

UC Berkeley

UC Berkeley Previously Published Works

Title

Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World

Permalink

<https://escholarship.org/uc/item/86q4p5p8>

ISBN

978-1-7281-4149-7

Authors

Fremont, Daniel J
Kim, Edward
Pant, Yash Vardhan
et al.

Publication Date

2020

Peer reviewed

Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VERIFAI

Daniel J. Fremont^{1,2}, Johnathan Chiu²,
Dragos D. Margineantu³, Denis Osipychiev³, and Sanjit A. Seshia²

¹ University of California, Santa Cruz, USA

² University of California, Berkeley, USA

³ Boeing Research & Technology, Seattle, USA

Abstract. We demonstrate a unified approach to rigorous design of safety-critical autonomous systems using the VERIFAI toolkit for formal analysis of AI-based systems. VERIFAI provides an integrated toolchain for tasks spanning the design process, including modeling, falsification, debugging, and ML component retraining. We evaluate all of these applications in an industrial case study on an experimental autonomous aircraft taxiing system developed by Boeing, which uses a neural network to track the centerline of a runway. We define runway scenarios using the SCENIC probabilistic programming language, and use them to drive tests in the X-Plane flight simulator. We first perform falsification, automatically finding environment conditions causing the system to violate its specification by deviating significantly from the centerline (or even leaving the runway entirely). Next, we use counterexample analysis to identify distinct failure cases, and confirm their root causes with specialized testing. Finally, we use the results of falsification and debugging to retrain the network, eliminating several failure cases and improving the overall performance of the closed-loop system.

Keywords: Falsification · Automated testing · Debugging · Simulation · Autonomous systems · Machine learning.

1 Introduction

The expanding use of machine learning (ML) in safety-critical applications has led to an urgent need for rigorous design methodologies that can ensure the reliability of systems with ML components [15,17]. Such a methodology would need to provide tools for *modeling* the system, its requirements, and its environment, *analyzing* a design to find failure cases, *debugging* such cases, and finally *synthesizing* improved designs.

The VERIFAI toolkit [1] provides a unified framework for all of these design tasks, based on a simple paradigm: simulation driven by formal models and specifications. The top-level architecture of VERIFAI is shown in Fig. 1. We first define an *abstract feature space* describing the environments and system configurations of interest, either by explicitly defining parameter ranges or using the SCENIC probabilistic environment modeling language [6]. VERIFAI then generates concrete tests by searching this space, using a variety of algorithms ranging from random sampling to global optimization techniques. Finally, we simulate the system for each test, monitoring the satisfaction or violation of a system-level specification; the results of each test are used to guide further

search, and any violations are recorded in a table for automated analysis (e.g. clustering) or visualization. This architecture enables a wide range of use cases, including falsification, fuzz testing, debugging, data augmentation, and parameter synthesis; Dreossi et al. [1] demonstrated all of these applications individually through several small case studies.

In this paper, we provide an *integrated* case study, applying VERIFAI to a complete design flow for a large, realistic system from industry: TaxiNet, an experimental autonomous aircraft taxiing system developed by Boeing for the DARPA Assured Autonomy project. This system uses a neural network to estimate the aircraft’s position from a camera image; a controller then steers the plane to track the centerline of the runway. The main requirement for TaxiNet, provided by Boeing, is that it keep the plane within 1.5 m of the centerline; we formalized this as a specification in Metric Temporal Logic (MTL) [11]. Verifying this specification is difficult, as the neural network must be able to handle the wide range of images resulting from different lighting conditions, changes in runway geometry, and other disturbances such as tire marks on the runway.

Our case study illustrates a complete iteration of the design flow for TaxiNet, analyzing and debugging an existing version of the system to inform an improved design. Specifically, we demonstrate:

1. Modeling the environment of the aircraft using the SCENIC language.
2. Falsifying an initial version of TaxiNet, finding environment conditions under which the aircraft significantly deviates from the centerline.
3. Analyzing counterexamples to identify distinct failure cases and diagnose potential root causes.
4. Testing the system in a targeted way to confirm these root causes.
5. Designing a new version of the system by retraining the neural network based on the results of falsification and debugging.
6. Validating that the new system eliminates some of the failure cases in the original system and has higher overall performance.

Following the procedure above, we were able to find several scenarios where TaxiNet exhibited unsafe behavior. For example, we found the system could not properly handle intersections between runways. More interestingly, we found that TaxiNet could get confused when the shadow of the plane was visible, which only occurred during certain times of day and weather conditions. We stress that these types of failure cases are meaningful counterexamples that could easily arise in the real world, unlike pixel-level adversarial examples [8]; we are able to find such cases because VERIFAI searches through a space of *semantic* parameters [3]. Furthermore, these counterexamples are *system-level*, demonstrating undesired behavior from the complete system rather than simply its ML component. Finally, our work differs from other works on validation of

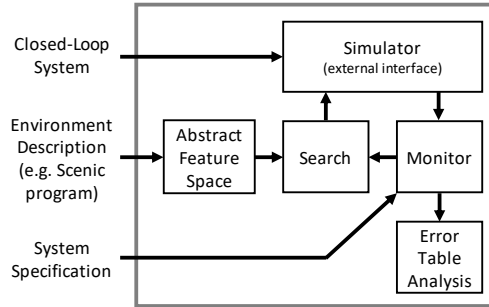


Fig. 1. Architecture of VERIFAI.

cyber-physical systems with ML components (e.g. [19]) in that we address a broader range of design tasks (including debugging and retraining as well as testing) and also allow designers to *guide* search by encoding domain knowledge using SCENIC.

For our case study, we extend VERIFAI in two ways. First, we interface the toolkit to the X-Plane flight simulator [12] in order to run closed-loop simulations of the entire system, with X-Plane rendering the camera images and simulating the aircraft dynamics. More importantly, we extend the SCENIC language to allow it to be used in combination with VERIFAI’s active sampling techniques. Previously, as in any probabilistic programming language, a SCENIC program defined a fixed distribution [6]; while adequate for modeling particular scenarios, this is incompatible with active sampling, where we change how tests are generated over time in response to feedback from earlier tests. To reconcile these two approaches, we extend SCENIC with *parameters* that are assigned by an external sampler. This allows us to continue to use SCENIC’s convenient syntax for modeling, while now being able to use not only random sampling but optimization or other algorithms to search the parameter space.

Adding parameters to SCENIC enables important new applications. For example, in the design flow we described above, after finding through testing some rare event which causes a failure, we need to generate a dataset of such failures in order to retrain the ML component. Naïvely, we would have to manually write a new SCENIC program whose distribution was concentrated on these rare events (as was done in [6]). With parameters, we can simply take the generic SCENIC program we used for the initial testing, and use VERIFAI’s cross-entropy sampler [1,14] to automatically converge to such a distribution [16]. Alternatively, if we have an intuition about where a failure case may lie, we can use SCENIC to encode this domain knowledge as a *prior* for cross-entropy sampling, helping the latter to find failures more quickly.

In summary, the novel contributions of this paper are:

- The first demonstration on an industrial case study of an integrated toolchain for falsification, debugging, and retraining of ML-based autonomous systems.
- An interface between VERIFAI and the X-Plane flight simulator.
- An extension of the SCENIC language with parameters, and a demonstration using it in conjunction with cross-entropy sampling to learn a SCENIC program encoding the distribution of failure cases.

We begin in Sec. 2 with a discussion of our extension of SCENIC with parameters and our X-Plane interface. Section 3 presents the experimental setup and results of our case study, and we close in Sec. 4 with some conclusions and directions for future work.

2 Extensions of VERIFAI

SCENIC with Parameters. To enable search algorithms other than random sampling to be used with SCENIC we extend the language with a concept of *external parameters* assigned by an *external sampler*. A SCENIC program can specify an external sampler to use; this sampler will define the allowed types of parameters, which can then be used in the program in place of any distribution. The default external sampler provides access to the VERIFAI samplers and defines parameter types corresponding to VERIFAI’s

continuous and discrete ranges. Thus for example one could write a SCENIC program which picks the colors of two cars randomly according to some realistic distribution, but chooses the distance between them using VERIFAI’s Bayesian Optimization sampler.

The semantics of external parameters is simple: when sampling from a SCENIC program, the external sampler is first queried to provide values for all the parameters; the program is then equivalent to one without parameters, and can be sampled as usual⁴.

X-Plane Interface. Our interface between X-Plane and VERIFAI uses the latter’s client-server architecture for communicating with simulators. The server runs inside VERIFAI, taking each generated feature vector and sending it to the client. The client runs inside X-Plane and calls its APIs to set up and execute the test, reporting back information needed to monitor the specifications. For our client, we used X-Plane Connect [18], an X-Plane plugin providing access to X-Plane’s “datarefs”. These are named values which represent simulator state, e.g., positions of aircraft and weather conditions. Our interface exposes all datarefs to SCENIC, allowing arbitrary distributions to be placed on them. We also set up the SCENIC coordinate system to be aligned with the runway, performing the appropriate conversions to set the raw position datarefs.

3 TaxiNet Case Study

3.1 Experimental Setup

TaxiNet’s neural network estimates the aircraft’s position from a camera image; the camera is mounted on the right wing and faces forward. Example images are shown in Fig. 2. From such an image, the network estimates the *cross-track error (CTE)*, the left-right offset of the plane from the centerline, and the *heading error (HE)*, the angular offset of the plane from directly down the centerline. These estimates are fed into a handwritten controller which outputs (the equivalent of) a steering angle for the plane.

The Boeing team provided the Berkeley team with an initial version of TaxiNet without describing which images were used to train it. In this way, the Berkeley team were not aware in advance of potential gaps in the training set and corresponding potential failure cases⁵. For retraining experiments, the same sizes of training and validation sets were used as for the original model, as well as identical training hyperparameters.

The semantic feature space defined by our SCENIC programs and searched by VERIFAI was 6-dimensional, made up of the following parameters⁶:

⁴ One complication arises because SCENIC uses rejection sampling to enforce constraints: if a sample is rejected, what value should be returned to active samplers that expect feedback, e.g. a cross-entropy sampler? By default we return a special value indicating a rejection occurred.

⁵ After drawing conclusions from initial runs of all the experiments, the Berkeley team were informed of the training parameters and trained their own version of TaxiNet locally, repeating the experiments. This was done in order to ensure that minor differences in the training/testing platforms at Boeing and Berkeley did not affect the results (which was in fact qualitatively the case). All numerical results and graphs use data from this second round of experiments.

⁶ We originally had additional parameters controlling the position and appearance of a tire mark superimposed on the runway (using a custom X-Plane plugin to do such rendering), but deleted the tire mark for simplicity after experiments showed its effect on TaxiNet was negligible.

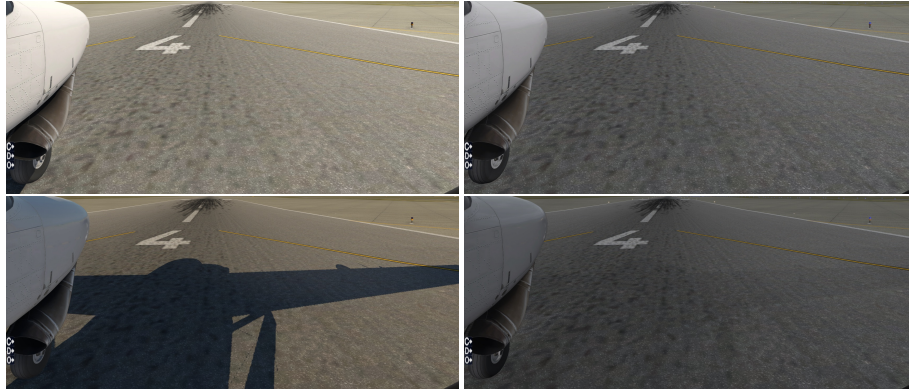


Fig. 2. Example input images to TaxiNet, rendered in X-Plane. Left/right = clear/cloudy weather. Top/bottom = 12 pm / 4 pm.

- the initial position and orientation of the aircraft (in 2D, on the runway);
- the type of clouds, out of 6 discrete options ranging from clear to stormy;
- the amount of rain, as a percentage, and
- the time of day.

Given values for these parameters from VERIFAI, the test protocol we used in all of our experiments was identical: we set up the initial condition described by the parameters, then simulated TaxiNet controlling the plane for 30 seconds.

The main requirement for TaxiNet provided by Boeing was that it should always track the centerline of the runway to within 1.5 m. For many of our experiments we created a greater variety of test scenarios by allowing the plane to start up to 8 m off of the centerline: in such cases we required that the plane approach within 1.5 m of the centerline within 10 seconds and then stay there for the remainder of the simulation. We formalized these two specifications as MTL formulas φ_{always} and $\varphi_{\text{eventually}}$ respectively:

$$\varphi_{\text{always}} = \square(\text{CTE} \leq 1.5) \quad \varphi_{\text{eventually}} = \diamond_{[0,10]}\square(\text{CTE} \leq 1.5)$$

While both of these specifications are true/false properties, VERIFAI uses a continuous quantity ρ called the *robustness* of an MTL formula [4]. Its crucial property is that $\rho \geq 0$ when the formula is satisfied, while $\rho \leq 0$ when the formula is violated, so that ρ provides a metric of *how close* the system is to violating the property. The exact definition of ρ is not important here, but as an illustration, for φ_{always} it is (the negation of) the greatest deviation beyond the allowed 1.5 m achieved over the whole simulation.

For additional experimental results, see the Appendix of the full version [5].

3.2 Falsification

In our first experiment, we searched for conditions in the nominal operating regime of TaxiNet which cause it to violate $\varphi_{\text{eventually}}$. To do this, we wrote a SCENIC program S_{falsif} modeling that regime, shown in Fig. 3. We first place a uniform distribution on

```

# Time of day: from 6 am to 6 pm. (+8 to get GMT, as used by X-Plane)
param zulu_time = ((6, 18) + 8) * 60 * 60

# Rain: 1/3 of the time. Clouds: rain requires types 3-5; otherwise 0-5.
clouds_and_rain = Options({
  tuple([Uniform(0, 1, 2, 3, 4, 5), 0]): 2, # no rain
  tuple([Uniform(3, 4, 5), (0.25, 1)]): 1 # 25% to 100% rain
})
param cloud_type = clouds_and_rain[0], rain_percent = clouds_and_rain[1]

# Plane: up to 8 m left/right, 2000 m down the runway, 30° left/right.
ego = Plane at (-8, 8) @ (0, 2000),
          facing (-30, 30) deg

```

Fig. 3. Generic SCENIC program $\mathcal{S}_{\text{falsif}}$ used for falsification and retraining.

time of day between 6 am and 6 pm local time (approximate daylight hours). Next, we determine the weather. Since only some of the cloud types are compatible with rain, we put a joint distribution on them: with probability $2/3$, there is no rain, and any cloud type is equally likely; otherwise, there is a uniform amount of rain between 25% and 100%⁷, and we allow only cloud types consistent with rain. Finally, we position the plane uniformly up to 8 m left or right of the centerline, up to 2000 m down the runway, and up to 30° off of the centerline. These ranges ensured that (1) the plane began on the runway and stayed on it for the entire simulation when tracking succeeded, and (2) it was always possible to reach the centerline within 10 seconds and so satisfy $\varphi_{\text{eventually}}$.

However, it was quite easy to find falsifying initial conditions within this scenario. We simulated over 4,000 runs randomly sampled from $\mathcal{S}_{\text{falsif}}$, and found many counterexamples: in only 55% of the runs did TaxiNet satisfy $\varphi_{\text{eventually}}$, and in 9.1% of runs, the plane left the runway entirely. This showed that TaxiNet’s behavior was problematic, but did not explain *why*. To answer that question, we analyzed the data VERIFAI collected during falsification, as we explain next.

3.3 Error Analysis and Debugging

VERIFAI builds a table which stores for each run the point sampled from the abstract feature space and the resulting robustness value ρ (see Sec. 3.1) for the specification. The table is compatible with the *pandas* data science library [13], making visualization easy. While VERIFAI contains algorithms for automatic analysis of the table (e.g., clustering and Principal Component Analysis), we do not use them here since the parameter space was low-dimensional enough to identify failure cases by direct visualization.

We began by plotting TaxiNet’s performance as a function of each of the parameters in our falsification scenario. Several parameters had a large impact on performance:

- **Time of day:** Figure 4 plots ρ vs. time of day, each orange dot representing a run during falsification; the red line is their median, using 30-minute bins (ignore the blue dots for now). Note the strong time-dependence: for example, TaxiNet works well in the late morning (almost all runs having $\rho > 0$ and so satisfying $\varphi_{\text{eventually}}$) but consistently fails to track the centerline in the early morning.

⁷ The 25% lower bound is because we observed that X-Plane seemed to only render rain at all when the rain fraction was around that value or higher.

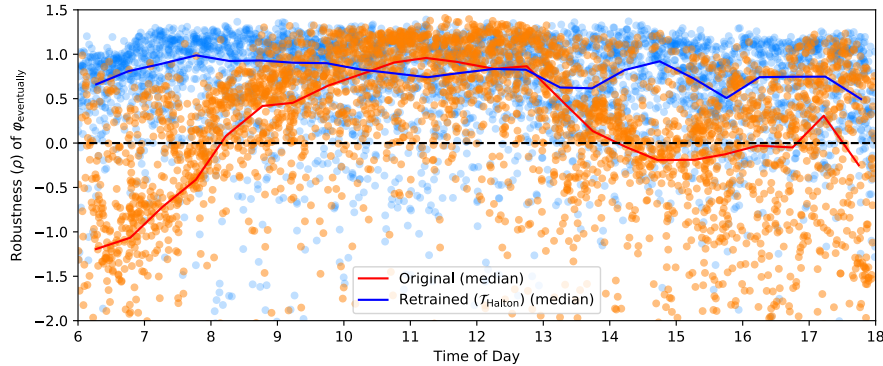


Fig. 4. Performance of TaxiNet as a function of time of day, before and after retraining.

- **Clouds:** Figure 5 shows the median performance curves (as in Fig. 4) for 3 of X-Plane’s cloud types: no clouds, moderate “overcast” clouds, and dark “stratus” clouds. Notice that at 8 am TaxiNet performs much worse with stratus clouds than no clouds, while at 2 pm the situation is reversed. Performance also varies quite irregularly when there are no clouds — we will analyze why this is the case shortly.
- **Distance along the runway:** The green data in Fig. 6 show performance as a function of how far down the runway the plane starts (ignore the orange/purple data for now). TaxiNet behaves similarly along the whole length of the runway, except around 1350–1500 m, where it veers completely off of the runway ($\rho \approx -30$). Consulting the airport map, we find that another runway intersects the one we tested with at approximately 1450 m. Images from the simulations show that at this intersection, both the centerline and edge markings of our test runway are obscured.

These visualizations identify several problematic behaviors of TaxiNet: consistently poor performance in the early morning, irregular performance at certain times depending on clouds, and an inability to handle runway intersections. The first and last of these are easy to explain as being due to dim lighting and obscured runway markings. The cloud issue is less clear, but VERIFAI can help us to debug it and identify the root cause.

Inspecting Fig. 5 again, observe that performance at 2–3 pm with no clouds is poor. This is surprising, since under these conditions the runway image is bright and clear; the brightness itself is not the problem, since TaxiNet does very well at the brightest time, noon. However, comparing images from a range of times, we noticed another difference: shortly after noon, the plane’s shadow enters the frame, and moves across the image over the course of the afternoon. Furthermore, the shadow is far less visible under cloudy conditions (see Fig. 2). Thus, we hypothesized that TaxiNet might be confused by the strong shadows appearing in the afternoon when there are no clouds.

To test this hypothesis, we wrote a new SCENIC scenario with no clouds, varying only the time of day; we used VERIFAI’s Halton sampler [9] to get an even spread of times with relatively few samples. We then ran two experiments: one with our usual test protocol, and one where we disabled the rendering of shadows in X-Plane. The results are shown in Fig. 7: as expected, in the normal run there are strong fluctuations

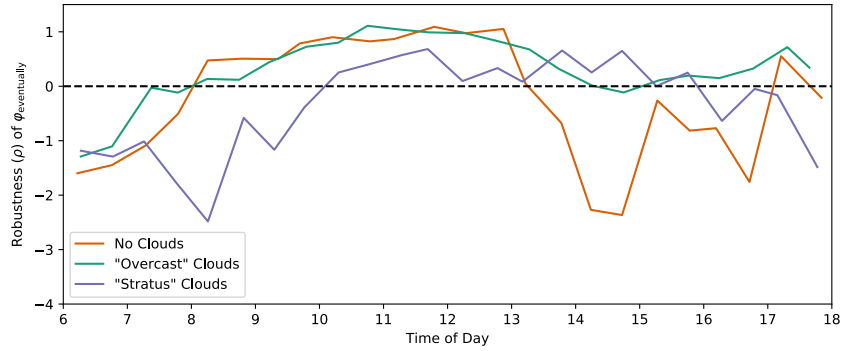


Fig. 5. Median TaxiNet performance by time of day, for different cloud types. (For clarity, individual runs are not shown as dots in this figure.)

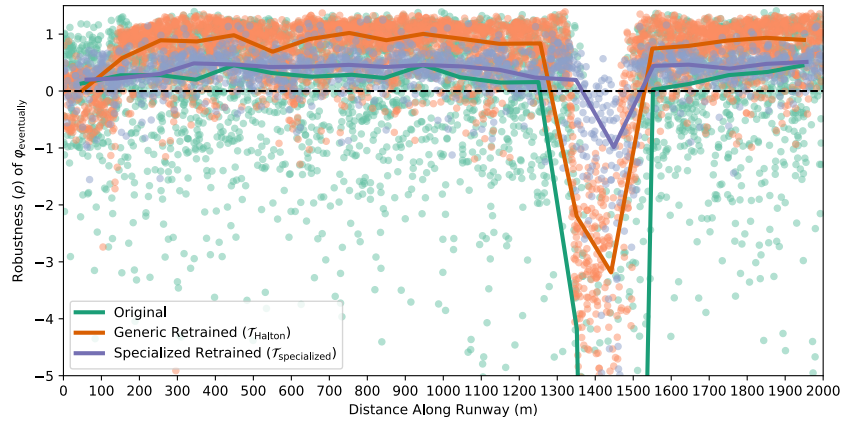


Fig. 6. TaxiNet performance by distance along the runway. Solid lines are medians. The lowest median value for original TaxiNet clipped by the bottom of the chart is -32 .

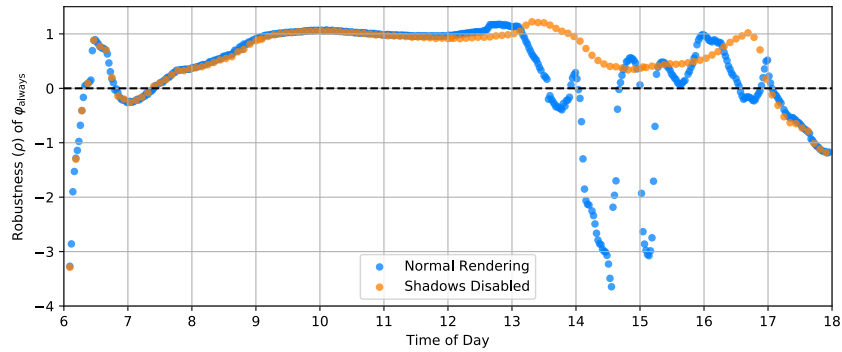


Fig. 7. TaxiNet performance (with fixed plane position) by time of day, with and without shadows.

in performance during the afternoon, as the shadow is moving across the image; with shadows disabled, the fluctuations disappear. This confirms that shadows are a root cause of TaxiNet’s irregular performance in the afternoon.

Figures 4 and 6 show that there are failures even at favorable times and runway positions. We diagnosed several additional factors leading to such cases, such as starting at an extreme angle or further away from the centerline; see the Appendix [5] for details.

Finally, we can use VERIFAI for fault localization, identifying which part of the system is responsible for an undesired behavior. TaxiNet’s main components are the neural network used for perception and the steering controller: we can test which is in error by replacing the network with ground truth CTE and HE values and testing the counterexamples we found above again. Doing this, we found that the system always satisfied $\varphi_{\text{eventually}}$; therefore, all the failure cases were due to mispredictions by the neural network. Next, we use VERIFAI to retrain the network and improve its predictions.

3.4 Retraining

The easiest approach to retraining using VERIFAI is simply to generate a new generic training set using the falsification scenario $\mathcal{S}_{\text{falsif}}$ from Fig. 3, which deliberately includes a wide variety of different positions, lighting conditions, and so forth. We sampled new configurations from the scenario, capturing a single image from each, to form new training and validation sets with the same sizes as for original TaxiNet. We used these to train a new version of TaxiNet, $\mathcal{T}_{\text{generic}}$, and evaluated it as in the previous section, obtaining much better overall performance: out of approximately 4,000 runs, 82% satisfied $\varphi_{\text{eventually}}$, and only 3.9% left the runway (compared to 55% and 9.1% before). A variant of $\mathcal{T}_{\text{generic}}$ using VERIFAI’s Halton sampler, $\mathcal{T}_{\text{Halton}}$, was even more robust, satisfying $\varphi_{\text{eventually}}$ in 83% of runs and leaving the runway in only 0.6% (a $15\times$ improvement over the original model). Furthermore, retraining successfully eliminated the undesired behaviors caused by time-of-day and cloud dependence: the blue data in Fig. 4 shows the retrained model’s performance is consistent across the entire day, and in fact this is the case for each cloud type individually.

However, this naïve retraining did not eliminate all failure cases: the orange data in Fig. 6 shows that $\mathcal{T}_{\text{Halton}}$ still does not handle the runway intersection well. To address this issue, we used a second approach to retraining: over-representing the failure cases of interest in the training set using a specialized SCENIC scenario [6].

We altered $\mathcal{S}_{\text{falsif}}$ as shown in Fig. 8, increasing the probability of the plane starting 1200–1600 m along the runway, a range which brackets the intersection; we also emphasized the range 0–400 m, since Fig. 6 shows the model also has difficulty at the start of the runway. We trained a specialized model $\mathcal{T}_{\text{specialized}}$ using training data from this scenario together with the validation set from $\mathcal{T}_{\text{generic}}$. The new model had even better overall performance than $\mathcal{T}_{\text{Halton}}$, with 86% of runs satisfying $\varphi_{\text{eventually}}$ and 0.5% leaving the runway. This is because performance near the intersection is significantly improved, as shown by the purple data in Fig. 6; however, while the plane rarely leaves the runway completely, it still typically deviates several meters from the centerline. Furthermore, performance is worse than $\mathcal{T}_{\text{generic}}$ and $\mathcal{T}_{\text{Halton}}$ over the rest of the runway, suggesting that larger training sets might be necessary for further performance improvements.

While in this case it was straightforward to write the SCENIC program in Fig. 8 by hand, we can also *learn* such a program automatically: starting from $\mathcal{S}_{\text{falsif}}$ (Fig. 3), we use cross-entropy sampling to move the distribution towards failure cases. Applying this procedure to $\mathcal{T}_{\text{generic}}$ for around 1200 runs, VERIFAI indeed converged to a distribution concentrated on failures. For example, the distribution of distances along the runway gave $\sim 79\%$ probability to the range 1400–1600 m, 16% to 1200–1400 m, and 5% to 0–200, with all other distances getting only $\sim 1\%$ in total. Referring back to Fig. 6, we see that these ranges exactly pick out where $\mathcal{T}_{\text{Halton}}$ (and $\mathcal{T}_{\text{generic}}$) has the worst performance.

Finally, we also experimented with a third approach to retraining, namely augmenting the existing training and validation sets with additional data rather than generating completely new data as we did above. The augmentation data can come from counterexamples from falsification [2], from a handwritten SCENIC scenario, or from a failure scenario learned as we saw above. However, we were not able to achieve better performance using such iterative retraining approaches than simply generating a larger training set from scratch, so we defer discussion of these experiments to the Appendix [5].

4 Conclusion

In this paper, we demonstrated VERIFAI as an integrated toolchain useful throughout the design process for a realistic, industrial autonomous system. We were able to find multiple failure cases, diagnose them, and in some cases fix them through retraining. We interfaced VERIFAI to the X-Plane flight simulator, and extended the SCENIC language with external parameters, allowing the combination of probabilistic programming and active sampling techniques. These extensions are publicly available [1,7].

While we were able to improve TaxiNet’s rate of satisfying its specification from 55% to 86%, a 14% failure rate is clearly not good enough for a safety-critical system (noting of course that TaxiNet is a simple prototype not intended for deployment). In future work, we plan to explore a variety of ways we might further improve performance, including repeating our falsify-debug-retrain loop (which we only showed a single iteration of), increasing the size of the training set, and choosing a more complex neural network architecture. We also plan to further automate error analysis, building on clustering and other techniques (e.g., [10]) available with VERIFAI and SCENIC, and to incorporate white-box reasoning techniques to improve the efficiency of search.

Acknowledgments. The authors are grateful to Forrest Laine and Tyler Staudinger for assistance with the experiments and TaxiNet, to Ankush Desai for suggesting using SCENIC as a prior for cross-entropy sampling, and to the anonymous reviewers.

This work was supported in part by NSF grants 1545126 (VeHICaL), 1646208, 1739816, and 1837132, the DARPA BRASS (FA8750-16-C0043) and Assured Autonomy programs, Toyota under the iCyPhy center, and Berkeley Deep Drive.

```
rd = Options({
  (0, 400): 0.35,      # 0.2
  (400, 1200): 0.1,   # 0.4
  (1200, 1600): 0.5,  # 0.2
  (1600, 2000): 0.05 # 0.2
})
ego = Plane at (-8, 8) @ rd
```

Fig. 8. Position distribution emphasizing the runway beginning and intersection. Probabilities corresponding to the original scenario (Fig. 3) shown in comments.

References

1. Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Ravanbakhsh, H., Vazquez-Chanlatte, M., Seshia, S.A.: VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems. In: 31st International Conference on Computer Aided Verification (CAV). pp. 432–442 (2019), <https://github.com/BerkeleyLearnVerify/VerifAI>
2. Dreossi, T., Ghosh, S., Yue, X., Keutzer, K., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Counterexample-guided data augmentation. In: 27th International Joint Conference on Artificial Intelligence (IJCAI). pp. 2071–2078 (7 2018). <https://doi.org/10.24963/ijcai.2018/286>
3. Dreossi, T., Jha, S., Seshia, S.A.: Semantic adversarial deep learning. In: 30th International Conference on Computer Aided Verification (CAV). pp. 3–26 (7 2018). https://doi.org/10.1007/978-3-319-96145-3_1
4. Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification. pp. 178–192. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
5. Fremont, D.J., Chiu, J., Margineantu, D.D., Osipychev, D., Seshia, S.A.: Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI (2020), <https://arxiv.org/abs/2005.07173>
6. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A language for scenario specification and scene generation. In: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). pp. 63–78 (2019). <https://doi.org/10.1145/3314221.3314633>
7. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A language for scenario specification and scene generation (2019), <https://github.com/BerkeleyLearnVerify/Scenic>
8. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. CoRR **abs/1412.6572** (2014)
9. Halton, J.H.: On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik* **2**(1), 84–90 (1960). <https://doi.org/10.1007/BF01386213>
10. Kim, E., Gopinath, D., Pasareanu, C.S., Seshia, S.A.: A programmatic and semantic approach to explaining and debugging neural network based object detectors. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2020)
11. Koymans, R.: Specifying real-time properties with metric temporal logic. *Real-time systems* **2**(4), 255–299 (1990)
12. Laminar Research: X-Plane 11 (2019), <https://www.x-plane.com/>
13. McKinney, W.: Data structures for statistical computing in python. In: van der Walt, S., Millman, J. (eds.) 9th Python in Science Conference. pp. 51–56 (2010), <https://pandas.pydata.org/>
14. Rubinstein, R.Y., Kroese, D.P.: The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning. Springer, New York, NY (2004). <https://doi.org/10.1007/978-1-4757-4321-0>
15. Russell, S., Dewey, D., Tegmark, M.: Research priorities for robust and beneficial artificial intelligence. *AI Magazine* **36**(4) (2015). <https://doi.org/10.1609/aimag.v36i4.2577>
16. Sankaranarayanan, S., Fainekos, G.E.: Falsification of temporal properties of hybrid systems using the cross-entropy method. In: Hybrid Systems: Computation and Control (part of CPS Week 2012), HSCC’12, Beijing, China, April 17–19, 2012. pp. 125–134 (2012). <https://doi.org/10.1145/2185632.2185653>, <https://doi.org/10.1145/2185632.2185653>
17. Seshia, S.A., Sadigh, D., Sastry, S.S.: Towards Verified Artificial Intelligence. CoRR (2016), <http://arxiv.org/abs/1606.08514>

18. Teubert, C., Watkins, J.: The X-Plane Connect Toolbox (2019), <https://github.com/nasa/XPlaneConnect>
19. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering. p. 303–314. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). <https://doi.org/10.1145/3180155.3180220>, <https://doi.org/10.1145/3180155.3180220>