ScenicGym: Reinforcement Learning with Data Generation Using Scenic

Kai Xu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-168
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-168.html

August 15, 2025

Acknowledgement

I am grateful of my advisor Prof. Sanjit Seshia. Special thanks to Prof. Shankar Sastry, who is the second reader, and also to Prof. Daniel Fremont. Thanks to Eddie Kim, and to the Learn and Verify Group. Special thanks to my family for their support all along.

ScenicGym: Reinforcement Learning with Data Generation Using Scenic

by

Kaifei Xu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Shankar Sastry

Summer 2025

The dissertation of Kaifei Xu, titled ScenicGym: Reinforcement Learning with Data Generation Using Scenic, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

ScenicGym: Reinforcement Learning with Data Generation Using Scenic

Abstract

ScenicGym: Reinforcement Learning with Data Generation Using Scenic

by

Kaifei Xu

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

There has been significant recent interest in using reinforcement learning for control in cyber-physical-systems (CPS). Domains affected include autonomous driving, robotics, and drone control. Much of these applications should be considered as safety-critical, where system failure can cause significant damage and injury to humans. Even in non-safety critical applications, such system failures could also be expensive. It is therefore important to be able to add assurance to the process of reinforcement learning, which is a challenge due to the statistical nature of modern learning algorithms. In this thesis, we present ScenicGym, which is an RL training tool based on the probabilistic programming language Scenic and its related toolkit VerifAI. The new tools allow RL researchers to train agents completely using data generated by concise Scenic programs with sampling conducted by VerifAI to incorporate edge cases. We demonstrate the use of ScenicGym and the influence of incorporating VerifAI's sampler with experiments in autonomous driving.

I once was lost.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express deep gratitude to my advisor Prof. Sanjit Seshia. While I was a second-semester junior who have just decided to go into research in EECS, he was the one who took a chance on me. I have grown tremendously under his mentorship, learning to be a better researcher, engineer, communicator, and part of a team. He led me into an exciting research field that I knew little about before and has always been a true role model of heartfelt scientific passion and a source of sincere guidance. It was also through his unwaivering support that I have the opportunity to become a master student and now further my studies as a PhD student. I am thus indebted to him for opening for me a colorful door to a brilliant new world.

I would also like to thank Prof. Shankar Sastry, who is also a reader of this work. His teaching gave me the foundational knowledge in robotics that powered much of my research, and his insights and constant excitement and joy during our conversations serve as powerful inspirations to me, both as a researcher and as a person. I would also like to thank him for the great support that allowed me to continue my studies here in the coming years.

I am also grateful for Prof. Daniel Fremont, who has mentored me throughout my time here in the Scenic project and is also a driving force that allowed me to pursue graduate studies. He has imparted me much wisdom during our weekly Scenic meetings, covering items from engineering, research directions, new ideas, and more. He also has always been generous with his time in guiding me through the many obstacles I encountered during development, and research would have been impossible without his patience and advice.

Eddie Kim has been my post-doc mentor ever since I arrived at the lab. I have learned from him a great deal about the practice of research, the value of collaboration and reaching out to people, and being a very good guy who is caring of others. I am glad that I have gotten to work with him, and I will always value the lessons he has taught me.

I want to thank all the members of the Learn and Verify Group for all the good times. To my fellow masters at the table, Alex and Anirudh, it has been a true blast. Special thanks to Aniruddha, Kai-Chun, and Beyazit for the laughter at all the events and the Scenic workshop. Kai-Chun led the first paper I was ever a part of, and this experience I will always cherish. I want to thank Victoria for the conversations exchanging thoughts on life and Gazebo, and a big thanks to Adwait for the fun times setting up the NAS on LAV.

To mom and dad, thank you for believing in me all this way, for encouraging me throughout my life wherever I am, and for teaching me to have love and faith. You are the best parents one can dream of. To Snow and Mocha, you two brighten up my days and nights, and growing up with you all these years has given me a heart I will not lose. I love you all.

Chapter 1

Introduction

The application of reinforcement learning (RL) to cyber-physical systems (CPS) necessitates strong safety guarantees. Generating such assurance usually falls under the realm of formal methods, but challenges arise. While RL techniques currently seem promising for domains including autonomous driving and robotics, assurance of any kind of learning-based systems is difficult due to the high uncertainty and variability of real-world environments combined with the statistical of modern learning algorithms. Application of techniques such as model checking that proves correctness is therefore a challenge.

Overall, the correctness guarantee of RL-based CPS (RL-CPS) aligns with the the general effort towards verified artificial intelligence [16], where the societal-scale deployment of AI software requires novel techniques for verification. In this context, developing robust RL-CPS with correctness guarantees presents a multi-faceted problem. First, there exists the question of specification. In the context of RL, the train-time specification is generally encapsulated by the reward function, the design of which is an art of its own. Modeling is the another issue. A properly abstracted model of both the system and its operating environment are necessary for a useful verification result, but finding the correct level of abstraction is tricky. A detailed model entailing the great complexity of the system can make the verification procedure intractable; in contrast, a model too simple yields useless guarantees. In the context of CPS, the uncertainty and unknown variables in the real-world environment also need to be modeled. This aspects raises the question as to how one would model the environmental uncertainty in a realistic way that allows for rigorous testing of the system without generating spurious counterexamples. Machine learning components brings in additional complexity. For instance, RL policies generally have a high dimensional feature space for inputs such as images and LiDAR sensor readings. Traversing these high dimensional spaces for the purposes of verification is not practical, and our verification model should allow us to circumvent examining system inputs at the level of raw sensor data. In summary, a linchpin to providing guarantees of robustness to learning-based CPS is to generate a compact and efficient model composed of the learning agent, the environment, their interaction, and their overall dynamics and evolution. The model should allow us to efficiently verify if the our learning-based CPS satisfy our specification, and in the event that complete verification is

difficult, the model should be usable for discovering meaningful counterexamples. For the purposes of reinforcement learning, there is the additional requirement that we should be able to use the model and the verification process to train the agent to correct its errors.

Simulation-based verification is a promising direction. This approach takes advantage of the recent advances in simulation technologies that allow us to simulate a CPS and its deployment environments at varying degrees of realism and efficiency. Simulation is also the predominant way by which autonomous CPS are developed, allowing for testing designs without the costly investment of physical implementation. Inside the simulation, the environmental and internal system dynamics and evolution are generally well-modeled by modern physics engine. Developments in modern graphics makes camera sensor inputs similar to real-world images, and new hardware and software implementations allows the simulation to be run at high efficiency and parallelism. The challenges left to be tackled is to efficiently model and represent the input space of the learning-based CPS, to develop the procedure by which the verification is conducted, and to invent methods by which we train the agent to master the discovered counterexample scenarios. In this thesis, we develop new tools that take advantage of the existing tool kits Scenic and VerifAI [6][21][4]to provide an integrated workflow that combines all three of the above points for the robust training of RL-CPS.

1.1 Background

Reinforcement Learning

Reinforcement learning differs from supervised learning schemes due to its problem setup where the agent learns the correct behavior from interacting with its operating environment rather than a set of demonstrations. The formal model of reinforcement learning is based on the Markov decision process (MDP)[17]. Formally, an MDP is a random process defined by the tuple (S, A, T, r), where S is the state space, A is the set of available actions to the agent, T is a function $S \times A \to \Delta S$ that defines the transition between state based on the action that was chosen. ΔS denotes the set of probability distributions over S. The function T yields distributions that are Markovian: the probability of the future state is dependent on past states and actions only through the present state and action. r is the reward function $S \times A \to \mathbb{R}$. To solve the RL problem is to learn a policy $\pi_\theta(a|s)$, which is a distribution, parameterized by θ, over the possible actions to take in the current timestep conditioned on the current state s. The typical goal is for the policy to maximize the expected long term reward, the exact definition of which differs among cases.

Scenic + VerifAI[6][21][4]

Moving towards verifying machine learning-based CPS, two particularly useful tools that have been developed are Scenic and VerifAI. Scenic is a probabilistic programming language that tackles the modeling problem in the verification process. Scenic is interfaced to a range

of simulators, and any controller and simulated model of a CPS that can be run inside the simulators can be run alongside Scenic. Scenic specifies environmental models that captures variation of the real-world operating conditions of the CPS, the hard and soft (probabilistic) constraints that the composed agent-environment model should satisfy in the form of logical properties, and the high-level interaction between the system and its environment. It thus provides the environmental model and defines the composition of the environmental model and the system model.

Scenic programs further offer a compact representation of the *semantic feature space* for the learning-based agents. To elaborate, the high-dimensional input space to learning-based systems are usually hopeless to specify and model directly. For instance, it is difficult to describe the kinds of images a vision-based controller will see on a pixel-by-pixel level. What is more practical is to describe the kinds scenarios (the semantic aspect) that will induce an input to our CPS. For instance, generating a specific configuration of cars and pedestrians on the street in simulation will cause a camera to see a particular image. Thus, the range of images the sensor sees is directly determined by the range of scenarios the simulation environment generates, and the latter is much more describable and analyzable. In Scenic, the semantic features are the parameters sampled from a Scenic program, such as positions of the cars, speed of the pedestrians walking, or the end effector position of a robot. Overall, Scenic can generate test scenarios for the CPS and monitor the evolution of the combined closed-loop system-environment model. In the context of training learning-based CPS, Scenic provides a framework that models a CPS' operating environment, and from this model, data can be generated, including data of rare events that are difficult to obtain from the real world.

VerifAI is a closely related tool that takes advantage of Scenic. VerifAI performs the function of the verifier and analyzer in the verification framework. Given a specification, a model of the cyber-physical system (controller, perception component, and the plant model), and a model of the system's operating environment, VerifAI conducts tasks including fuzz testing, falsification, counterexample analysis, data augmentation, hyperparameter synthesis, and model parameter synthesis. In more detail, VerifAI accepts a specification, which can be in the form of metric temporal logic (MTL) or a generic reward/objective function. A Scenic program is provided that specifies a simulator in which to test the target CPS. This Scenic program serves to define the environmental model, variation in the environmental condition, and the agent-environment interaction. It also defines the semantic feature space, which is the main analysis target of VerifAI and usually takes the form of distributions specified in the Scenic program. The specified simulator will run the controller, the plant model, and the perception modules; the simulator also defines the evolution dynamics of the simulation world via its physics engine. While operating, VerifAI samples environmental parameters from the semantic feature space and monitors each simulation episode for violation of the specification. VerifAI collects the violation results from past episodes into an error table, on which the toolkit performs analysis to discern which features in the semantic feature space are most correlated with the violation of the specification. VerifAI possesses a suite of samplers for the semantic feature space. Some are passive, where the sampler does not take

any inputs nor consider past simulation histories when sampling parameters for an episode. Passive samplers include the likes of uniform random sampling, simulated-annealing, and Halton sampling, which have shown to perform well in falsification. The other samplers, such as the multi-arm bandit, cross-entropy, and Bayesian optimization samplers, are active. The analysis result from the error table analysis, usually a number, is fed back into these active samplers prior to the next episode and informs future sampling. Generally, based on the feedback, the active samplers attempt to sample for episodes that results in stronger violations of the specification. Overall, Scenic and VerifAI combines to tackle the complex modeling, analysis, and verification challenges that arise in developing assured AI-enabled CPS.

1.2 Motivations and Contributions

Training RL-based CPS comes with inherent challenges. Among those is the problem of dataset construction. Modern RL dataset, such as the Habitat Synthetic Scenes Dataset (HSSD)[10] for indoor robotics and NuScenes for autonomous driving[3], are often hand-curated. HSSD is hand-made by artists, and NuScenes is derived from real-world traffic scenarios. The disadvantage of this approach is clear: creating such datasets is labor-intensive, and the resulting datasets are often bulky. For certain new domains, such as drone navigation, where such datasets are still lacking, researchers might have to wait a long time before obtaining a datasets that allows them to make progress. Furthermore, datasets are often not portable across simulators. This issue impedes cross-simulator training methods, where one may wish to pretrain a policy on a fast, low-fidelity simulator, and fine-tune on a slower but more realistic simulator. Data interpretability and case coverage are also challenges. It is generally hard to reason about the distribution of scenarios inside a large datasets: what kinds of scenarios are included, and how much of each kind is in the dataset? It is further difficult to have any confidence that the dataset covers important edge cases under which the agent may fail to perform correctly.

Scenic and VerifAI can help alleviate these problems. Scenic programs serve as natural means of describing and generating RL training episodes. Being simulator agnostic, Scenic programs can be ported across any simulators to which Scenic has an interface, allowing for cross-simulator training. During training time, VerifAI can analyze regions in the semantic feature space that are likely to result in low reward/bad behavior on the agent's part. VerifAI's samplers will then generate training episodes with parameters sampled from those regions. This aspect of training with Scenic and VerifAI assures case-coverage within the bounds of the types of scenarios specified by the Scenic program. We do note that this scheme could be somewhat naive from an assurance perspective since we are assuming that the reward function is a good specification of the desired behavior of an agent. This assumption is often untrue, as we have seen in cases of reward-hacking among RL agents. However, given a reward function derived from a good specification, the training process described above should help reduce incorrect behaviors in deployment. Within the scope of

this thesis, we assume in our experiments that the reward functions specify the appropriate system behaviors.

Scenic and VerifAI has been put to great use in complete training of RL agents in prior works [1], where the toolkit was used to effectively train a soccer-playing agent in the Google Soccer training environment. Scenic and VerifAI has also been used before to robustify learning-based CPS in edge cases through data augmentation by generating counterexample scenarios where the system violates specification[5][7]. These work left for exploration the from-scratch training of RL agent using Scenic/VerifAI. What also needs to be developed is a RL training tool that is integrated with Scenic and VerifAI that can be used with all modern RL framework, thus combining training, modeling, error analysis, and verification into a single workflow. To this end, some technical obstacles also need to be addressed. In its current form, Scenic is incompatible with existing RL API's and libraries, and incorporating its use into existing RL code base is difficult. This issue is due to the API not allowing the user to control stepping of the simulation or relay information such as reward and observation to any external RL algorithms. The rigid logic of the current Scenic backend also requires modification of the interface between Scenic and the simulator to incorporate the RL algorithms. Therefore, Scenic needs a new API that allows it to be used in standard RL workflows.

In this work, we developed the tools ScenicGym, which is a set of API wrappers around Scenic that inherits from OpenAI Gym/Farama Gymnasium[2][20] and PettingZoo[18] for both single-agent and multi-agent RL training. This direction allows the user to use Scenic and VerifAI training with the API popularized by Open AI Gym, where we have the standard methods such as `step` and `reset` which controls the training episodes and returns information about the training such as reward and observation. We then demonstrate using the new API on some simple RL tasks.

Chapter 2

Modeling RL Episodes with Scenic

In this chapter, we introduce how to use Scenic[7][21] to model single and multi-agent RL scenarios. Scenic programs allow modeling of RL episodes in a way that is concise, modular, and compositional. This approach is advantageous in a number of use cases since users can model complex behaviors of the environment (which would include non-learning agents) that obey spatio-temporal constraints. This plays important roles in both single and multi-agent training.

2.1 Definitions

We will illustrate components of the Scenic program corresponding to the MDP formulation of RL. Recall that an MDP can be thought to consists of the tuple (S, A, p, r, ρ_0), where S is the set of states, A is the actions available to the agent, p is is Markovian conditional probability of future states dependent on the current state and actions. r is the reward function, and ρ_0 is the distribution of the initial states.

2.2 Modeling Initial Condition Distributions

We shall use the program in Figure 2.1 to illustrate using Scenic to model an RL episode. The figure illustrates a robot navigation task. There is one learning agent `spot`, and one agent that is part of the environment `fetch`. The goal of `spot` is to reach its goal point without colliding with `fetch`. There are parameters sampled by the initial distribution, corresponding to the x-position of `spot` and the x and y position of `fetch`. These three distributions together corresponds define ρ_0 in our MDP. In general, any distribution specified in the Scenic program contribute to defining the initial condition. We would like to make a few notes about the Scenic program. We declared `spot` to be a learning agent, as shown by the true settting of `is_learning_agent` field. This means that `spot` is the agent that we are training, whereas `fetch` is just a part of the environment. In the backend, Scenic then marks `spot` as an agent whose observation and rewards we keep track.

```
model scenic.simulators.habitat.model
from scenic.simulators.habitat.actions import *
from scenic.simulators.habitat.behaviors import *
from scenic.simulators.habitat.model import *


goal_point = (-3.2, -1.0, 0)
goal_region = CircularRegion(goal_point, 0.7)

param verifaiSamplerType = 'halton'
param spot_y = Range(-6.5, -4.0)
param fetch_x = VerifaiRange(-5.5, -4.6)
param fetch_y = VerifaiRange(-3.8, -2.0)

spot = new SpotRobot at (-3.2, globalParameters.spot_y, 0),
                        with yaw 90 deg, with is_learning_agent True
fetch = new FetchRobot at (globalParameters.fetch_x, globalParameters.fetch_y, 0),
                        with yaw 0 deg, with behavior Traverse(1.5, 0, 0)

monitor Reward():
    done = False
    last_distance_from_goal = (goal_point - spot.position).norm()

    while True:
        if done:
            terminate
        spot.reward = 0

        distance_from_goal = (distance from spot to goal_point)
        d_distance = distance_from_goal - last_distance_from_goal
        spot.reward += -d_distance

        if (spot intersects goal_region):
            spot.reward += 10
            done = True

        if (spot intersects fetch):
            spot.reward -= 10
            done = True

        last_distance_from_goal = distance_from_goal
        wait

require monitor Reward()
```

Figure 2.1: An Example Training Program

VerifAI[4] Sampling for Initial Conditions

To take advantage of VerifAI, we make some changes to the program shown in Figure 2.1. As shown in Figure 2.2, we first specify a new Scenic `param` called `verifaiSamplerType`. This determines the sampling technique used by VerifAI, which in this case is Halton sampling. `Range` in the original program are not replaced by `VerifaiRange`, which declares these

variables and their distributions to be analyzed VerifAI. Note that distributions not changed to `VerifaiRange`, such as the parameter `spot_y` are still the original naive continuous uniform distribution, and VerifAI does not influence those values. It is worth noting that despite its name, `VerifaiRange` would in general not be a continuous uniform distribution due to the underlying sampling algorithm.

During training, the active VerifAI sampler will accept a feedback value. The user is free to define the feedback value to pass into the VerifAI sampler. For reinforcement learning, this value can be the episode cumulative reward or the cumulative discounted reward. VerifAI analyzes the history of this feedback in past episodes to determine which parameter is most responsible for changes in the feedback value. VerifAI also determines regions in the specified distributions that are likely to yield a low feedback and samples from these regions in the next episode. Qualitatively, this process generates difficult episodes for the agent. From the perspective of theory, this behavior breaks the independence of the initial conditions across different training episodes, which is usually not seen in standard RL training. The effects of this setup can be a topic for a future investigation.

```
param verifaiSamplerType = 'halton'
param spot_y = Range(-6.5, -4.0)
param fetch_x = VerifaiRange(-5.5, -4.6)
param fetch_y = VerifaiRange(-3.8, -2.0)

spot = new SpotRobot at (-3.2, globalParameters.spot_y, 0),
                        with yaw 90 deg, with is_learning_agent True
fetch = new FetchRobot at (globalParameters.fetch_x, globalParameters.fetch_y, 0),
                        with yaw 0 deg, with behavior Traverse(1.5, 0, 0)
```

Figure 2.2: Modification to the Scenic Program in Figure 2.1 to Use VerifAI

2.3 Environment Dynamics and Agent-Environment Interactions

Returning to the Scenic program in Figure 2.1, we note that the environment transition dynamics is partly defined by behaviors in the environment objects. Notice that `fetch` possesses a Scenic `behavior` called `Traverse`, which specifies how an actor behaves as part of the training environment. In our case, `fetch` purely traverses back and forth along a straight line, but Scenic `behaviors` are allowed to be interactive. For instance, we can define a behavior for `fetch` where it moves at different trajectories based on its distance to `spot`. Other factors outside of Scenic, such as simulator physics, still contribute to the environment dynamics as usual.

2.4 Modeling Rewards

There is some freedom in ways to model the reward. We demonstrate the current recommended way with the Scenic `monitor` construct. The `monitor` is not attached to any single agent. Rather, it is an object of the episode itself. In Figure 2.1, the `while True` loop keeps the `monitor` running throughout the episode, with the required `wait` keyword at the bottom prompting the loop body to be executed once per simulation timestep. In the context of RL, a `monitor` has several advantages. First, it possesses a global view of the training episode. A `monitor` can access any information inside the simulation, such as agent positions and object velocities. The `monitor` allows the user to identify the spatio-temporal relationship at each timestep using any of the Scenic operators, which provide convenience in considering the state of the episode. For instance, we used the `distance from` operator to identify the distance between two agents in 2.1. We also use the `intersects` operator to see if our agent has entered its goal zone, defined by `goal_region` early in the program using the `CircularRegion` class provided by Scenic. This allow us to define a circular area with the goal point as the center as the region we want the robot to enter.

In computing the reward, we allow the `spot`agent to possess a `reward` field, which can be modified by the `monitor`. This reward is eventually yielded by ScenicGym to the higher-level RL algorithm when stepping the RL environment, which we will see in the next chapter. We note that once a goal state is reached, as characterized by `if` blocks setting the variable `done` to be true, we can use the Scenic `terminate` keyword to end the episode. For instance, in the case of collisions or reaching a destination region, characterized by statements involving the keyword `intersects` to be true, we give `spot`an additional reward/penalty and set `done` to be true. Note that during training, if we set the episode length to be a particular number of time steps, the episode will terminate once that time limit is reached (despite us not explicitly stating this in the Scenic program). We will see how to set this timestep in the next chapter.

Using a `monitor` is also helpful in multi-agent scenarios. In Figure 2.3, and 2.4 we have two learning agents, `ego` and `car`. Those two vehicles arrive at an intersection and seek to cross without collision. In the `monitor` of this program, we are able to, with concise syntax, explicitly calculate the spatial relationship between the cars and assign reward and terminations from a global view rather than having to program the reward computation for each agent. This program also demonstrates the use of the `record` statement at the very bottom, where we can record any values computable from the Scenic program for post-episode analysis. In this case, we are recording our agents' total episode return for computing the VerifAI sampler feedback, which we will demonstrate in the next chapter. The `final` keyword means we are only recording the value of `episode_return` at the end of the episode. There is a corresponding `initial` keyword. If there is no such temporal keyword, then we will record the value at every single timestep.

```
from math import pi
param map = localPath(root_user + '/ScenicGymClean/assets/maps/CARLA/Town04.xodr')
param carla_map = 'Town04'
param time_step = 1.0/10
param camera_position = (311, 255, 0)
model scenic.domains.driving.model

success_reward = 10.0
driving_reward = 1.0
crash_penalty = -5.0
out_of_road_penalty = -5.0
speed_reward = 0.1

ego_dir = 180
car_dir = -90
ego_x = 312
car_y = 247

param ego_y = Range(255, 265)
param car_x = Range(290, 306)

ego = new Car on (ego_x, globalParameters.ego_y, 0), facing ego_dir deg,
                    with name "agent0",
                    with goal (311, 235, 0),

car = new Car on (globalParameters.car_x, car_y, 0), facing car_dir deg,
                    with name "agent1",
                    with goal (320, 246, 0),

monitor Reward():
    done = False
    ego_success = False
    car_success = False

    last_long_1 = ego.position[1]
    last_long_2 = car.position[0]

    drive_dir_1 = ego_dir * pi/180
    drive_dir_2 = car_dir * pi/180

    while True:

        ego.zero_reward()
        car.zero_reward()

        current_long_1 = ego.position[1]
        current_long_2 = car.position[0]

        ego.add_reward(driving_reward * (-1) * (current_long_1 - last_long_1))
        car.add_reward(driving_reward * (current_long_2 - last_long_2))

        last_long_1 = current_long_1
        last_long_2 = current_long_2

        ego.add_reward(speed_reward * ego.speed_km_h/ego.max_speed_km_h)
        car.add_reward(speed_reward * car.speed_km_h/car.max_speed_km_h)

```

Figure 2.3: A 2-Car Intersection Crossing Scene

```
        angle_rescale = lambda angle: angle + 2 * pi if angle < 0 else angle

        ego.add_reward(-abs(angle_rescale(ego.yaw) - angle_rescale(drive_dir_1))/pi * 0.1)
        car.add_reward(-abs(angle_rescale(car.yaw) - angle_rescale(drive_dir_2))/pi * 0.1)

        if (distance from ego to ego.goal) < 1:
            ego_success = True
            ego.add_reward(success_reward)

        if (distance from car to car.goal) < 1:
            car_success = True
            car.add_reward(success_reward)

        if ego_success and car_success:
            done = True

        # can do this since the cars can only crash with each other if they do crash
        if ego.metaDriveActor.crash_vehicle or car.metaDriveActor.crash_vehicle:
            ego.add_reward(crash_penalty)
            car.add_reward(crash_penalty)
            done = True

        ego.episode_return += ego.reward
        car.episode_return += car.reward

        if done:
            terminate
        wait


require monitor Reward()
record final ego.episode_return as agent0_return
record final ego.episode_return as agent1_return
```

Figure 2.4: Figure 2.3 cont.

2.5 Interfacing to New RL Environments and Simulators

Scenic currently has interfaces to a number of learning-oriented simulators. Interfacing Scenic to a new simulator is a straightforward process. We refer the reader to the Scenic repository and documentation for details.

Chapter 3

ScenicGym

We hereby discuss the background design and implementation of ScenicGym. In order to make the Scenic/VerifAI[6][21][4] suite a viable and convenient tool for the RL practitioners and thus merging modeling, verification, error/counterexample analysis, and training into one tool, a new API to Scenic is necessary. The prior API requires making significant and tedious changes at the simulator-interface level of Scenic each time a developer wishes to try a different algorithms. This issue has in the past made conducting RL experiments with Scenic a task that can only be tackled by Scenic veterans. A new API ought to address this issue. Specifically, the API should be compatible with modern RL libraries (such as Stable Baselines 3 and RLLib [14][12]), and it should be easily incorporable into any existing RL code base. Creating a Farama Gymnasium/OpenAI Gym-like API[2][20] is a natural thought, since the gym-style API has become industry standard. Certain design and implementation choices and issues needed to be addressed. In this section we illustrate the design of ScenicGym.

3.1 Design

ScenicGym consists of RL environments that inherit from the Farama Gymnasium environments, which has become an staple in the field. Corresponding wrappers for the OpenAI Gym as well as PettingZoo[18] for multi-agent training are also available. The use of the tool is shown in the following code block that was used to train an agent in Habitat using the Scenic program in Figure 2.1 with Stable Baselines 3. As shown in the `scenic_env` function on line 27, to instantiate a ScenicGym environment for training, the user specify a Scenic program that models the training episodes (line 33), compile the program into a Scenic scenario object, which serves as one of the environment's `__init__` arguments. To specify the simulator to use, a `Scenic` simulator instance (`HabitatSimulator` on line 41) is instantiated and also passed into the environment instantiation. Besides some standard keyword argument such as action and observation spaces, the user also pass in a function `feedback_fn` that processes the simulation and training episode results to generate a feedback value for the VerifAI samplers. In the vanilla case, this function can be one that computes the episode

cumulative return. We were then able to create a vectorized environment on line 56. Note that to make this vector environment with Scenic, the wrapper on line 47 is necessary due to reasons we will discuss when talking about running parallel training environments.

```
1  import gymnasium as gym
2  from scenic.gym import ScenicGymEnv
3  from pprint import pprint
4  import os
5  import numpy as np
6  import scenic
7  from scenic.simulators.habitat import HabitatSimulator
8  import datetime
9  from stable_baselines3.common.vec_env import SubprocVecEnv
10 from stable_baselines3 import PPO
11 from stable_baselines3.common.callbacks import (EvalCallback,
12                                                  ProgressBarCallback,
13                                                  CallbackList,
14                                                  CheckpointCallback)
15 import datetime
16 import argparse
17
18 parser = argparse.ArgumentParser()
19
20 parser.add_argument('-m', '--model', type=str)
21 parser.add_argument('-r', '--resume', action="store_true")
22 args = parser.parse_args()
23
24 def max_cum_reward(result):
25     return result.records["agent_return"]
26
27 def scenic_env():
28
29     root_user = os.path.expanduser("~")
30     obs_space = gym.spaces.Box(0, 255, (256, 256, 3), np.uint8)
31     action_space = gym.spaces.Box(-2.0, 2.0, (2,), np.float32)
32
33     scenic_file = "train_scenes/train.scenic"
34
35
36     scenario = scenic.scenarioFromFile(scenic_file,
37                                        model="scenic.simulators.habitat.model",
38                                    mode2D=False)
```

```
39
40      env = ScenicGymEnv(scenario,
41                         HabitatSimulator(),
42                         render_mode=None,
43                         max_steps=150,
44                         observation_space=obs_space,
45                         action_space=action_space
46                         feedback_fn=max_cum_reward)
47      env = gym.wrappers.RecordEpisodeStatistics(env)
48
49      return env
50
51  if __name__ == "__main__":
52      now = datetime.datetime.now()
53      now = now.strftime("%m_%d_%H_%M")
54      model_folder_name = f"habitat_nav_{now}"
55
56      env = SubprocVecEnv([scenic_env for _ in range(6)])
57      eval_env = scenic_env()
58
59      eval_callback = EvalCallback(eval_env,
60      best_model_save_path=f"./sb_models/{model_folder_name}",
61                                log_path=f"./sb_models/{model_folder_name}",
62                                eval_freq=1000,
63                                deterministic=True,
64                                render=False)
65
66      model = PPO("CnnPolicy", env, verbose=2)
67      model.learn(total_timesteps=100_000, callback=eval_callback, progress_bar=True)
68      model.save(f"habitat_nav_{now}")
```

Backend Design

The interaction between ScenicGym, Scenic, VerifAI, and the underlying simulators are shown in 3.1. At the start of the episode, Scenic samples for a new scene using a specified VerifAI sampler. The information about the new scene is then sent to the simulator via the Scenic-simulator interface. The scene is spawned in the simulator, and the episode begins. At each timestep, the ScenicGym environment computes the observation, reward, termination, and other information, and relays it back to the outer training loop. At the end of each episode, ScenicGym internally computes a feedback value for the VerifAI sampler based on the user's settings, and the next episode is generated. One new feature of Scenic that allowed

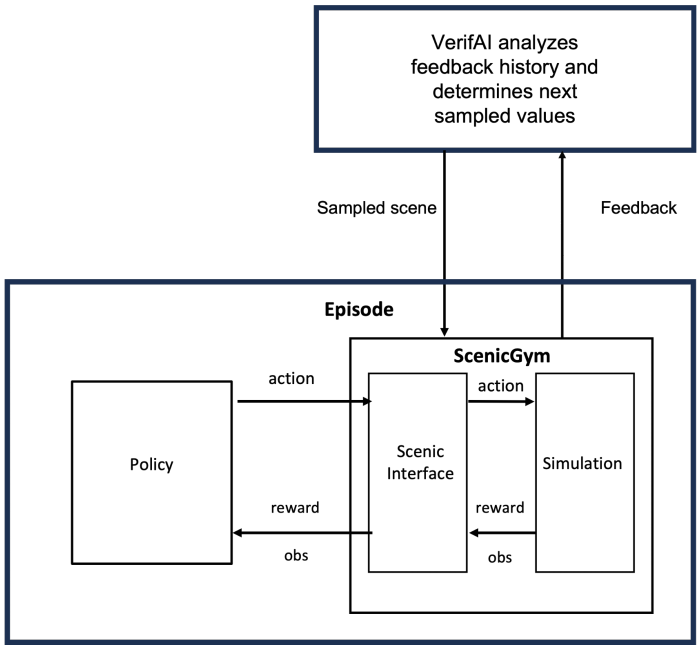


Figure 3.1: Overall Design of ScenicGym

for this construction is the ability for sequential stepping, where Scenic steps the simulation only when the user calls an `advance` function.

3.2 On Running Parallel Training Environments

ScenicGymallow running parallel training environments. The current requirement is that the parallelism has to be multi-process rather than mere multi-thread. Supporting multi-thread training is an ongoing work. This was why the wrapper on line 47 in the code block above was necessary, as it forces Gymnasium to use true multi-processing rather than multi-threading. Please see the code repository for more instructions and examples on configuring this for some common RL APIs.

3.3 Using RL Algorithm Libraries

The gym-like interface of ScenicGym allows compatibility with existing RL libraries. For instance, the code block shown above was using Stable Baselines 3, and training conducted for in the next section has been done using Ray. Please see the code repository for more instructions and examples.

Chapter 4

Experiments

We describe experimental results in this section that illustrate training with ScenicGym and influence of VerifAI[4] sampling.

4.1 Setup

The training Scenic program is shown in 2.3. The experiment is a multi-agent autonomous driving task where two cars arrive at an intersection seeking to cross without collision. The two cars are controlled by the same policy, so the training scenario is self-play. The policy takes in a lidar sensor observation and outputs a vector setting the acceleration and steering angle. The reward consists of incentives for speed, a straight steering, and reaching the goal point. There is a penalty for collision. We conducted our experiments inside the MetaDrive simulator [11] with Scenic. Training was done using PPO [15] inside RLLib of Ray using Ray's default PPO policy and training setting. The feedback we choose to give to the VerifAI active samplers is the maximum episode return among the two agents, which is the value that VeriAI's active samplers seek to minimize.

4.2 Training Methods

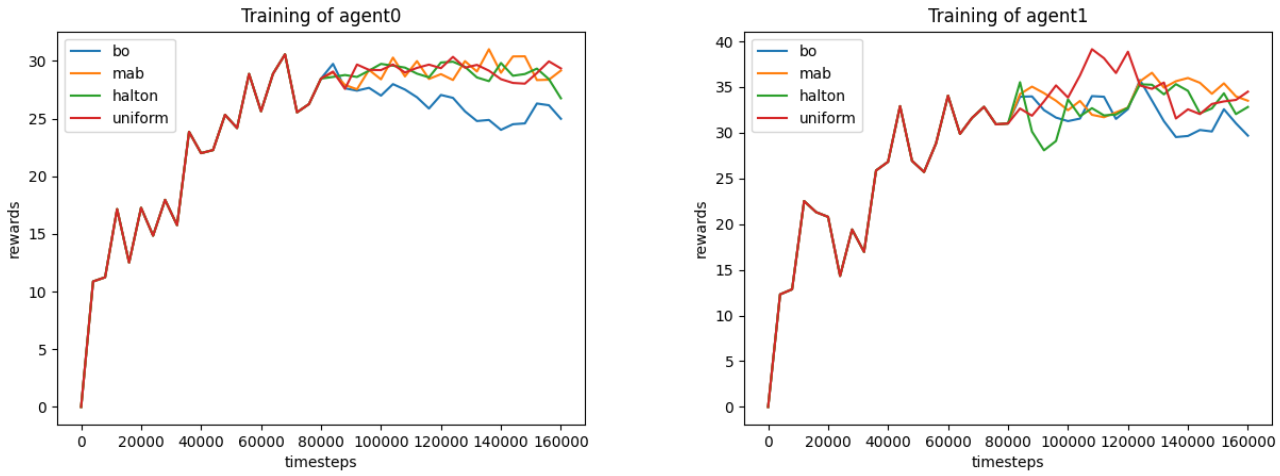
We first pre-trained a policy with Scenic using the standard uniform sampler (no active sampling). This is because during early stages in the training, the policy is likely to have poor performance across the distribution. An active sampler would then sample the scenes mostly from the small sections of the distributions that it encountered early on, reducing the variety of scenes the agent would experience, hindering learning. The pretraining was conducted for 80,000 timesteps.

We proceeded to conduct four experiments, each with a different VerifAI sampler, and examine the effects on the learning outcomes. The samplers examined are Bayesian optimization (BO), multi-armed bandit (MAB), and Halton[9]. We also trained a model continue using the uniform sampler as a baseline. Each experiment started with the pretrained model

and proceeded 80,000 timesteps. For each sampler, 5 trainings were conducted, each using a different seed. Any evaluation result is the average over the 5 trials. To evaluate the performance of each policy facing similarly distributed scenarios during and after training, we deployed the policy in simulation using scenes sampled from the training Scenic program under uniform sampling. We used 3 seeds, each running for 30 episodes and then take the mean episode return.

Sampler	Agent 0	Agent 1
BO	24.984 ± 0.513	29.683 ± 2.519
MAB	29.172 ± 1.116	33.505 ± 4.921
Halton	26.762 ± 4.741	32.821 ± 7.166
Uniform	29.353 ± 1.836	34.497 ± 4.052

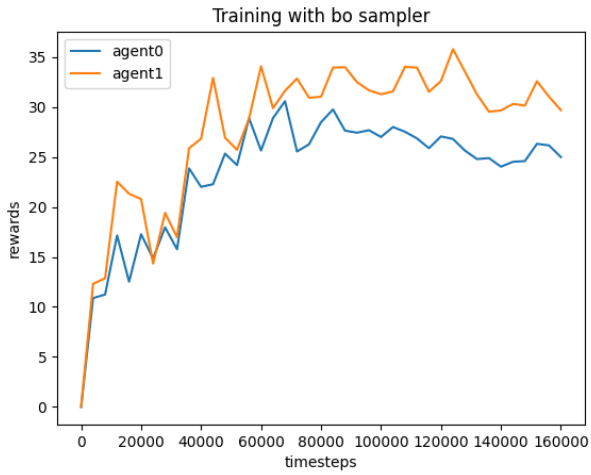
Table 4.1: Mean Episode Return ± Standard Deviation Evaluated in Under Scene Generated from Uniform Sampling



(a) Agent 0 Training Comparison

(b) Agent 1 Training Comparison

Figure 4.1: Comparison of training results between different samplers for each agent. Besides the BO sampler, the other samplers have relatively similar performance. This could be due to the reduced sample diversity resulting from the BO sampler



(a) Bayesian Optimization

(b) Multi-Armed Bandit

(c) Halton

(d) Uniform

Figure 4.2: Comparison of training results between the two agents in each sampler. Agent 1 generally outperforms Agent 0, likely due to the larger variation we allow on its initial position that includes points closer to its goal

4.3 Results and Discussion

Table 4.1 shows the final evaluation performance of each policy. Figure 4.1 and 4.2 shows the training curve for individual policies for each agent. In 4.1 we compare, for each agent, the

training performance using different samplers, and in Figure 4.2, we compare the performance of the two agents during the same training session with the same sampler. Note that the training curves also include the pretraining stages using a uniform sampler. It is the general trend that Agent 1 performs better than Agent 0, with a larger variance. This is most likely be due to differences between each car's configuration as the intersection, since we assigned to Agent 1 a greater range of values for its starting distance from the intersection, which include points closer to the goal point. We note that the MAB performed on par with the uniform finetune baselines. MAB is the archetypal active sampler, which would actively seek to discover scenarios yielding low rewards among the agents while explore under-sampled regions for new difficult episodes. The Halton sampler and BO sampler did not perform as well as the uniform baseline. The Halton sampler is deterministic: given a range on the real line, it "samples" points in a way such that the range is gradually covered, and each segment of the range is visited equally often. This behavior of the sampler likely yielded the larger variance in the trained policy's performance across trials: the diversity of samples generated could yield a more varied range of performance. The BO sampler seems to have significantly underperformed. We hypothesize that this is due to Bayesian optimization having a weaker emphasis on sample diversity when generating scenes, unlike the Halton and MAB samplers. This decreased diversity likely hurt training. While the uniform baselines seems to have performed better than the policies trained with the VerifAI samplers, we would like to note that it is likely that a full falsification could offer more definitive insight. Overall, the experiments points to sample diversity to be an important factor when choosing the VerifAI samplers for RL training, even during the later stages.

Chapter 5

Conclusions

We introduce a new Scenic/VerifAI[6][21][4] based tool, ScenicGym, for training RL agent using data generated by the Scenic probabilistic programming language. Scenario coverage during training is enforced by VerifAI. The new API is compatible with existing RL workflows and libraries, and demonstrated VerifAI's sampler being able to influence the final performance during evaluation. Some directions merit further explorations.

5.1 Future Work

Cross-Simulator Training

One direction that we did not describe in this thesis was cross-simulator training. ScenicGym currently allows for cross-simulator training between simulators to which Scenic has an interface. This direction is especially important in tasks like vision-based manipulation in robotics, where the visual and physical fidelity of the simulation environment are important. However, for training for these tasks are often bottle-necked by the intense computational resource requirements of high-end simulators. So it would be desirable to pre-train a policy on a low-fidelity simulator that incurs lower computational cost and then transfer the training to a high-fidelity simulator for finetuning to improve robustness. ScenicGym/Zoo could serve as a uniformed platform for these kinds of training. Towards this end, one ongoing effort is interfacing Scenic to MetaSim, which is a part of RoboVerse [8]. MetaSim is a "wrapper" simulator that provides a single unified API for using an array of popular robotics simulator, such as IsaacSim[13] and Mujoco[19]. The user can transfer training setup/scenarios from one simulator to another with a keyword argument. Interfacing Scenic to this simulator will provide additional environmental modeling and error analysis capability, yielding a strong cross-simulator training platform for assured AI-enabled robotics.

Hierarchical Reinforcement Learning

One of Scenic's native features is generating compositional scenarios, where, starting from a parent scenario, we can randomly "transition" into a second child scenario. For instance, the parent scenario could be a car driving on the road, for which we have a Scenic program. We then provide a few other Scenic programs, such as entering an intersection, turning, or lane-changing. Scenic can start in the parent scenario and transition into these sub-scenarios based on user specified distributions. This offers a good foundation for training hierarchical reinforcement learning, where an RL agent not only needs to be able to perform individual tasks, but also needs to be able to discern which task it should perform and how to correctly transition between its different operating modes.

Bibliography

[1] Abdus Salam Azad, Edward Kim, Mark Wu, Kimin Lee, Ion Stoica, Pieter Abbeel, Alberto Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Programmatic Modeling and Generation of Real-time Strategic Soccer Environments for Reinforcement Learning". In: *Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press, Feb. 2022.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[3] Holger Caesar, Varun Bankiti, Alex H. Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. *nuScenes: A multimodal dataset for autonomous driving*. 2020. arXiv: `1903.11027 [cs.LG]`. URL: `https://arxiv.org/abs/1903.11027`.

[4] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems". In: *31st International Conference on Computer Aided Verification (CAV)*. July 2019.

[5] Daniel J. Fremont, Johnathan Chiu, Dragos D. Margineantu, Denis Osipychev, and Sanjit A. Seshia. "Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI". In: *32nd International Conference on Computer Aided Verification (CAV)*. July 2020.

[6] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Scenic: a language for scenario specification and scene generation". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '19. ACM, June 2019, pp. 63–78. DOI: `10.1145/3314221.3314633`. URL: `http://dx.doi.org/10.1145/3314221.3314633`.

[7] Daniel J. Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A. Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. "Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World". In: *23rd IEEE International Conference on Intelligent Transportation Systems (ITSC)*. Sept. 2020.

[8] Haoran Geng, Feishi Wang, Songlin Wei, Yuyang Li, Bangjun Wang, Boshi An, Charlie Tianyue Cheng, Haozhe Lou, Peihao Li, Yen-Jen Wang, Yutong Liang, Dylan Goetting, Chaoyi Xu, Haozhe Chen, Yuxi Qian, Yiran Geng, Jiageng Mao, Weikang Wan, Mingtong Zhang, Jiangran Lyu, Siheng Zhao, Jiazhao Zhang, Jialiang Zhang, Chengyang Zhao, Haoran Lu, Yufei Ding, Ran Gong, Yuran Wang, Yuxuan Kuang, Ruihai Wu, Baoxiong Jia, Carlo Sferrazza, Hao Dong, Siyuan Huang, Yue Wang, Jitendra Malik, and Pieter Abbeel. *RoboVerse: Towards a Unified Platform, Dataset and Benchmark for Scalable and Generalizable Robot Learning*. 2025. arXiv: 2504.18904 [cs.RO]. URL: https://arxiv.org/abs/2504.18904.

[9] John H. Halton. "On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals". In: *Numerische Mathematik* (1960).

[10] Mukul Khanna, Yongsen Mao, Hanxiao Jiang, Sanjay Haresh, Brennan Shacklett, Dhruv Batra, Alexander Clegg, Eric Undersander, Angel X. Chang, and Manolis Savva. *Habitat Synthetic Scenes Dataset (HSSD-200): An Analysis of 3D Scene Scale and Realism Tradeoffs for ObjectGoal Navigation*. 2023. arXiv: 2306.11290 [cs.CV]. URL: https://arxiv.org/abs/2306.11290.

[11] Quanyi Li, Zhenghao Peng, Lan Feng, Qihang Zhang, Zhenghai Xue, and Bolei Zhou. "Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[12] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. *RLlib: Abstractions for Distributed Reinforcement Learning*. 2018. arXiv: 1712.09381 [cs.AI]. URL: https://arxiv.org/abs/1712.09381.

[13] NVIDIA. *Isaac Sim*. Version 5.0.0. URL: https://github.com/isaac-sim/IsaacSim.

[14] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[16] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. "Toward Verified Artificial Intelligence". In: *Communications of the ACM* 65.7 (2022), pp. 46–55.

[17] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2020.

[18] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. "Pettingzoo: Gym for multi-agent reinforcement learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 15032–15043.

[19] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[20] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. "Gymnasium: A Standard Interface for Reinforcement Learning Environments". In: *arXiv preprint arXiv:2407.17032* (2024).

[21] Eric Vin, Shun Kashiwa, Matthew Rhea, Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. *3D Environment Modeling for Falsification and Beyond with Scenic 3.0*. 2023. arXiv: 2307.03325 [cs.PL]. URL: https://arxiv.org/abs/2307.03325.