102). ACM.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator. In *Conference on robot learning, CoRL* (pp. 1–16).

Dreossi, T., Donzé, A., & Seshia, S. A. (2017). Compositional falsification of cyber-physical systems with machine learning components. In *NASA formal methods, NFM* (pp. 357–372). https://doi.org/10.1007/978-3-319-57288-8_26

Dreossi, T., Fremont, D. J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., & Seshia, S. A. (2019). VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In I. Dillig, & S. Tasiran (Eds.), *Computer aided verification—31st international conference, CAV 2019, New York City, NY, USA, July 15–18, 2019, proceedings, part I, lecture notes in computer science* (Vol. 11561, pp. 432–442). Springer. https://doi.org/10.1007/978-3-030-25540-4_25

Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A. L., & Seshia, S. A. (2018). Counterexample-guided data augmentation. In J. Lang (Ed.), *Proceedings of the 27th international joint conference on artificial intelligence, IJCAI 2018, July 13–19, 2018, Stockholm, Sweden* (pp. 2071–2078). ijcai.org. https://doi.org/10.24963/ijcai.2018/286

Dreossi, T., Jha, S., & Seshia, S. A. (2018). Semantic adversarial deep learning. In *30th international conference on computer aided verification (CAV)*.

DuPont. (2012). Global automotive color popularity report. https://web.archive.org/web/20130818022236/http://www2.dupont.com/Media_Center/en_US/color_popularity/Images_2012/DuPont2012ColorPopularity.pdf

Elmas, T., Burnim, J., Necula, G., & Sen, K. (2013). CONCURRIT: A domain specific language for reproducing concurrency bugs. In: Proceedings of the 34th ACM SIGPLAN conference on programming language design and implementation, PLDI '13 (pp. 153–164). Association for Computing Machinery. https://doi.org/10.1145/2491956.2462162

Filipowicz, A., Liu, J., & Kornhauser, A. (2017). *Learning to recognize distance to stop signs using the virtual world of grand theft auto 5*. Tech. rep., Princeton University.

Fisher, M., Ritchie, D., Savva, M., Funkhouser, T., & Hanrahan, P. (2012). Example-based synthesis of 3d object arrangements. In *ACM SIGGRAPH 2012, SIGGRAPH Asia '12*.

Foretellix. (2020). Measurable scenario description language. https://www.foretellix.com/wp-content/uploads/2020/07/M-SDL_LRM_OS.pdf

Fremont, D., Yue, X., Dreossi, T., Ghosh, S., Sangiovanni-Vincentelli, A. L., & Seshia, S. A. (2018). Scenic: Language-based scene generation. Tech. Rep. UCB/EECS-2018-8, EECS Department, University of California. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-8.html

Fremont, D. J. (2019). Algorithmic improvisation. Ph.D. thesis, University of California. https://escholarship.org/uc/item/3812m6wx

Fremont, D. J., Chiu, J., Margineantu, D. D., Osipychev, D., & Seshia, S. A. (2020). Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI. In *32nd international conference on computer aided verification (CAV)*.

Fremont, D. J., Donzé, A., Seshia, S. A., & Wessel, D. (2015). Control improvisation. In *35th IARCS annual conference on foundation of software technology and theoretical computer science (FSTTCS), LIPIcs* (Vol. 45, pp. 463–474).

Fremont, D. J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., & Seshia, S. A. (2019). Scenic: A language for scenario specification and scene generation. In K. S. McKinley, & K. Fisher (Eds.), *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation (PLDI)* (pp. 63–78). ACM. https://doi.org/10.1145/3314221.3314633

Fremont, D. J., Kim, E., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A. L., & Seshia, S. A. (2020). Scenic: A language for scenario specification and data generation. https://arxiv.org/abs/2010.06580

Fremont, D. J., Kim, E., Pant, Y. V., Seshia, S. A., Acharya, A., Bruso, X., Wells, P., Lemke, S., Lu, Q., & Mehta, S. (2020). Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE intelligent transportation systems conference, ITSC 2020* (pp. 913–920). IEEE. arxiv:2003.07739

Geiger, A., Lenz, P., & Urtasun, R. (2012). Are we ready for autonomous driving? the Kitti vision benchmark suite. In *Computer vision and pattern recognition, CVPR* (pp. 3354–3361). https://doi.org/10.1109/CVPR.2012.6248074

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The language and its implementation*. Addison-Wesley.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems* (pp. 2672–2680).

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. CoRR. arXiv:1412.6572

Goodman, N., Mansinghka, V. K., Roy, D., Bonawitz, K., Tenenbaum, J. B. (2008). Church: A universal language for generative models. In *Uncertainty in artificial intelligence 24 (UAI)* (pp. 220–229).

Goodman, N. D., Stuhlmüller, A. (2014). The design and implementation of probabilistic programming languages. Retrieved July 11, 2018, from http://dippl.org

Gordon, A. D., Henzinger, T. A., Nori, A. V., & Rajamani, S. K. (2014). Probabilistic programming. In *FOSE 2014* (pp. 167–181). ACM.

Gupta, A., Vedaldi, A., & Zisserman, A. (2016). Synthetic data for text localisation in natural images. In *Computer vision and pattern recognition, CVPR* (pp. 2315–2324). https://doi.org/10.1109/CVPR.2016.254

Jaderberg, M., Simonyan, K., Vedaldi, A., & Zisserman, A. (2014). Synthetic data and artificial neural networks for natural scene text recognition. CoRR. arXiv:abs/1406.2227

Jiang, C., Qi, S., Zhu, Y., Huang, S., Lin, J., Yu, L. F., Terzopoulos, D., & Zhu, S. C. (2018). Configurable 3d scene synthesis and 2d image rendering with per-pixel ground truth using stochastic grammars. *International Journal of Computer Vision,* 1–22.

Johnson-Roberson, M., Barto, C., Mehta, R., Sridhar, S. N., Rosaen, K., & Vasudevan, R. (2017). Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? In *International conference on robotics and automation, ICRA* (pp. 746–753). https://doi.org/10.1109/ICRA.2017.7989092

Jung, A. (2018). imgaug. https://github.com/aleju/imgaug

Kim, E., Gopinath, D., Pasareanu, C. S., & Seshia, S. A. (2020). A programmatic and semantic approach to explaining and debugging neural network based object detectors. In *2020 IEEE/CVF conference on computer vision and pattern recognition, CVPR 2020* (pp. 11125–11134). IEEE. https://doi.org/10.1109/CVPR42600.2020.01114

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-Time Systems, 2*(4), 255–299.

Kulkarni, T., Kohli, P., Tenenbaum, J. B., & Mansinghka, V. K. (2015). Picture: A probabilistic programming language for scene perception. In *IEEE conference on computer vision and pattern recognition (CVPR)* (pp. 4390–4399).

Laminar Research. (2019). X-plane 11. https://www.x-plane.com/

Liang, X., Hu, Z., Zhang, H., Gan, C., & Xing, E. P. (2017). Recurrent topic-transition GAN for visual paragraph generation. ArXiv preprint. arXiv:1703.07022

Liebelt, J., & Schmid, C. (2010). Multi-view object class detection with a 3d geometric model. In *Computer vision and pattern recognition, CVPR* (pp. 1688–1695). https://doi.org/10.1109/CVPR.2010.5539836

Majumdar, R., Mathur, A. S., Pirron, M., Stegner, L., & Zufferey, D. (2019). Paracosm: A language and tool for testing autonomous driving systems. CoRR. arxiv:1902.01084

Marchesi, M. (2017). Megapixel size image creation using generative adversarial networks. ArXiv preprint (2017). arXiv:1706.00082

Michel, O. (2004). Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems, 1*(1), 39–42.

Milch, B., Marthi, B., & Russell, S. (2004). Blog: Relational modeling with unknown objects. In *ICML 2004 workshop on statistical relational learning and its connections to other fields* (pp. 67–73).

Moosavi-Dezfooli, S., Fawzi, A., & Frossard, P. (2016). Deepfool: A simple and accurate method to fool deep neural networks. In *Computer Vision and Pattern Recognition, CVPR* (pp. 2574–2582). https://doi.org/10.1109/CVPR.2016.282

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Gool, L. V. (2006). Procedural modeling of buildings. *ACM Transactions Graphics, 25*(3), 614–623. https://doi.org/10.1145/1141911.1141931

Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., & Shurek, G. (2006). Constraint-based random stimuli generation for hardware verification. In *Proc. of AAAI* (pp. 1720–1727).

Nguyen, A. M., Yosinski, J., & Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Computer vision and pattern recognition, CVPR* (pp. 427–436). https://doi.org/10.1109/CVPR.2015.7298640

Nori, A. V., Hur, C. K., Rajamani, S. K., & Samuel, S. (2014). R2: An efficient MCMC sampler for probabilistic programs. In *AAAI* (pp. 2476–2482).

Pei, K., Cao, Y., Yang, J., & Jana, S. (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *Symposium on operating systems principles, SOSP* (pp. 1–18). https://doi.org/10.1145/3132747.3132785

Ritchie, D. (2014). Quicksand: A lightweight embedding of probabilistic programming for procedural modeling and design. In *3rd NIPS workshop on probabilistic programming*. https://dritchie.github.io/pdf/qs.pdf

Ritchie, D. (2016). Probabilistic programming for procedural modeling and design. Ph.D. thesis, Stanford University. https://purl.stanford.edu/vh730bw6700

Rockstar Games. (2015). Grand theft auto v. Windows PC version. https://www.rockstargames.com/games/info/V

Rong, G., Shin, B. H., Tabatabaee, H., Lu, Q., Lemke, S., Možeiko, M., Boise, E., Uhm, G., Gerow, M., Mehta, S., Agafonov, E., Kim, T. H., Sterner, E., Ushiroda, K., Reyes, M., Zelenkovsky, D., Kim, S. (2020). LGSVL simulator: A high fidelity simulator for autonomous driving. arxiv:2005.03778

Ros, G., Sellart, L., Materzynska, J., Vázquez, D., & López, A. M. (2016). The SYNTHIA dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Computer vision and pattern recognition, CVPR* (pp. 3234–3243). https://doi.org/10.1109/CVPR.2016.352

Rubinstein, R. Y., & Kroese, D. P. (2004). *The cross-entropy method: A unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning*. Springer. https://doi.org/10.1007/978-1-4757-4321-0

Russell, S., Dietterich, T., Horvitz, E., Selman, B., Rossi, F., Hassabis, D., Legg, S., Suleyman, M., George, D., & Phoenix, S. (2015). Letter to the editor: Research priorities for robust and beneficial artificial intelligence: An open letter. *AI Magazine, 36*, 4.

Saheb-Djahromi, N. (1978). Probabilistic LCF. In *Mathematical foundations of computer science* (pp. 442–451). Springer.

Seshia, S. A., Sadigh, D., & Sastry, S. S. (2016). Towards verified artificial intelligence. ArXiv e-prints.

Stark, M., Goesele, M., & Schiele, B. (2010). Back to the future: Learning shape models from 3d CAD data. In *British machine vision conference, BMVC* (pp. 1–11). https://doi.org/10.5244/C.24.106

Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute force vulnerability discovery*. Addison-Wesley.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2014). Intriguing properties of neural networks. In *International conference on learning representations (ICLR)*.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., & Fergus, R. (2013). Intriguing properties of neural networks. CoRR. arxiv:1312.6199

Thorn, E., Kimmel, S., & Chaka, M. (2018). A framework for automated driving system testable cases and scenarios. Tech. Rep. DOT HS 812 623, National Highway Traffic Safety Administration. https://www.nhtsa.gov/sites/nhtsa.dot.gov/files/documents/13882-automateddrivingsystems_092618_v1a_tag.pdf

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *International conference on intelligent robots and systems, IROS* (pp. 23–30). https://doi.org/10.1109/IROS.2017.8202133

Vazquez, D., Lopez, A. M., Marin, J., Ponsa, D., & Geronimo, D. (2014). Virtual and real world adaptation for pedestrian detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 36*(4), 797–809.

Wong, S. C., Gatt, A., Stamatescu, V., & McDonnell, M. D. (2016). Understanding data augmentation for classification: when to warp? In *Digital image computing: Techniques and applications (DICTA), 2016 international conference on* (pp. 1–6). IEEE.

Wood, F., Meent, J. W., & Mansinghka, V. (2014). A new approach to probabilistic programming inference. In *Artificial intelligence and statistics* (pp. 1024–1032).

Wu, B., Iandola, F. N., Jin, P. H., & Keutzer, K. (2017). Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Conference on computer vision and pattern recognition workshops, CVPR workshops* (pp. 446–454). https://doi.org/10.1109/CVPRW.2017.60

Xu, Y., Jia, R., Mou, L., Li, G., Chen, Y., Lu, Y., & Jin, Z. (2016). Improved relation classification by deep recurrent neural networks with data augmentation. ArXiv preprint. arXiv:1601.03651