Scalable Techniques for Sampling-Based Falsification of AI-Based Cyber Physical Systems

Kesav Viswanadha

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-103
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-103.html

May 14, 2021

Acknowledgement

I would like to thank my advisor, Professor Sanjit A. Seshia, for all of his guidance during this program and for providing me the opportunity to engage in really interesting research efforts as part of the Learn and Verify group at UC Berkeley. I would also like to thank my mentors, Edward Kim and Professor Daniel J. Fremont, for all their invaluable contributions both to this work and to my research experience. I would also like to thank Francis Indaheng for his collaboration on the evaluation sections of this work. Last but not least, I would like to thank my family and friends for being incredibly supportive throughout this journey. None of this would be possible without you.

Scalable Techniques for Sampling-Based Falsification of AI-Based Cyber Physical Systems

by

Kesav Viswanadha

A thesis 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

Scalable Techniques for Sampling-Based Falsification of AI-Based Cyber Physical Systems

by Kesav Viswanadha

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sanjit A. Seshia
Research Advisor

5/14/2021

(Date)

* * * * * * *

Professor Venkatachalam Anantharam
Second Reader

5/11/21

(Date)

Abstract

Scalable Techniques for Sampling-Based Falsification of AI-Based Cyber Physical Systems

by

Kesav Viswanadha

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

As autonomous vehicle (AV) technology grows more widespread, questions still persist about how to effectively verify their safety. Much progress has been made in developing testing methodologies such as falsification for autonomous vehicles that interface with driving simulators to generate rich sets of scenarios. We present extensions to the Scenic scenario specification language and VerifAI toolkit that improve the scalability of such methods by allowing for falsification to be done more efficiently and with more complex models of the end goal. We first present a parallelized framework that is interfaced with both the simulation and sampling capabilities of Scenic and the falsification capabilities of VerifAI, reducing the execution time bottleneck inherently present in simulation-based testing. We then present an extension of VerifAI's falsification algorithms to support multi-dimensional objective optimization during sampling, using the concept of rulebooks to specify multiple metrics and a preference ordering over the metrics that can be used to guide the counterexample search process. Lastly, we evaluate the benefits of these extensions with a comprehensive set of experiments written in the Scenic language.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Sanjit A. Seshia, for all of his guidance during this program and for providing me the opportunity to engage in really interesting research efforts as part of the Learn and Verify group at UC Berkeley. I would also like to thank my mentors, Edward Kim and Professor Daniel J. Fremont, for all their invaluable contributions both to this work and to my research experience. I would also like to thank Francis Indaheng for his collaboration on the evaluation sections of this work. Last but not least, I would like to thank my family and friends for being incredibly supportive throughout this journey. None of this would be possible without you.

Chapter 1

Introduction

With the advent of autonomous vehicles (AVs) in recent years, a number of fundamental questions have arisen about the reliability of self-driving systems, especially given the increased dependence on these systems that has led to reduced human involvement during driving. The intent of such systems is to perform driving tasks in a way that feels natural, abides by local traffic laws, and most importantly, avoids driving errors typically made by humans that lead to crashes. As such, it is crucial to thoroughly vet AV systems before they are deemed fit for the road. The question of what needs to be tested and prioritized during the development phase is an ongoing conversation both in the industry and academia [20].

It is unquestionable that autonomous vehicles are far from perfect. In 2017, Uber pulled its self-driving vehicles from the road following a crash where the AV perception module failed to recognize another car with a human driver making a left turn [28]. Similar situations arise in the news quite frequently, and can sometimes lead to injuries and death. Therefore, formal verification of autonomous vehicles, and more generally, safety-critical systems, is very much a necessary step in reducing the number of casualties caused by such systems in the real world. However, autonomous vehicles rely heavily on complex artificial intelligence (AI) and machine learning (ML) components, which pose unique challenges for formal methods [30].

The motivation for this work is to improve the scalability of software designed to test autonomous vehicle systems in simulation. While the rise of open-source and commercial driving simulators has led to massive improvements in the ability to test driving systems, systematically finding bugs in an efficient way is still a major challenge given that simulation of a full self-driving stack in a complex simulator environment is very computation intensive.

1.1 Problem Definition

Efficient Simulation-Based Falsification

Much of this work is rooted in concepts from formal methods as well as optimization. This problem can be formulated in a variety of ways, but one natural one that comes to mind is

a simulation-based falsification problem. Given a system S and a specification ϕ, the goal of falsification and fuzz testing is to generate environment inputs x that cause the system to violate the specification [13]. We draw these inputs from a from a *semantic feature space* defined by a Scenic program (more on this in Section 1.2). Interfacing this with a simulator, we generate a *trace*, or sequence of states, of the system given the initial input x. We can see that this formulation is well-suited to the autonomous driving domain; we are given the self-driving stack S, and it is the job of the developer to develop quantitative metrics to evaluate the performance of the AV system and provide a set of test inputs to the environment that represent a wide variety of possible situations that an autonomous vehicle could encounter. Of course, the space of all possible inputs to the environment is boundless, and part of the challenge of testing autonomous vehicles is that by nature, some new regions of this search space are only discovered post-production when unexpected situations arise.

Because of the level of uncertainty introduced by the many neural-network based components of autonomous vehicles as well as the space of possible environment inputs being hard to control, this formal verification process is one of iterative improvement. We attempt to address some of the numerous challenges presented in formulating specifications and modeling environments in a way that allows for scalable falsification of self-driving systems [30]. One possible goal during verification is to find as many (interesting) corner cases as possible that cause unexpected behavior in the self-driving stack. However, this poses some problems as this goal can be reached in a rather simple way; if an input x is found that violates the specification, we can typically find many inputs x' that are very similar to x that also violate the spec. While these inputs are technically different from each other, there is not much *diversity* in the set of generated semantic feature vectors.

To mitigate this, another possible goal that can be formulated for this problem is to maximize the diversity of inputs that lead to a counterexample. We qualitatively assess this metric for the methods presented in Chapter 3. We can also consider that in the case that we are trying to have as comprehensive of a test case suite as possible, we may be interested in a notion of coverage of a search space. This is particularly important in testing as the space of possible inputs is very large, so we want sample inputs from a large area of the semantic feature space defined by the environment. Chapter 2 talks more about the metrics used to this end and methods we implement for improving these metrics for a set of benchmarks.

Verification with Multiple Metrics

The second part of the problem formulation comes from the motivation that there are typically multiple metrics involved in the evaluation of an AV system. In this case our specification ϕ is a vector of either real or boolean values, where the j-th entry of ϕ corresponds to the value returned by the specification of the j-th metric. We are given a partial order over the entries of ϕ in terms of which metrics are violated and their relative importance. The formal definition of this partial ordering is discussed in Chapter 3. We will now discuss some of the prior work and foundation for the contributions made towards a solution to the above problems.

1.2 Related Work

This work relies on several tools developed at UC Berkeley; we provide some background on these tools as they pertain to the contributions made in this paper.

Overview of Scenic and Related Tools

Scenic is a probabilistic programming language developed by Daniel Fremont et al [15, 14]. It allows the specification of *abstract scenarios*, which are definitions of parameters in an environment that are generated according to a random distribution. In the domain of AVs, these parameters could be the locations, positions, and heading of the ego vehicle, other vehicles, pedestrians, or inanimate objects in the scene; their positions relative to each other; and other external factors such as the time of day, weather, colors of cars and buildings, etc. As a programming language, Scenic is very similar to Python, with special syntax added for easily defining variables in terms of distributions and relating them to each other in an intuitive way.

For example, consider the simple Scenic program in Figure 1.1. The script is able to import maps in OpenDrive format [3] as Scenic houses a built-in parser that turns these maps into road objects internal to Scenic, encoded in the `network` variable. Scenic also provides an interface to several open-source driving simulators; in this case we use the CARLA simulator [12] (see Figure 1.2). We then instantiate an object which uniform randomly samples from all the possible roads in the network, as well as a lane random uniformly sampled from the road that is picked. Note that when the scenario is compiled, it still is only encoded as an abstract scenario; therefore, the `select_road` and `select_lane` values are still not sampled. This example shows how Scenic is able to internally maintain the dependencies between randomly sampled variables and only assign them concrete values at runtime.

The program in Figure 1.1 is an example of a *static scenario*; the initial configurations of the objects are generated in Scenic, but it is now up to the user to decide how to use these in a full simulation. Scenic also supports the specification of *dynamic scenarios* - these are scenarios for which parameters are time-dependent and sampled by Scenic at every timestep [16]. Figure 1.3 gives an example of such a dynamic scenario; the ego and parked cars are both assigned *behaviors*. These behaviors are also written in the Scenic language and specify certain ways for a vehicle to drive. The Scenic interface with the simulator then uses these behaviors to sample the actions at each timestep and converts them into simulator-specific controls. The simulator then updates the state of world and sends the pertinent information back to the Scenic interface, and the simulation continues until a maximum number of timesteps is reached. All of the extensions made to Scenic in this work can be applied to both dynamic and static scenarios.

```
param map = localPath('../../tests/formats/opendrive/' +
                      'maps/CARLA/Town01.xodr')
param carla_map = 'Town01'
model scenic.domains.driving.model

select_road = Uniform(*network.roads)
select_lane = Uniform(*select_road.lanes)
ego = Car on select_lane.centerline

right_sidewalk = network.laneGroupAt(ego)._sidewalk

Pedestrian on visible right_sidewalk

```

Figure 1.1: Example Scenic script which places a pedestrian and car in the environment [4].

Figure 1.2: A sampled scene of the script in Figure 1.1 in the CARLA simulator [12].

```
param map = localPath('../../tests/formats/opendrive/' +
'maps/CARLA/Town05.xodr')
param carla_map = 'Town05'
param time_step = 1.0/10

model scenic.domains.driving.model

behavior PullIntoRoad():
    while (distance from self to ego) > 15:
        wait
    do FollowLaneBehavior(laneToFollow=ego.lane)

ego = Car with behavior DriveAvoidingCollisions(avoidance_threshold=5)

rightCurb = ego.laneGroup.curb
spot = OrientedPoint on visible rightCurb
badAngle = Uniform(1.0, -1.0) * Range(10, 20) deg
parkedCar = Car left of spot by 0.5,
                facing badAngle relative to roadDirection,
                with behavior PullIntoRoad

require (distance to parkedCar) > 20

monitor StopAfterInteraction:
        for i in range(50):
                wait
        while ego.speed > 2:
                wait
        for i in range(50):
                wait
        terminate
```

Figure 1.3: An example of a dynamic scenario written using Scenic [4].

Overview of VerifAI and Related Tools

VerifAI is a toolkit written in the Python language that provides implementations of many algorithms for the formal verification of AI-based cyber physical systems (CPS) [13]. The main use case of VerifAI is *falsification*, which can be loosely defined as the systematic generation of samples defined by a semantic feature space with the intent of finding counterexamples to a specification over an AV system. Specifications are also written in VerifAI using what is known as a *monitor*; these terms are more formally defined in Chapter 2, but roughly speaking, VerifAI monitors give users the flexibility to write both simple and complex metrics to use to evaluate a simulation run.

VerifAI also supports encoding monitors as metric temporal logic (MTL) formulas. These are similar in nature to linear temporal logic formulas, except that instead of encoding conditions as booleans, we use real numbers that can quantify how strongly a condition is satisfied or weakened. An example of this could be the following: `"G(collisioncone0 & collisioncone1 & collisioncone2)"` [36]. This can be translated as the three variables having to be positive over the entire course of the simulation; if any of them become negative, the sample corresponding to the simulation is considered a counterexample.

VerifAI provides a number of sampling strategies to efficiently search the feature space defined either directly with VerifAI's APIs or using a Scenic program. Before this work, VerifAI only supported the creation of objectives that return a single real number as the definitive evaluation metric of the AV system. Many of the sampling algorithms implemented, including cross-entropy sampling, Bayesian optimization, and simulated annealing, are algorithms commonly used in the optimization domain that are specifically meant to drive the search process towards regions of the feature space that have lots of counterexamples.

Other tools have been developed for the falsification of simulated systems on various platforms [2]. Another related field that has become increasingly popular is adversarial machine learning - this is particularly useful for the falsification of deep neural networks (DNNs), as these are differentiable systems and it is possible to perform black-box attacks by maliciously perturbing inputs in a way that causes incorrect output from the DNN [17]. These techniques, however, are rather restricted in their scope for autonomous vehicles as they are only able to falsify specific components of the driving stack, such as perception, prediction, planning, or control [26]. The techniques we present that extend the capabilities of VerifAI are intended to work on the system level; users are able to provide specifications over the end-to-end system as well as independent components that the falsification algorithms will try to violate.

Overview of Parallelization of Self-Driving Simulations

A big part of this work is in improving the efficiency of sampling-based falsification methods, many of which are reliant on being able to run open-source driving simulators such as CARLA in parallel. Samples, or concrete parameters generated to initialize a simulated scenario, need to be simulated using a world with a dynamics model, which is a computa-

tionally intensive task. Therefore, being able to run many simulations at once in parallel is beneficial in improving the runtime of a falsifier. The functionality to run many instances of a simulator is natively available in CARLA [35], and we take advantage of this in our work on parallelizing the falsification effort, as we are able to easily run multiple instances of CARLA and have simulations run in each independently. This has also been implemented by the LGSVL simulator [21], but the current setup involving running simulations on a cloud cluster makes it more well-suited to coverage testing rather than the active sampling methods implemented in VerifAI and augmented in this work. There has also been some recent work on parallel computation over temporal logic formulas. Cralley et al have proposed a variety of optimization algorithms that can be efficiently run in parallel for the purposes of falsification, including annealing algorithms [11], system simulation, and parallel robustness computations over temporal logic formulas. These robustness computations are akin to the computations performed by monitors in VerifAI; we apply these algorithms in the context of end-to-end integration with Scenic, which allows not only the generation of samples but the simulation of these samples with scenarios generated by Scenic. We discuss more on the implementation and novel idea of this work below.

Overview of Multi-Dimensional Objective Specification and Falsification

One use case of falsification that we present involves sampling in a way that optimizes multiple evaluation metrics at the same time. Censi et al [9] provide a framework for specifying multi-dimensional objectives using a concept called *rulebooks*. This concept is elaborated further in Chapter 3, but the high-level idea is to encode pairwise preferences between different metrics as edges in a directed graph (referred to as the "rulebook"). This graph then defines a partial pre-order over the dimensions of the objective which can be used to guide planning algorithms which optimize over these multiple objectives. There are many possible ways to reconcile the ambiguity present in ordering multi-dimensional metric vectors, and these are just a few of the ones pertinent to the implementation we present on top of VerifAI for this work. For example, it is also possible to construct a Pareto frontier of samples that partially violate the multi-dimensional objective, as has been tried in the reinforcement learning community [25]; however, this does not leverage the preference ordering over metrics that exists in many practical cases. There is also recent work on formulating principles of driving, such as "no collisions" or "obey traffic laws", in a mathematical sense using specific metrics [37]. These are particularly useful in applying concepts of multi-dimensional objective falsification to an AV industry setting, where there may be many objectives to keep in mind during testing.

There has also been some work in implementing planning algorithms in autonomous vehicles that simultaneously optimize multiple evaluation metrics. Abstractly speaking, this is a similar idea to multi-dimensional objective falsification – except instead of finding optimal trajectories, we are trying to find trajectories that are as suboptimal as possible.

For example, Castro et al have designed a planning algorithm called MVRRT* [8] which incrementally builds up better and better trajectories that minimize a level of unsafety computed over multiple metrics. Many of the same techniques can be used for both tasks as it is not hard to formulate one task in terms of the other.

1.3 Novel Contributions and Outline

The novel contribution of this work two-fold: in Chapter 2, we discuss the theory and implementation of parallelized falsification in the VerifAI toolkit. This work is unique in that it leverages several different components involved in the falsification of AV systems into one pipeline that provides end-to-end service with relative ease on the user's end. Whereas many open-source simulators provide the ability to spawn objects in an environment and execute actions, this pipeline aims to make this process much simpler by allowing the specification of environment inputs and behaviors at a higher semantic level in a way that is simulator-agnostic. This effort also incorporates ideas from parallelized optimization to make sure that the active samplers implemented in VerifAI are able to make use of the parallelism and more effectively find regions of counterexamples.

In Chapter 3, we discuss the approach and implementation of several new algorithms in the Scenic and VerifAI toolkits that help improve the scalability of falsification algorithms by extending them to work with multi-dimensional objectives. This is a key step forward in AV testing, as it is much more representative of the real world to be evaluating AV systems on multiple metrics at the same time. We present many novel ideas in the specification and optimization over these metrics that are built on ideas already discussed in the literature and partially put forth by Scenic and VerifAI.

We also evaluate the work covered by each of these chapters with a comprehensive set of Scenic programs that cover a wide range of possible scenarios that an AV system could encounter in the real world. This library of Scenic programs serves as a benchmark for the performance of the parallelized falsification pipeline, as well as a basis on which to demonstrate the efficacy of the novel multi-dimensional objective falsification algorithms [4, 5].

Chapter 2

Parallelization of Search Algorithms

2.1 Approach

Background

Before discussing the implementation of parallelization, we define several pertinent terms and notations that will be used throughout the rest of this work:

- Let x denote a *semantic feature vector* sampled by VerifAI, having dimension d.
- The i-th (scalar) value of the feature vector is denoted by x_i.
- Define $\mathcal{S} \subseteq \mathbb{R}^d$ as the space of all possible sampled features.
- Let $\rho(x) : \mathcal{S} \mapsto \mathbb{R}$ denote a specification function which takes in a feature vector x and outputs a scalar value over the trajectory of the system generated from the parameters x.
- Let ρ^* be the *threshold* of the specification, such that if $\rho(x) < \rho^*$, x is a *counterexample* to the specification function. Typically in VerifAI, we set $\rho^* = 0$ and normalize the output of the monitor accordingly.

Common terms used in VerifAI:

- **Monitor:** A Python function which maps a trajectory to a scalar value, i.e. calculates $\rho(x)$ after the full trajectory according to the parameters x is generated by the system.
- **Scene:** An object describing all the objects present (at the current timestep) in the environment, such as the ego vehicle, other vehicles, pedestrians, buildings, etc. Also contains information about the current state of various objects, including their current position and trajectory, as well as information specific to the simulator being used for falsification, such as the map being used.

```
ego = Car at VerifaiRange(-5, 5) @ 2, with behavior FollowLaneBehavior
car1 = Car at 4 @ VerifaiRange(-1, -2)
```

Figure 2.1: Example Scenic code snippet instantiating two cars whose x-coordinates are determined by independent uniform random distributions

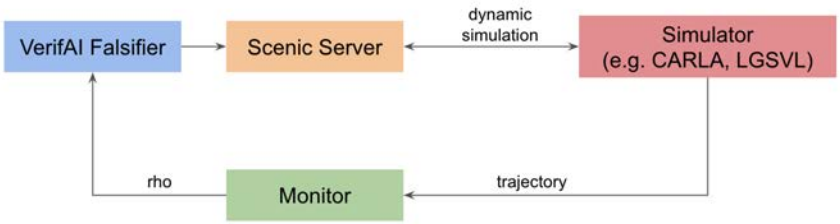

Figure 2.2: Typical pipeline for falsification using VerifAI.

- **Sampler:** Responsible for generating values of the feature vector x according to the constraints specified in the Scenic program. Can be done in either Scenic or VerifAI depending on the user's preference. May or may not maintain internal state as needed to generate the next sample (i.e. the history of previously sampled values and their outcomes given the specification).
- **Falsifier:** An artifact of VerifAI that takes as input a sampler and a monitor, and generates a certain number of samples (specified by the user), creating an *error table* and *safe table* to store the samples that violate and satisfy the specification, respectively.
- **Scenic Server:** A property of the falsifier which is responsible for communicating with the simulator to generate a time-series sequence of actions for a given sample. This is how the trajectory is generated from a feature vector x.

Figure 2.1 is an example of a Scenic program that instantiates a feature vector in $\mathbb{R}^2$; x_1 is the x-coordinate of the ego vehicle, and x_2 is the y-coordinate of the of another car present in the scene. The use of the Scenic syntax `VerifaiRange(low, high)` allows the use of VerifAI's specialized samplers to sample the values of the individual features, constrained by the continuous range denoted by `[low, high]`.

These specialized samplers broadly fall into two categories: passive samplers, which emphasize exploration and coverage of the search space, and active samplers, which emphasize exploitation and the localization of regions of the search space which result in *falsifying counterexamples*, i.e. samples x where $\rho(x) < \rho^*$.

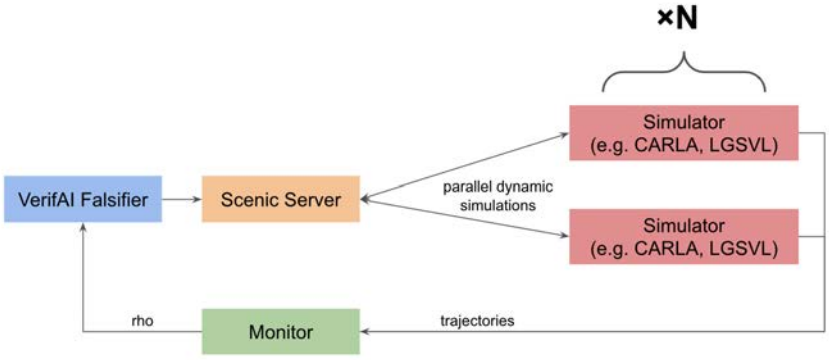

Figure 2.3: Parallelized pipeline for falsification using VerifAI.

Figure 2.2 demonstrates a typical pipeline during the execution of a VerifAI falsifier driven by a Scenic program. Initial parameters are generated using samplers in either Scenic or VerifAI; these parameters are then used to open a connection between the Scenic Server and the driving simulator and generate a trajectory in the simulated world. This trajectory is then evaluated by the monitor, deemed either a safe example or a counterexample, and added to the corresponding table in the falsifier.

The biggest bottleneck in this pipeline is the *generation of the trajectory* from the feature vector; i.e. the communication between the Scenic Server and the simulator. Empirically, it was observed that the time to run a CARLA simulation that lasts 30 seconds in simulator time took almost 30 seconds to run in physical time as well. Many common tasks that involve falsification, especially for a complex specification, require hundreds or even thousands of samples to begin seeing meaningful results [19]; this means that a single falsifier run can take on the order of several hours.

Parallelized Pipeline

To combat the simulation bottleneck, we leverage process-level parallelism to run multiple simulations at once. Figure 2.3 demonstrates the modified falsification pipeline to include multiple parallel workers that can all run simulations in tandem. One important aspect to note about this pipeline is that it requires multiple instances of the simulator to be running; many widely-used simulators (such as CARLA or LGSVL) listen on specific ports for HTTPS or WebSocket connections, which means that each simulator instance needs to communicate with a Scenic Server instance on a different port.

The parallelization of the falsifier pipeline required significant changes to the underlying architecture in VerifAI. For instance, the sampling process was decoupled from the trajectory generation process; this was necessary to ensure that the parallel workers do not perform repeated work by simulating the same samples. Consider, for example, the Halton sam-

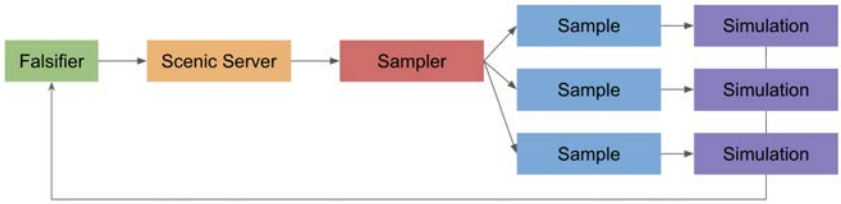

Figure 2.4: Centralized sampling architecture for parallel falsification.

Figure 2.5: Legacy sampling and updating process in active samplers. The `nextSample()` function is responsible for updating the internal state by calling `update()` with the most recently generated sample.

pler, which is a passive sampler that partitions the search space into smaller and smaller intervals over time and samples from the middle of the current interval. Given the variable `VerifaiRange(-1, 1)`, the Halton sampler would generate the following values at each sample, deterministically:

$$0, -0.5, 0.5, -0.75, -0.25, 0.25, 0.25, ...$$

Due to the deterministic nature of the Halton sampler (and many others in VerifAI, including those that are optimization-based), if each parallel worker generates its own sequence of samples with its own sampler, they will all simulate the same set of samples, defeating the purpose of parallelism in the first place. Therefore, we centralize the sampling process in the Scenic Server; Figure 2.4 shows the modified architecture implemented to support this.

For passive samplers, these changes were sufficient to ensure the same functionality as the unparallelized sampler, as the sampler only updates its internal state based on the values generated at sample-time. However, for active samplers, there is a notion of using the history of previously generated samples and their outcomes in simulation to drive the internal state (through some form of optimization), and thus, future samples generated as well. Figure 2.5 shows how active samplers were originally implemented in VerifAI; the key idea is that they are able to update their internal state at sample-time, as they cache the most recently generated sample, which without parallelization is the sample that is used to update the internal state. As a concrete example, consider the following pseudocode for

the cross-entropy sampler based on work by Sankaranarayanan et al [29], which updates a probability distribution over a set of continuous intervals:

```
class CrossEntropySampler(low, high)

    discretize the interval (low, high) into N buckets
    initialize probability_distribution with probability 1/N for each bucket
    initialize bucket_of_last_sample

    function sample():
        if bucket_of_last_sample was a counterexample:
            increase probability of bucket_of_last_sample and re-normalize
        sample the next bucket from probability_distribution
        cache the sampled bucket in bucket_of_last_sample
        return a random uniformly sampled value from that bucket
```

Looking at this example, we can see that in the parallel setting this will not work due to the non-determinism of the execution order of parallel processes. When the `sample()` function is called, it is not guaranteed that the simulation run corresponding to `last_sample` has actually finished running. This will cause blocking and make the sampling process inefficient in a parallel setting. To mitigate this issue, we decouple the sampling process from the state updating process, as shown in Figure 2.6. For the sake of consistency, we applied these changes to the parallelized and unparallelized versions of the various samplers. The pseudocode for the updated sampler pipeline is as follows:

```
class CrossEntropySampler(low, high)

    discretize the interval (low, high) into N buckets
    initialize probability_distribution with probability 1/N for each bucket
    initialize bucket_of_last_sample

    function sample():
        sample the next bucket from probability_distribution
        return a random uniformly sampled value from that bucket

    function update(bucket_value, rho):
        if rho corresponds to a counterexample:
            increase probability of bucket_value and re-normalize
```

In this new setup, the falsifier (which is centralized in one process) is responsible for calling the `sample()` and `update()` functions as simulation runs from the parallel workers finish and new samples are needed. This way, the sampler does not need to cache any generated samples

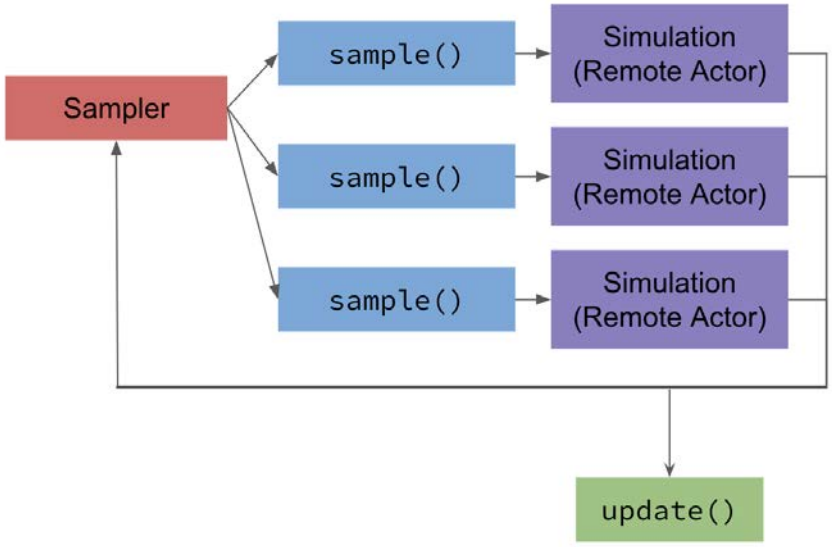

Figure 2.6: Decoupled sampling and updating process for parallel active samplers.

and will only update the probability distribution when simulation runs finish. For example, a possible execution of this parallelized falsifier could be: `sample value1, sample value2, sample value3, update for value2, update for value1, update for value3`. Because of the nondeterminism of the order that the parallel processes finish, we need to make sure that we can do an update to the internal state with any sample, not just the most recently generated sample. To do this, we remove the guarantee that the sampler is always choosing samples according to the most recently computed probability distribution, as it is possible for `sample1` to be generated and then `sample2` to be generated before the simulation corresponding to `sample1` has finished running. However, this guarantee is not particularly important in the long run, as the probability distributions will not change much with each update. Therefore, sampling from a slightly older distribution will not result in much lower quality samples.

The parallelization was implemented in the source code for the VerifAI Python library [5] using RAY, a framework that encapsulates process-level parallelism using the concept of remote *actors* (classes) and functions, which run in their own process [23]; Figure 2.6 shows where in the pipeline these actors are used. In this case, the remote functions were responsible for generating a scene from a sample, and running a simulation with that scene to create a trajectory. These remote functions are called from the centralized falsifier, creating *futures* which can be polled with some frequency to check whether they have finished executing or not. Once the result of the execution is ready, the `update()` function is called, a new sample is generated, and a new future is created in the given worker using the new sample.

2.2 Evaluation

There are two primary metrics that are logical to benchmark the effectiveness of parallelization. One way is to run a falsifier with a fixed number of samples in both a parallel and serial setting and record the wall clock times of each. Another perhaps more intuitive way, which is used in the experiments in this section, is to run a falsifier for a fixed amount of time in both a parallel and serial setting and record the number of counterexamples generated in each. In either case, the hypothesis prior to experimentation is that the parallel workers will allow for more efficient searching over the feature space and lead to counterexamples being discovered more quickly.

The Newtonian Simulator

One practical but considerable roadblock faced when using VerifAI is the heavy reliance on computationally intensive driving simulators such as CARLA or LGSVL. These simulators require a powerful GPU and specific operating systems to be run, and because these simulators are attempting to be hyper-realistic in simulating the real world, simulation takes a long time to run. To cut down on experimentation time and facilitate testing of new features during development, we present a so-called "Newtonian Simulator" which incorporates a simplistic dynamics model to simulate the motion of vehicles. The controls sent to a vehicle from either a real-world AV system such as Apollo or a custom-written controller can be broken down into three simple actions: `throttle`, `brake`, and `steering`. We use equations relating steering angles to the corresponding angular acceleration given the length of the car [27], and relate the throttle value to the linear acceleration of the car using an estimate of typical maximum acceleration for today's cars. Given this information, as well as a provided Δt which indicates the granularity of the timesteps, we update the position and velocity vectors of the vehicle using Euler's method for estimating solutions to ordinary differential equations.

Figure 2.7 shows a screen capture of a simulation run in the Newtonian simulator. The simulator supports basic rendering of environment maps using the OpenDrive format [3], as seen by the lane lines and the turning trajectories plotted in the simulator. The simulator does not support collision detection or interaction with objects in the environment (such as pedestrians, buildings, sidewalks, etc); however, one can easily detect collisions between vehicles with a VerifAI monitor that calculates the distance between the centers of each vehicle and the heading angle of each vehicle to determine an overlap.

Experiment Setup

A set of Scenic scripts was developed by Francis Indaheng [18] to comprehensively benchmark the performance of the parallelism and the advanced sampling/falsification methods

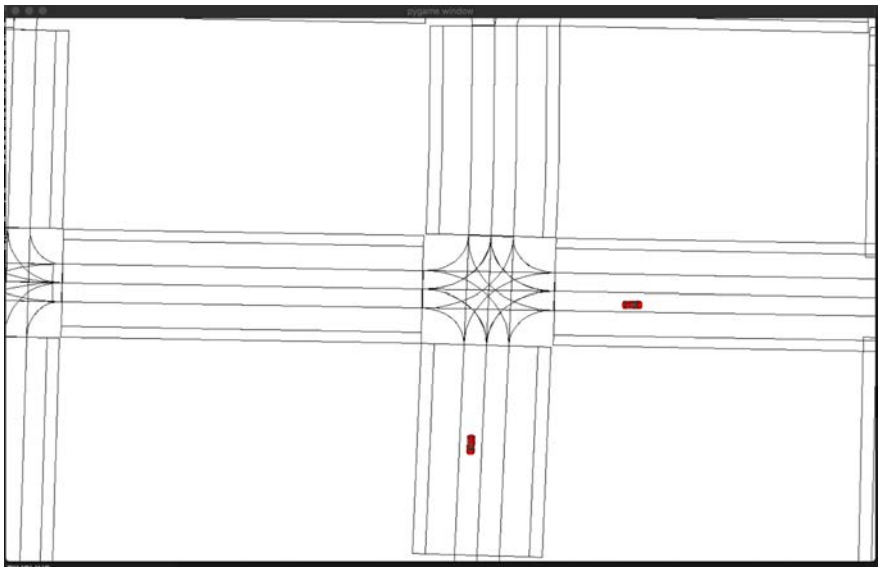

Figure 2.7: A screen capture from a simulation run using the Newtonian simulator.

described in the next chapter [1]. These scripts are based on the list of pre-crash scenarios described by the National Highway Traffic Safety Administration (NHTSA) [24]. These include:

1. Ego vehicle goes straight at 4-way intersection and must suddenly stop to avoid collision when adversary vehicle from oncoming parallel lane makes a left turn.

2. Ego vehicle makes a left turn at 4-way intersection and must suddenly stop to avoid collision when adversary vehicle from oncoming parallel lane goes straight.

3. Ego vehicle either goes straight or makes a left turn at 4-way intersection and must suddenly stop to avoid collision when adversary vehicle from perpendicular lane continues straight.

4. Ego vehicle either goes straight or makes a left turn at 4-way intersection and must suddenly stop to avoid collision when adversary vehicle from perpendicular lane makes a left turn.

5. Ego vehicle waits at 4-way intersection for adversary vehicle from oncoming parallel lane to complete a left turn before making a right turn.

6. Ego vehicle waits at 4-way intersection for adversary vehicle from perpendicular lane to pass before making a right turn.

[1] Source of these Scenic scripts available at https://github.com/findaheng/Scenic/tree/behavior_prediction/examples/carla/Behavior_Prediction

7. Ego vehicle makes a left turn at 3-way intersection and must suddenly stop to avoid collision when adversary vehicle from perpendicular lane continues straight.
8. Ego vehicle goes straight at 3-way intersection and must suddenly stop to avoid collision when adversary vehicle makes a left turn.
9. Ego vehicle waits at 3-way intersection for adversary vehicle from perpendicular lane to pass before making a right turn.
10. Ego Vehicle waits at 4-way intersection while adversary vehicle in adjacent lane passes before performing a lane change to bypass a stationary vehicle waiting to make a left turn.
11. Ego vehicle performs a lane change to bypass a slow adversary vehicle before returning to its original lane.
12. Adversary vehicle performs a lane change to bypass the slow ego vehicle before returning to its original lane.
13. Ego vehicle performs a lane change to bypass a slow adversary vehicle but cannot return to its original lane because the adversary accelerates. Ego vehicle must then slow down to avoid collision with leading vehicle in new lane.
14. Ego vehicle performs multiple lane changes to bypass two slow adversary vehicles.
15. Ego vehicle performs multiple lane changes to bypass three slow adversary vehicles.
16. Ego vehicle must suddenly stop to avoid collision when pedestrian crosses the road unexpectedly.
17. Both ego and adversary vehicles must suddenly stop to avoid collision when pedestrian crosses the road unexpectedly.
18. Ego vehicle makes a left turn at an intersection and must suddenly stop to avoid collision when pedestrian crosses the crosswalk.
19. Ego vehicle makes a right turn at an intersection and must yield when pedestrian crosses the crosswalk.
20. Ego vehicle goes straight at an intersection and must yield when pedestrian crosses the crosswalk.

The setup for these experiments is as follows: a selected subset of the scenarios above, encoded in Scenic programs, were run for 30 minutes with a VerifAI falsifier. For all of these scenarios, the monitor returns the minimum distance in meters between centers of the ego vehicle and an adversary vehicle. Through empirical observation, we decided that the threshold should be set to $p^* = 5$ meters. This approximately corresponds to counterexamples being

logged any time there is a crash. These falsification runs were done on a 20-CPU Ubuntu server that uses a Titan RTX GPU to run CARLA. All parallelized experiments were run using 5 workers to perform simulation. This was the maximum the server could handle, as the single GPU shares resources across all the threads that were running CARLA. Each Scenic program was run with both a parallel and serial falsifier configuration, as well as a passive sampler (Halton) and an active sampler (cross-entropy). We benchmark the speedup factor by computing the ratio of the number of counterexamples found in the parallel falsification run to the number of counterexamples found in the serial falsification run. For the passive samplers, we present a metric of coverage in the form of a confidence interval. We consider sampling to be a Bernoulli process, where the probability of a random sampler violating the specification is given by some unknown parameter p. Using the Clopper-Pearson method [10], we generate a 95% confidence interval for the true value of p based on the observed probability $\hat{p}$. We compare the width of this interval for both the parallel and serial falsifiers for each scenario. Table 2.1 shows the observed number of samples and counterexamples from running these experiments, as well as the ratio of the parallel falsifier to serial falsifier confidence interval width.

We also benchmark the percentage of time spent in sampling and simulation for each iteration of the falsifier run. Because the sampling is centralized, the overhead of generating samples is irreducible by parallelizing the falsifier. Furthermore, the amount of time spent in sampling is greatly dependent on the scenario; this is because VerifAI uses an algorithm known as *rejection sampling*. Given a feature space, not all combinations of feature values will result in valid scenarios (e.g. a car might be placed on a sidewalk, or two vehicles' positions might overlap with each other). These validity checks are performed by analyzing the output from the simulator, which means that we have to keep sampling and rejecting samples that turn out to be invalid. Because of this, scenarios that have a very small region of the feature space with valid samples will require lots of rejection sampling to generate a single valid sample. However, sampling itself is generally relatively fast as it does not require any computation-intensive operations.

Nevertheless, the sampling time is non-negligible compared to the simulation time, especially when running the Newtonian simulator which has relatively simple calculations during simulation. For example, in the case of the Newtonian simulator running scenario 1 in headless mode (no display), roughly 75% of the total time taken by the falsifier is spent in sampling. By Amdahl's law [1], we have that the maximum possible speedup of a program is given by $\frac{1}{1-p}$, where p is the fraction of the program that is parallelizable. Therefore, if only 25% of the program can be parallelized in the Newtonian simulator, we will only see a theoretical maximum speedup of $1/0.75 \approx 1.33$ with infinite compute. Empirically, however, we observed that the overhead of instantiating multiple workers and coordinating inter-process communication outweighed the benefits of parallelism, causing parallel falsification to run slower than serial falsification for the Newtonian simulator in headless mode.

However, the sampling process is much faster than the simulation process for the more involved simulators such as CARLA; we found that for a variety of scenarios, the sampling

time was at most 20% of the total time spent running the falsifier. If we consider 80% as an upper bound on the fraction p of the program that is parallelizable, we have that the speedup factor s from 5 workers is bounded by:

$$s \leq \frac{1}{0.2 + 0.8/5} \approx 2.78$$

Therefore, we can see that we will not receive exactly 5x speedup from running the scenarios with 5 workers if the sampling time is significant. This is reflected in the data in Table 2.1, as the speedup factor is close to but not quite 5 for many scenarios.

Experiment Name	(**Counterexamples**/Total Samples) Halton (Serial)	Cross-Entropy (Serial)	Halton (Parallel)	Cross-Entropy (Parallel)	Confidence Interval Width Ratio	Speedup Factor
Scenario 1	**53**/203	**72**/204	**259**/831	**359**/782	0.51	4.94
Scenario 2	**92**/199	**88**/187	**454**/852	**422**/798	0.48	4.87
Scenario 11	**83**/106	**85**/106	**322**/393	**337**/428	0.48	3.92
Scenario 12	**87**/108	**92**/105	**331**/448	**369**/462	0.53	4.15
Scenario 13	**32**/138	**54**/128	**77**/352	**159**/374	0.61	2.55
Scenario 15	**56**/107	**79**/97	**180**/330	**300**/334	0.56	3.08
Scenario 16	**254**/885	**662**/855	**627**/2059	**1491**/2017	0.66	2.33

Table 2.1: Comparison of serial and parallel falsifiers with cross-entropy and Halton samplers run in CARLA. The confidence interval ratio measures the ratio of the width of the parallel falsification confidence interval to the width of the serial falsification confidence interval.

We can see that for most of these scenarios, there are approximately 4-5x more counterexamples generated by the parallelized falsifier in the same amount of time as the serial falsifier. There is some variance in the results, as the runtime of a single simulation is also dependent on how many other processes are running during the falsification runs (the machine is a shared machine with many others). Another interesting feature of the data is that for all of the scenarios, the width of the confidence interval for the parallel Halton sampler is roughly half of that of the serial Halton sampler. This is approximately what is expected, as

the Clopper-Pearson method generates a confidence interval whose width is inversely proportional to $\sqrt{n}$, where n trials are run. If we consider this confidence interval width to be our metric of coverage, then the data show that through the parallelized falsification pipeline, we are able to get roughly double the coverage of the feature space with 5 parallel workers. It remains to do more extensive experiments with an industrial setup (e.g. dozens or even hundreds of parallel workers on a cloud instance) and see how this coverage metric scales.

Chapter 3

Multi-Dimensional Objectives

3.1 Approach

Background

VerifAI monitors, as described in the previous chapter, are functions $\mathbb{R}^d \mapsto \mathbb{R}$ that can only output a single scalar value to describe the safety of a simulation run. While this has its uses, in a typical autonomous driving setting, there are many different metrics of interest for a given scenario. In a simple case, consider the following scenario: an ego vehicle is following another vehicle on a city road. One can immediately come up with a few salient metrics for this scenario; for example, the ego vehicle should not collide with any other vehicles or objects in the scene, all traffic laws should be obeyed, and the ego should maintain a certain minimum safe distance from the vehicle it is following. There are many such metrics that can be formulated along the same lines, and there are specific equations computing these metrics given the dynamics of the system over time [37]. The three aforementioned metrics also implicitly have a *priority ordering*: it is most important that there is no collision, then obeying all traffic laws, then least important is the minimum safe distance. This chapter discusses the specification and falsification of such multi-dimensional objectives and some evaluation of these metrics.

The primary use case for these multi-dimensional objectives is in the setting of an active sampler; we are concerned with algorithms for falsifying as many of these objectives as possible while keeping in mind the priority ordering that may exist. At first glance, there are many different ways to formulate this problem mathematically; we discuss one possibility here that allows for easy integration with the architecture already present in VerifAI.

Censi et al. describe a way to specify objectives using a concept of *rulebooks* [9]. To flesh out this idea, we present some mathematical notations that we will use in this chapter:

- As before, let x denote a *feature vector* sampled by VerifAI, having dimension d.
- We slightly modify the definition of $\rho(x)$. It is now a function mapping $\mathbb{R}^d \mapsto \mathbb{R}^m$, where m is the number of metrics specified.

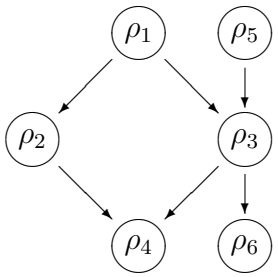

Figure 3.1: Example of a rulebook over six functions $\rho_1 ... \rho_6$ [9]

- Let $\mathcal{O} \subseteq \mathbb{R}^m$ denote the space of all possible outputs from $\rho(x)$.
- We use $\rho_j(x)$ to denote the j-th entry of the objective computed over the the sample x.
- We keep the notation of ρ^* as a threshold, but it is now a vector over $\mathbb{R}^m$ and ρ_j^* corresponds to the threshold value for ρ_j
- We use the operators $\preceq$ and $\succeq$ to relate objective vectors, i.e. $\rho(x_1) \succeq \rho(x_2)$ means that x_1 either violates metrics of higher importance than x_2 or they are exactly the same value. We use $\prec$ and $\succ$ as the strict versions of these operators, respectively.

In the single-dimensional case, there is a well-defined notion of comparison; for example, a sample x_1 such that $\rho(x_1) = -1$ is "more" of a counterexample than a sample x_2 having $\rho(x_2) = 1$. However, in the multi-dimensional case, this comparison is no longer well-defined. One way to get around this is to have a *total ordering* over the various components of the objective; that is, we can re-order the metrics such that ρ_1 is the most important component, ρ_2 the second most important, and so on, with ρ_m being the least important. In this case, we can definitively compare any two ρ values using a lexicographical ordering, i.e.

$$\rho(x_1) \succ \rho(x_2) \triangleq \exists k \, (\rho_k(x_1) < \rho_k(x_2) \wedge \forall k' < k, \; \rho_{k'}(x_1) = \rho_{k'}(x_2))$$

However, it is not always practical to create such a total ordering over the components of the objectives. Censi et al. estimate that a typical autonomous vehicle developer attempting to cover an urban driving setting will have on the order of several hundred different metrics [9], and it is not always clear when looking at pairs of objectives whether one is strictly more important than the other.

Specification of Objectives using Rulebooks

We present a form of multi-objective sampling that incorporates a rulebook $\mathcal{R}$ - which is defined as a directed graph where the nodes are components of the objective function ρ_j

and a directed edge from ρ_j to ρ_k means that ρ_j is more important than ρ_k. We denote this using the $>_R$ operator, i.e. $\rho_j >_R \rho_k$. Figure 3.1 shows an example rulebook where we have six components of the objective function $\rho_1 ... \rho_6$ [9]. In this example, we can make several inferences, such as ρ_1 is more important than ρ_3, ρ_3 is more important than ρ_4, and ρ_5 is more important than ρ_3. However, there are also many pairs of objective components that cannot be compared; for example ρ_1 and ρ_5. Because of these indeterminate incomparisons, the rulebook $\mathcal{R}$ only provides a *partial ordering* over the objective components. This makes it still unclear how to order samples keeping this in mind. We can define our $\succ$ operator as follows:

$$\rho(x_1) \succ \rho(x_2) \triangleq \forall i, (\rho_i(x_2) < \rho_i(x_1) \implies \exists j \neq i \mid \rho_j >_R \rho_i \wedge \rho_j(x_1) < \rho_j(x_2))$$

Intuitively, we can think of this partial ordering as preferring examples that have lower values of higher priority objectives; however, if there is any indeterminate or higher priority objective that is higher a result, the $\succ$ comparison does not hold. As an example, consider our rulebook from Figure 3.1. Let $\rho(x_1) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T$, and $\rho(x_2) = \begin{bmatrix} 1 & 1 & 2 & 1 & 0 & 1 \end{bmatrix}^T$. In this case we have $\rho(x_2) \succ \rho(x_1)$ because $\rho_5(x_2) < \rho_5(x_1)$, and even though $\rho_3(x_2) > \rho_3(x_1)$, $\rho_5 >_R \rho_3$ according to the rulebook, so the comparison of ρ_5 for the samples takes precedence.

Multi-Dimensional Objective Falsifcation

We saw in chapter 2 an example of how the cross-entropy sampler works on scalar objectives. In this section, we present several possible approaches to extend this functionality to vector-valued objectives that were tried before arriving at the final implementation. First, we consider a naïve approach with a slightly modified `update()` function, such as the following:

```
function update(bucket_value, rho):
    compute list of booleans b for whether each rho_j < rho_j*
    if b is the best possible counterexample:
        increase probability of bucket_value and re-normalize
```

This brings up several interesting points. First off, we compute a value b which is a boolean for each component of the objective vector representing whether or not that component was falsified, i.e. $b_j \iff \rho_j < \rho_j^*$. For the sake of consistency with scalar-objective active samplers, this is what ended up being used in the actual VerifAI implementation of a multi-dimensional objective sampler. Clearly, we can see that the "best possible counterexample" as defined by our rulebook is an x such that $\forall j, \rho_j(x) < \rho_j^*$, i.e. every objective is falsified. However, in almost any real-world use case, it is impossible to falsify every single objective, and thus this approach is unrealistic and will not learn anything over the course of the sampling process. Another possibility that was considered is to do several updates, using exponential decay when traversing through the graph. Pseudocode for this algorithm could be the following:

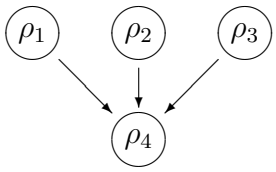

Figure 3.2: Example of a rulebook that refutes exponential dropoff counting scheme.

```
function update(bucket_value, rho):
    compute list of booleans b for whether each rho_j < rho_j*
    num_falsifications = 0
    initialize list skip which is false for every node in the rulebook
    for (node, depth) in a DFS traversal of the rulebook:
        if skip[node]:
            continue
        if not b[node]: // node is not falsified
            for each subnode in node's descendants in the rulebook:
                skip[subnode] = True
        else: // node is falsified
            num_falsifications += 1.0 / (2**depth)
    increase probability of bucket_value by num_falsifications times
    more than in the original cross-entropy sampler
    rescale probabilities to sum to 1
```

This works slightly better, but it is not difficult to create specific rulebooks where this algorithm breaks down pretty quickly. For example, consider the rulebook in Figure 3.2. For this rulebook, ρ_4 will have the highest weight of any node in the exponential dropoff counting scheme ($1/2 + 1/2 + 1/2 = 3/2$, whereas the others have weight 1), even though it has the lowest priority. This means that the sampler will actually bias over time towards samples that falsify ρ_4 first, which defeats the entire purpose of the rulebook. Such patterns as the above (more source nodes than sink nodes) could occur quite frequently throughout a given rulebook, so this is definitely a potential drawback of this approach. To avoid using these heuristics and mitigate some of the issues with the active samplers that are exacerbated in a multi-dimensional setting, we will discuss a few alternative sampling strategies.

The Epsilon-Greedy Sampler

Most of the samplers available in VerifAI focus either entirely on exploration or entirely on exploitation; while this worked well in practice for single-dimensional objectives, there are cases where the active samplers (such as cross-entropy) converge on one specific subregion of the search space very quickly and potentially miss out on diversity of counterexamples. For example, the cross-entropy sampler increases the probability of sampling from a bucket any time a counterexample is found in that bucket; therefore, if counterexample(s) are found in one specific bucket early on, a positive feedback loop ensues which could cause the sampler to sample from that bucket more and more without ever exploring any other buckets.

We first present a simple approach to balancing these trade-offs: the *epsilon-greedy* sampler. This algorithm is nearly identical to the cross-entropy sampler, with the difference being the introduction of a hyperparameter $\epsilon \in [0, 1]$. With probability ϵ, we sample uniform randomly across all buckets, and with probability $1 - \epsilon$, we sample from the current probability distribution computed over the buckets.

There has been much work on epsilon-greedy sampling in the reinforcement learning domain [32], where the goal is to balance exploration and exploitation over a state-action space, given some learned policy by the agent $\pi(s)$. This can be thought of as an equivalent where the action space is the set of buckets over which we sample, and the reward function $R(s, a)$ is only dependent on the action a as there is no notion of agent state in VerifAI sampler. In these cases, the value of ϵ is determined through empirical observation. Typically, it is important to decrease the value of ϵ over the course of the sampling process; this roughly corresponds to the increased confidence that we have in our learned probability distribution as we generate more samples. For decaying the value of epsilon, we typically use $\epsilon \sim 1/t$ [33].

The Multi-Armed Bandit Sampler

We present a more robust version of the cross-entropy sampler here called the *multi-armed bandit sampler*; the idea of this sampler is to balance the trade-off between exploitation and exploration. To understand the motivation for the sampler, we first look at the formulation of the multi-armed bandit problem. Consider a bandit which has multiple lotteries, or "arms", to choose from, each being a random variable offering a probabilistic reward. The bandit does not know ahead of time which arm gives the highest expected reward, and the idea here is to learn this information by efficiently sampling various arms, while also maximizing average earned reward during the sampling process. This problem can be modeled as minimizing expected regret, i.e. the difference between the expected earned reward over the sampling policy $\mathcal{A}$ and the expected earned reward of an optimal policy $\mathcal{A}^*$, which is to always choose the arm that gives the highest expected reward [34]:

$$\arg\min_{\mathcal{A}} R_n(\mathcal{A}^*) - R_n(\mathcal{A})$$

Carpentier et al. show in their landmark paper on the Upper Confidence Bound (UCB) Algorithm that this quantity can be minimized, subject to a confidence parameter δ, by the implementing the following sampling scheme [7]:

```
function multi_armed_bandit_sample():
    for i in range(N):
        compute quantity Q_j for each arm j
        sample from the arm corresponding to argmax(Q_j)
```

Where the quantity Q_j is dependent on the number of timesteps t, the number of times the arm j was sampled $T_j(t-1)$, the observed reward of arm j given by $\hat{\mu}_j$, and the confidence parameter δ:

$$Q_j = \hat{\mu}_j + \sqrt{\frac{2}{T_j(t-1)} \ln\left(\frac{1}{\delta}\right)}$$

Qualitatively, this works as a balance between exploitation of the reward distribution learned so far ($\hat{\mu}_j$), and exploration of seldom sampled buckets (the $\sqrt{\frac{2}{T_j(t-1)}}$ term in the expression for Q_j). We can easily see that this can be readily adapted to our cross-entropy sampler in VerifAI, taking $\hat{\mu}_j$ to be the proportion of counterexamples found in bucket j.

Computing $\hat{\mu}_j$ is a little more complex for the multi-dimensional objective case. Given the values of b that are computed during the state update, there is a lot of ambiguity in the way that a counterexample is defined since some metrics may be violated while others may not. To mitigate this, we present the following incremental algorithm which builds up counterexamples that falsify more and more objectives (according to the priority order) over time. The steps of this algorithm are as follows. This assumes that the sampler is responsible for generating a d-dimensional feature vector.

Setup

1. Split the range of each component of the feature vector into N buckets, as in the cross-entropy sampler.

2. Initialize matrix T of size $d \times N$ where T_{ij} will keep track of the number of times that bucket j was visited for variable x_i.

3. Initialize a dictionary c mapping each maximal counterexample found so far to a matrix c_b of size $d \times N$ where $c_{b,ij}$ counts how many times sampling bucket j for variable x_i resulted in the specific counterexample b.

4. Sample from each bucket once initially, updating c and T according to the update algorithm described below. The purpose of this is to avoid division by zero when computing Q, as $T_j(t-1) = 0$ at initialization [34].

Sampling

1. Compute a matrix $\hat{\mu}$ where $\hat{\mu}_{ij}$ represents the observed reward from sampling bucket j for variable i by taking $\sum_b c_{b,ij}$.
2. Compute a matrix Q based on the upper confidence bound formula above. For the confidence parameter, we use a time-dependent value of $\frac{1}{\delta} = t$.
3. To sample x_i, take the bucket $j^* = \arg\max_j Q_{ij}$. *Break ties uniformly at random.* This is a key step in the sampling process as it is frequently the case initially that several buckets will have the exact same Q_j value, so we need to avoid bias towards any specific bucket. Sample uniform randomly within the range represented by bucket j^*.

Updating Internal State

1. Given the objective vector value ρ, we compute our vector of booleans b as described above.
2. If b does not exist in the dictionary c and is among the set of maximal counterexamples found so far, i.e. $\forall b' \in c, b' \not\succ b$ as defined by the rulebook $\mathcal{R}$, add b as a key to the dictionary c and initialize its value as $0^{d \times N}$.
3. For any $b' \in c$ such that $b \succ b'$, remove b' from c.
4. Increment the count c_b at each position $c_{b,ij}$ for the bucket j sampled from variable x_i.

One interesting aspect of this algorithm that is worth noting is that the total number of counterexamples found in a given bucket can decrease over the course of the sampling process, as the set of maximal counterexamples is updated to represent the most recently found "best" counterexample.

Example of Incremental Multi-Dimensional Objective Falsification

Consider again the excerpt of a Scenic script from Figure 2.1. In this example, we instantiate two cars whose positions each contain a single continuous variable that needs to be sampled by VerifAI. Below is an example of a multi-objective VerifAI monitor that could be written to evaluate trajectories generated by Scenic.

```
def specification(traj):
    min_dist = np.inf
    N = len(traj)
    for i, val in enumerate(traj):
        obj1, obj2 = val
        min_dist = min(min_dist, obj1.distanceTo(obj2))
    angles = np.zeros((N - 1,))
    for i in range(1, N):
        t1, t2 = traj[i - 1], traj[i]
        ego_pos1, _ = t1
        ego_pos2, _ = t2
        v_ego = (ego_pos2 - ego_pos1) * 10
        angle = math.atan2(v_ego.y, v_ego.x)
        angles[i - 1] = angle
    rho = (min_dist - 5, (10 * math.pi / 180) - np.ptp(angles))
    return rho
```

In this case, the two objectives are that the ego vehicle should maintain some minimum distance from the other vehicle in the scene, and that it should drive straight without too much variation in its *heading* angle. We have no preference for one metric over the other; therefore, our rulebook $\mathcal{R}$ is a completely disconnected graph of 2 nodes. Let our threshold vector $\rho^* = \begin{bmatrix} 0 & 0 \end{bmatrix}$; this is why we normalize the values in the monitor above, such that the minimum distance required between the ego and other vehicles is 5 meters, and the maximum range allowed in the heading angle is 10°.

Using this example, we consider a possible execution of the incremental multi-dimensional falsification algorithm. For the sake of simplicity, we assume that the samples are given to us (not necessarily according to the UCB algorithm described above). We follow the state of the sampler and falsifier over the course of a few samples. A few notes about the algorithm:

- A *counterexample* is defined as a sample that produces a value of b that is not strictly worse than any of the highest priority examples b' found by the sampler so far. Therefore, it is possible for the number of counterexamples to decrease over the sampling process as better values of b are found.

- The count matrix is updated by incrementing the all indices (i, j) such that bucket j was used to sample the value x_i.

- The dictionary c contains all maximal values of b (according to the rulebook) found by the sampler so far.

Initial State of Sampler

Count matrix T : $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Counterexample dictionary c: {}

Sampling Iteration 1

Indices of Buckets Sampled: $\begin{bmatrix} 4 & 2 \end{bmatrix}$

$\rho(x) = \begin{bmatrix} -1 & 1 \end{bmatrix} \rightarrow b =$ (True, False)

Sampler State after Iteration 1

Count matrix T : $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

Counterexample dictionary c: {

(True, False): $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$

}

Total number of samples: 1

Total number of counterexamples: 1

Sampling Iteration 2

Indices of Buckets Sampled: $\begin{bmatrix} 1 & 2 \end{bmatrix}$

$\rho(x) = \begin{bmatrix} -1 & 1 \end{bmatrix} \rightarrow b =$ (True, False)

Sampler State after Iteration 2

Count matrix T : $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}$

Counterexample dictionary c: {

(True, False): $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 \end{bmatrix}$

}

Total number of samples: 2

Total number of counterexamples: 2

Sampling Iteration 3

Indices of Buckets Sampled: $\begin{bmatrix} 3 & 3 \end{bmatrix}$

$\rho(x) = \begin{bmatrix} -1 & -1 \end{bmatrix} \rightarrow b =$ (True, True)

Sampler State after Iteration 3

Count matrix T : $\begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 & 0 \end{bmatrix}$

Counterexample dictionary c: {

(True, True): $\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

}

Total number of samples: 3

Total number of counterexamples: 1

Discussion

As mentioned earlier, we can see that the total number of counterexamples has decreased over the course of the sampling process, because sample iteration 3 produced a sample that violated both metrics, which takes precedence over the first two samples (which were considered counterexamples up to that point). We now present a few properties of this algorithm.

Theorem 1 *The maximum possible size of the counterexample dictionary c grows on the order of* $\Theta\left(\frac{2^m}{\sqrt{m}}\right)$.

Proof. It is easy to see that the maximum possible number of boolean vectors b that we can compute over the metrics is 2^m, since b can be thought of as a bit vector of size m. This means that the tightest bound on the number of keys in c is no higher than $O(2^m)$. We can also show that given m metrics, if there is no preference ordering on these metrics, then for all $k \leq m$, all samples that violate exactly k metrics are incomparable to each other. That means we will have to potentially keep track of all of these counterexamples in c. This problem is equivalent to finding a set U of maximal cardinality containing subsets of a set A where no element of U is a subset of another element of U. By Sperner's theorem [22], this maximal cardinality is achieved when U contains all the subsets of A of size $k = \lfloor m/2 \rfloor$. This value is given by:

$$\binom{m}{\lfloor m/2 \rfloor} = \frac{m!}{\left(\frac{m}{2}!\right)^2}$$

Using Stirling's approximation [31], we have that:

$$x! \sim \sqrt{2\pi x}\left(\frac{x}{e}\right)^x$$

Where the $\sim$ operator means that the two functions have the same asymptotic growth. Plugging in, we get:

$$\begin{aligned}\binom{m}{\lfloor m/2 \rfloor} &\sim \frac{\sqrt{2\pi m}\left(\frac{m}{e}\right)^m}{\pi m\left(\frac{m}{2e}\right)^m} \\ &= \sqrt{\frac{2}{\pi m}}2^m = \Theta\left(\frac{2^m}{\sqrt{m}}\right)\end{aligned}$$

This means that the algorithm described above has nearly-exponential memory requirements in theory; however, in practice, the algorithm is usually much more efficient for the following reasons:

1. For a large m, the number of generated samples is several orders of magnitude lower than the theoretical upper bound on the number of counterexamples;
2. In the majority of cases, patterns arise in the generated samples such that specific sets of metrics are violated in certain regions of the search space. Therefore, the number of varied counterexamples that are generated is relatively small;

3. The rulebook $\mathcal{R}$ will typically have many edges describing the preference ordering over the nodes. This has the effect of creating more pairs of comparable counterexamples (b_1, b_2), which means that keys will more frequently be evicted from c. It is easy to see that if the rulebook describes a total ordering, there will be at most one counterexample in the dictionary at any given time.

We also present the following theorem, which follows from the incremental nature of the algorithm:

Theorem 2 *The incremental multi-armed bandit algorithm guarantees that at any given point in the sampling process, the regret metric $R_n(\mathcal{A}^*) - R_n(\mathcal{A})$ is being minimized.*

The UCB-1 algorithm shows this result for a scalar reward function [7]. We can look at the multi-dimensional case as a version of the original UCB-1 algorithm where we essentially restart the sampling process anytime a reward of a higher priority is discovered. Therefore, even though the optimization problem is constantly changing as better counterexamples are discovered, this incremental algorithm is a repeated application of UCB-1 that is constantly optimizing the regret bound for a given family of reward functions determined by the best counterexample(s) found so far.

3.2 Evaluation

We first present the same set of experiments as in Chapter 2, run with both the epsilon-greedy sampler and the multi-armed bandit sampler in a parallel setting. We document the number of counterexamples generated by each method, compared to the cross-entropy sampler and the Halton sampler that we tested in the previous section. The results are shown in Table 3.1.

For the evaluation of the multi-dimensional objective multi-armed bandit sampler, we used a Scenic program[1] that instantiates the ego vehicle, along with m adversarial vehicles at random positions with respect to a 4-way intersection and has all of them drive towards the intersection and either go straight or make a turn [18]. The monitor returns an m-dimensional vector where ρ_j the ego vehicle's distance in meters from adversarial vehicle j. Through empirical observations of the scenario runs, we set the threshold vector as $\rho_j^* = 5 \; \forall j$. Table 3.2 lists the results of running these experiments with a variety of rulebooks and once again using the serial and parallel falsification pipelines. The setup and hardware used for the experiments is the same as in Chapter 2. We use $\emptyset$ to denote a rulebook with no edges; this means there is no preference between the objectives. We use L_m to denote a linked list over the m objectives that represents a total ordering; i.e. it assigns the ordering $\rho_1 > \rho_2 > ... > \rho_m$. We also use G to represent the priority ordering shown in Figure 3.3.

[1]Source of Scenic programs available at https://github.com/findaheng/Scenic/blob/behavior_prediction/examples/carla/Behavior_Prediction/intersection/intersection_11.scenic.

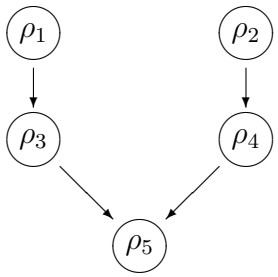

Figure 3.3: Rulebook configuration G used for experimentation.

For the sake of compactness, we represent the set of maximal counterexamples discovered by the falsifier as a bit vector; if bit j (1-indexed) is set, that means we have $\rho_j(x) < \rho_j^*$. Moreover, because of the difference in the priority of the best counterexamples found, the number of counterexamples returned by the falsifier may be lower for falsification runs that were actually better, as better counterexamples may be more rare in the feature space and may only occur later on during the sampling process.

Experiment Name	(**Counterexamples**/Total Samples)			
	Halton	Cross-Entropy	Epsilon-Greedy	Multi-Armed Bandit
Scenario 1	**258**/831	**358**/782	**313**/842	**323**/803
Scenario 2	**454**/852	**422**/798	**493**/891	**399**/791
Scenario 11	**322**/393	**337**/428	**329**/404	**326**/398
Scenario 12	**331**/448	**369**/463	**334**/441	**326**/435
Scenario 13	**77**/352	**159**/374	**153**/396	**123**/334
Scenario 15	**180**/330	**300**/334	**260**/345	**286**/341
Scenario 16	**627**/2059	**1491**/2017	**1345**/1884	**1412**/1889

Table 3.1: Comparison of epsilon-greedy and multi-armed bandit samplers with cross-entropy and Halton samplers.

We can also visually show the differences between the various samplers. Figure 3.4

demonstrates a Scenic program that partially recreates the 2017 Uber crash scenario mentioned in the introduction [28]. Figure 3.5 shows a comparison of the points sampled in a feature space described by a Scenic program for the cross-entropy, Halton, epsilon-greedy, and multi-armed bandit samplers. We plot the values of the three variables sampled by VerifAI, namely `DISTANCE_TO_INTERSECTION`, `UBER_SPEED`, and `HESITATION_TIME`, for each iteration. For the sake of computational simplicity, we ran the script which each sampler for 60 seconds using the Newtonian simulator. Visually, the multi-armed bandit sampler seems to find a middle ground between the cross-entropy sampler and the Halton sampler. The cross-entropy sampler zones in on a specific region of the feature space very quickly and barely tries anything outside that region, but has a counterexample rate of 93%. Meanwhile, the Halton sampler samples relatively uniformly over the entire feature space but finds lots of safe examples (the purple points), having a counterexample rate of only 71%. Epsilon-greedy with $\epsilon = 0.5$ uses random sampling for approximately half the samples and cross-entropy sampling the other half, resulting in a counterexample rate of 82%. The multi-armed bandit sampler seems to effectively benefit from active and passive sampling strategies, as it finds a relatively large region with lots of counterexamples but it also tries a lot of points throughout the feature space as a result of the exploration bonus term, ending with a final counterexample rate of 89% – only slightly lower than cross-entropy.

Configuration	Rulebook	Samples	Counterexamples	Maximal Counterexample(s) Found
Serial, $m = 5$	$\emptyset$	122	22	11001, 10101, 10011, ...
Parallel, $m = 5$	$\emptyset$	638	16	11110, 11101, 11011, 10111, 01111
Serial, $m = 5$	L_5	126	1	11110
Parallel, $m = 5$	L_5	613	1	11111
Serial, $m = 5$	G	130	3	11101, 11011
Parallel, $m = 5$	G	613	2	11111

Table 3.2: Comparison of different rulebooks and the counterexamples found for each.

```
1
2  # Parameters of the scenario.
3  param DISTANCE_TO_INTERSECTION = VerifaiRange(-20, -10)
4  param HESITATION_TIME = VerifaiRange(0, 10)
5  param UBER_SPEED = VerifaiRange(10, 20)
6
7  # Ego vehicle just follows the trajectory specified later on.
8  behavior EgoBehavior(trajectory):
9      do FollowTrajectoryBehavior(trajectory=trajectory,
10     target_speed=globalParameters.UBER_SPEED)
11     terminate
12
13 # Crossing car hesitates for a certain amount of time
14 # before starting to turn.
15 behavior CrossingCarBehavior(trajectory):
16     while simulation().currentTime < globalParameters.HESITATION_TIME:
17         wait
18     do FollowTrajectoryBehavior(trajectory = trajectory)
19     terminate
20
21 # Find all 4-way intersections and set up
22 # trajectories for each vehicle.
23 ...
24
25 # Spawn each vehicle in the middle of its starting lane.
26 uberSpawnPoint = startLane.centerline[-1]
27 crossingSpawnPoint = otherLane.centerline[-1]
28
29 ego = Car following roadDirection from uberSpawnPoint for
30 globalParameters.DISTANCE_TO_INTERSECTION,
31         with behavior EgoBehavior(trajectory = ego_trajectory)
32
33 crossing_car = Car at crossingSpawnPoint,
34                  with behavior
35                  CrossingCarBehavior(crossing_car_trajectory)
36
```

Figure 3.4: A Scenic script partially recreating the Uber crash scenario.

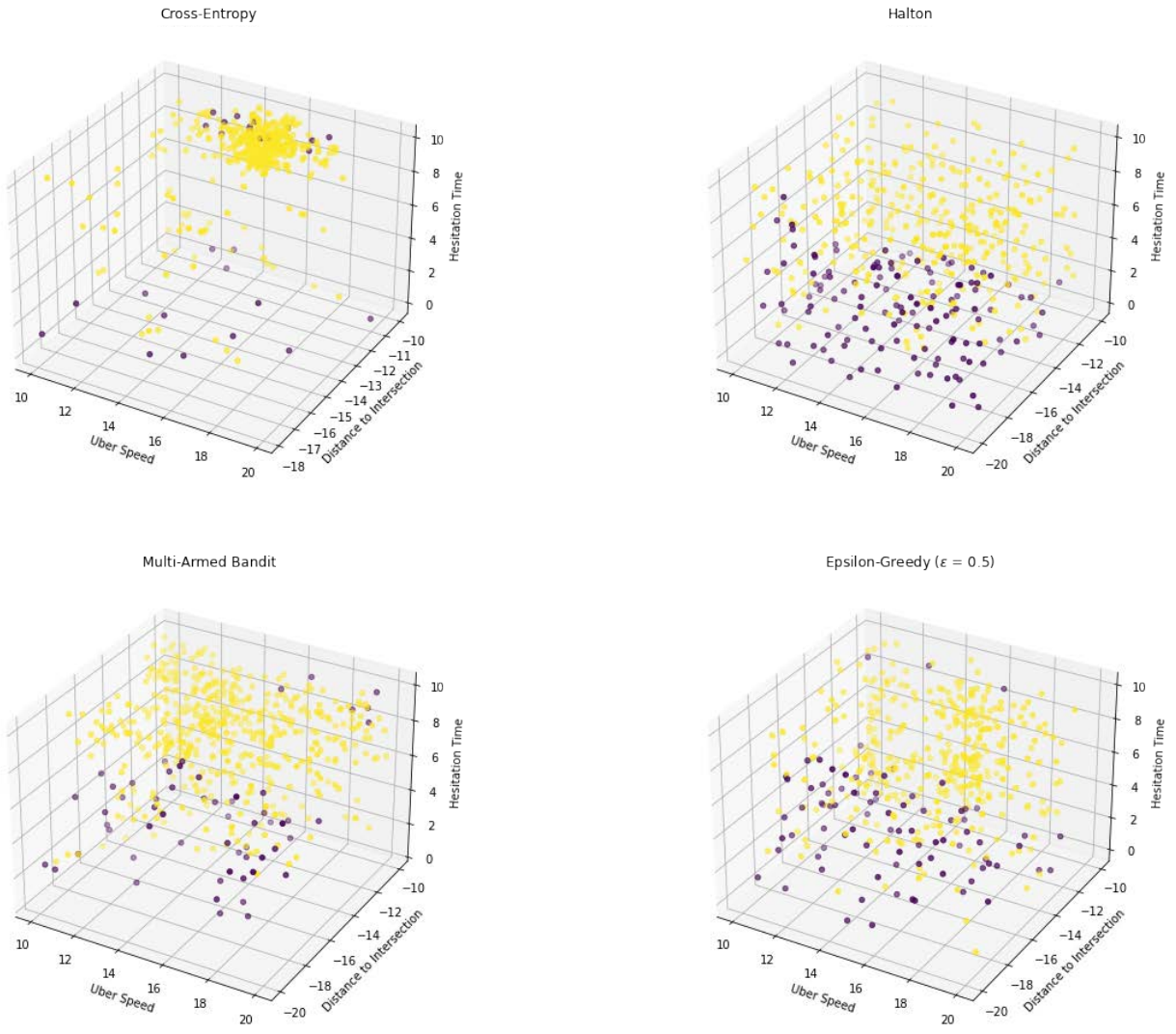

Figure 3.5: Comparison of points sampled in feature space for various samplers. Yellow points are counterexamples and purple points are safe examples.

Chapter 4

Conclusion

With autonomous vehicles becoming more prevalent on the road, it is increasingly important to verify AV systems before they hit the road, to ensure the safety of people both in the ego vehicle and in the surrounding environment. The extensions we have made to Scenic and VerifAI appear to be increasing efficiency in finding counterexamples, and will be useful in the falsification of multiple metrics at once. These contributions appear to be a good start towards a more effective sampling-based falsifier, and below we present some potential ideas on how to further the ideas we have presented to provide more functionality.

4.1 Future Work

Random Linearization

One idea that could be interesting to test with the rulebook implementation is to perform "random linearizations"; that is, choose a random topological sort of the directed acyclic graph $\mathcal{R}$ and impose that as a total ordering over the objectives. This is promising as it would allow for straightforward falsification as there would only ever be one highest counterexample found so far. It remains to be shown in experimentation whether this would empirically work as well as an incremental algorithm over the original rulebook for sampling, and it is also an open question as to what metrics would be used to benchmark the performance of these algorithms.

There are also few potential theoretical issues that need to be dealt with in this heuristic. In general, the problem of counting the number of topological sorts of a DAG is NP-hard [6]. This means that sampling uniform randomly from all topological sorts may not be a tractable problem in many cases. Therefore, a necessary step of this process would be to come up with an approximation algorithm to sample topological sorts in a way that is as close to uniform as possible. Another problem may be that the linearization will impose ordering of pairwise objectives that was unintended by the user. Because of this, the sampling algorithm may bias towards one objective while ignoring others that may have originally been considered

equally important. One way to potentially mitigate this is to re-sample the linearized graph with some certain frequency to help smooth out the bias towards specific objectives over time.

Covariance Analysis of Features

The current samplers implemented in VerifAI optimize the sampled values of each feature individually with the assumption that they are all pairwise independent. In practice, however, this is generally not the case. Consider for example a simple scenario with an ego vehicle and an adversary heading toward an intersection from perpendicular directions (without any collision avoidance for simplicity). We can imagine that in such a scenario, a crash will happen when the two cars are going at roughly the same speed. If we encode this in VerifAI as a vector $x \in \mathbb{R}^2$, we can imagine that in the majority of counterexamples, we will have $x_1 \approx x_2$. However, this is not a relationship that can be learned by the active samplers in VerifAI as the assumption is that each feature individually contributes to the value of $\rho(x)$ without any interaction with other features. Therefore, one possible way to deal with this would be to perform covariance analysis on the returned counterexamples from a falsifier run, and use that to determine if any highly correlated variables should be optimized together during sampling.

Real-World Testing

Scenic and VerifAI are tools meant to help guide testing in simulation; however, it is crucial to run a comprehensive set of test cases in the real world as well. Such testing can be expensive, time-consuming, and prone to error; it would therefore be a big step forward if the multi-objective sampling in VerifAI can be used to help determine what scenarios are most challenging for an AV system and place more of an emphasis on those when performing field testing to make the process more effective. Some progress has already been made on this front, using VerifAI and Scenic in simulation to define test cases that can be run in the real world [16].

Bibliography

[1] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[2] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Parosh Aziz Abdulla and K. Rustan M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254–257. ISBN: 978-3-642-19835-9.

[3] *ASAM OpenDRIVE*. URL: https://www.asam.net/standards/detail/opendrive/.

[4] BerkeleyLearnVerify. *BerkeleyLearnVerify/Scenic* . URL: https://github.com/BerkeleyLearnVerify/Scenic.

[5] BerkeleyLearnVerify. *BerkeleyLearnVerify/VerifAI* . URL: https://github.com/BerkeleyLearnVerify/VerifAI.

[6] Graham Brightwell and Peter Winkler. "Counting Linear Extensions is #P-Complete". In: *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*. STOC '91. New Orleans, Louisiana, USA: Association for Computing Machinery, 1991, pp. 175–181. ISBN: 0897913973. DOI: 10.1145/103418.103441. URL: https://doi.org/10.1145/103418.103441.

[7] Alexandra Carpentier, Alessandro Lazaric, Mohammad Ghavamzadeh, Rémi Munos, and Peter Auer. "Upper-Confidence-Bound Algorithms for Active Learning in Multi-armed Bandits". In: *Algorithmic Learning Theory*. Ed. by Jyrki Kivinen, Csaba Szepesvári, Esko Ukkonen, and Thomas Zeugmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 189–203. ISBN: 978-3-642-24412-4.

[8] Luis I. Reyes Castro, Pratik Chaudhari, Jana Tumova, Sertac Karaman, Emilio Frazzoli, and Daniela Rus. "Incremental Sampling-based Algorithm for Minimum-violation Motion Planning". In: *CoRR* abs/1305.1102 (2013). arXiv: 1305.1102. URL: http://arxiv.org/abs/1305.1102.

[9] Andrea Censi, Konstantin Slutsky, Tichakorn Wongpiromsarn, Dmitry S. Yershov, Scott Pendleton, James Guo Ming Fu, and Emilio Frazzoli. "Liability, Ethics, and Culture-Aware Behavior Specification using Rulebooks". In: *CoRR* abs/1902.09355 (2019). arXiv: 1902.09355. URL: http://arxiv.org/abs/1902.09355.

[10] C. J. CLOPPER and E. S. PEARSON. "THE USE OF CONFIDENCE OR FIDUCIAL LIMITS ILLUSTRATED IN THE CASE OF THE BINOMIAL". In: *Biometrika* 26.4 (Dec. 1934), pp. 404–413. ISSN: 0006-3444. DOI: 10.1093/biomet/26.4.404. eprint: https://academic.oup.com/biomet/article-pdf/26/4/404/823407/26-4-404.pdf. URL: https://doi.org/10.1093/biomet/26.4.404.

[11] Joseph Cralley, Ourania Spantidi, Bardh Hoxha, and Georgios Fainekos. "TLTk: A Toolbox for Parallel Robustness Computation of Temporal Logic Specifications". In: *Runtime Verification*. Ed. by Jyotirmoy Deshmukh and Dejan Ničković. Cham: Springer International Publishing, 2020, pp. 404–416. ISBN: 978-3-030-60508-7.

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

[13] 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.

[14] 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 annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. June 2019.

[15] Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. "Scenic: A Language for Scenario Specification and Data Generation". In: *CoRR* abs/2010.06580 (2020). arXiv: 2010.06580. URL: https://arxiv.org/abs/2010.06580.

[16] 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.

[17] Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. "Adversarial Machine Learning". In: *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*. AISec '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 43–58. ISBN: 9781450310031. DOI: 10.1145/2046684.2046692. URL: https://doi.org/10.1145/2046684.2046692.

[18] Francis Indaheng. *A Scenario-Based Approach to Testing Autonomous Vehicle Behavior Prediction Models in Simulated Environments*. 2021.

[19] Edward Kim, Divya Gopinath, Corina S. Pasareanu, and Sanjit A. Seshia. "A Programmatic and Semantic Approach to Explaining and Debugging Neural Network Based Object Detectors". In: *CoRR* abs/1912.00289 (2019). arXiv: 1912.00289. URL: http://arxiv.org/abs/1912.00289.

[20] Philip Koopman and Michael Wagner. "Challenges in Autonomous Vehicle Testing and Validation". In: *SAE International Journal of Transportation Safety* 4.1 (2016), pp. 15–24. ISSN: 23275626, 23275634. URL: http://www.jstor.org/stable/26167741.

[21] *LG and Unity collaborate on autonomous vehicle simulation*. URL: https://swsolutions.lge.com/insights/news/lgsvl-simulator.

[22] L. D. Meshalkin. "Generalization of Sperner's Theorem on the Number of Subsets of a Finite Set". In: *Theory of Probability & Its Applications* 8.2 (1963), pp. 203–204. DOI: 10.1137/1108023. eprint: https://doi.org/10.1137/1108023. URL: https://doi.org/10.1137/1108023.

[23] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. "Ray: A Distributed Framework for Emerging AI Applications". In: *CoRR* abs/1712.05889 (2017). arXiv: 1712.05889. URL: http://arxiv.org/abs/1712.05889.

[24] Wassim G Najm, John D Smith, and Mikio Yanagisawa. *Pre-Crash Scenario Typology for Crash Avoidance Research*. Apr. 2007. URL: https://www.nhtsa.gov/sites/nhtsa.gov/files/pre-crash_scenario_typology-final_pdf_version_5-2-07.pdf.

[25] Matteo Pirotta, Simone Parisi, and Marcello Restelli. "Multi-Objective Reinforcement Learning with Continuous Pareto Frontier Approximation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1 (Feb. 2015). URL: https://ojs.aaai.org/index.php/AAAI/article/view/9617.

[26] Adnan Qayyum, Muhammad Usama, Junaid Qadir, and Ala I. Al-Fuqaha. "Securing Connected & Autonomous Vehicles: Challenges Posed by Adversarial Machine Learning and The Way Forward". In: *CoRR* abs/1905.12762 (2019). arXiv: 1905.12762. URL: http://arxiv.org/abs/1905.12762.

[27] Rasmaxim. *rasmaxim/pygame-car-tutorial*. URL: https://github.com/rasmaxim/pygame-car-tutorial.

[28] George Sandeman. *Uber pulls its self-driving cars from the road after accident left autonomous vehicle on its side*. Mar. 2017. URL: https://www.thesun.co.uk/news/3181916/uber-pulls-its-self-driving-cars-from-the-road-after-accident-left-autonomous-vehicle-on-its-side/.

[29] Sriram Sankaranarayanan and Georgios Fainekos. "Falsification of Temporal Properties of Hybrid Systems Using the Cross-Entropy Method". In: *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '12. Beijing, China: Association for Computing Machinery, 2012, pp. 125–134. ISBN: 9781450312202. DOI: 10.1145/2185632.2185653. URL: https://doi.org/10.1145/2185632.2185653.

[30] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. "Towards Verified Artificial Intelligence". In: *ArXiv e-prints* (July 2016). arXiv: 1606.08514.

[31] *Stirling's Approximation*. URL: https://mathworld.wolfram.com/StirlingsApproximation.html.

[32] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[33] *The Epsilon-Greedy Algorithm*. Dec. 2017. URL: https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/.

[34] *The Upper Confidence Bound Algorithm*. Sept. 2016. URL: https://banditalgs.com/2016/09/18/the-upper-confidence-bound-algorithm/.

[35] *Traffic Manager*. URL: https://carla.readthedocs.io/en/latest/adv_traffic_manager/#multisimulation.

[36] *Welcome to VerifAI's documentation!* URL: https://verifai.readthedocs.io/en/latest/?badge=latest.

[37] Jeffrey Wishart, Steven Como, Maria Elli, Brendan Russo, Jack Weast, Niraj Altekar, and Emmanuel James. "Driving Safety Performance Assessment Metrics for ADS-Equipped Vehicles". In: Apr. 2020. DOI: 10.4271/2020-01-1206.