Formal Analysis of AI-Based Autonomy: From Modeling to Runtime Assurance*

Hazem Torfah[1], Sebastian Junges[1], Daniel J. Fremont[2], and Sanjit A. Seshia[1]

[1] University of California, Berkeley, USA
{torfah,sjunges,sseshia}@berkeley.edu
[2] University of California, Santa Cruz, USA
dfremont@ucsc.edu

Abstract. Autonomous systems are increasingly deployed in safety-critical applications and rely more on high-performance AI/ML-based components. Runtime monitors play an important role in raising the level of assurance in AI/ML-based autonomous systems by ensuring that the autonomous system stays safe within its operating environment. In this tutorial, we present VerifAI, an open-source toolkit for the formal design and analysis of systems that include AI/ML components. VerifAI provides features supporting a variety of use cases including formal modeling of the autonomous system and its environment, automatic falsification of system-level specifications as well as other simulation-based verification and testing methods, automated diagnosis of errors, and automatic specification-driven parameter and component synthesis. In particular, we describe the use of VerifAI for generating runtime monitors that capture the safe operational environment of systems with AI/ML components. We illustrate the advantages and applicability of VerifAI in real-life applications using a case study from the domain of autonomous aviation.

1 Introduction

In recent years, there has been an increase in autonomous and semi-autonomous systems operating in complex environments and relying on artificial intelligence (AI) and machine learning (ML) components to perform challenging tasks in perception, prediction, planning, and control. However, the unpredictability and opacity of AI/ML-based components has hindered the deployment and adoption of autonomous systems in safety-critical applications. To raise the level of assurance in autonomous systems, it is thus important to understand under which environment conditions the behavior of an AI/ML-based component is trusted to keep the system in a safe state. Runtime monitors can help monitor environment conditions to maintain situational awareness, and are also useful for tasks

* This work is partially supported by NSF grants 1545126 (VeHICaL), 1646208 and 1837132, by the DARPA contracts FA8750-18-C-0101 (AA) and FA8750-20-C-0156 (SDCPS), by Berkeley Deep Drive, by the Toyota Research Institute, and by Toyota under the iCyPhy center.

such as hardware failure detection, sensor validation, performance evaluation, and system health management. Thus, runtime monitors are indispensable in the deployment of autonomy in cyber-physical systems (CPS).

Developing techniques to automatically construct runtime monitors that accurately capture the safe operating conditions of AI/ML-based components is a key challenge in the quest for safe and reliable autonomous systems [40]. It is particularly important to capture conditions that ensure that system-level safety specifications are met. Traditionally, monitors are constructed from formal specifications given in some logical formalism, which poses a problem in the setting of autonomous systems: while the requirements on the system are typically well-understood and easily formalized, the conditions on the *environment* under which the system will be correct are not (fully) known. Consider, for example, an ML-based perception module in an autonomous car used for tracking speed signs. The perception module is trained on a data set of labeled images of signs, but correct behavior of the module may depend on other factors such as the velocity of the car or the weather. Analyzing the module in isolation with this data set does not tell us anything about its behavior under such environment factors. One instead needs to construct a model of the system's environment and its other components, and analyze the behavior of the module in that context. There is an urgent need for a systematic approach for understanding at design time how AI/ML-based components behave in complex environments, and from this analysis, constructing reliable runtime monitors that ensure AI/ML-based components are only executed under their safe operation conditions.

In this tutorial, we present such an approach based on VerifAI [11], an open-source toolkit for the formal design and analysis of systems that include AI or ML components. We explore the various features of VerifAI through its capabilities of generating runtime monitors for capturing the safe operation environment of AI/ML-based components. VerifAI provides multiple features including (i) formal modeling of autonomous systems and their environment; (ii) formal specification of AI-based autonomous systems; (iii) falsification, fuzz testing, and other simulation-based runtime verification methods; (iv) computational methods for diagnosing and explaining the success and failure of AI/ML-based autonomous systems; (v) specification-driven synthesis of parameters for AI-based autonomy, and (vi) techniques for runtime monitoring and assurance. We present the complete pipeline of VerifAI, from modeling the environment of a system to generating the corresponding runtime assurance modules.

VerifAI follows a data-driven approach to learning monitors. Data is generated using VerifAI's simulation-based runtime verification techniques. VerifAI allows us to analyze AI/ML-based components using system-level specifications. To scale to complex high-dimensional feature spaces, VerifAI operates on an abstract semantic feature space. This space is typically represented using Scenic, a probabilistic programming language for modeling environments [17]. Using Scenic, we can define scenarios, distributions over spatial and temporal configurations of objects and agents, in which we want to deploy and analyze a system. By simulating the system in the different sampled scenes and applying

the formal analytical backends of VerifAI, we can create the training data sets needed for building the monitors.

Once the training data is created, specialized algorithms for learning different types of monitors can be applied. We discuss the different types of learning algorithms for learning monitors from data. These vary from exact algorithms to statistical approaches such as PAC-based learning algorithms with their passive and active variants. We discuss the advantages and disadvantages of the these algorithms with respect to our setting.

VerifAI has been applied in several real case studies including with industrial partners (e.g., see [19, 16]). We report on one case study from the domain of autonomous aviation, in collaboration with Boeing [16]. We analyzed TaxiNet, an experimental autonomous aircraft taxiing system developed by Boeing for the DARPA Assured Autonomy project. In a previous effort using VerifAI [16], we falsified the system, diagnosed root causes for a variety of identified failure cases, and generated synthetic data to retrain the system, eliminating several of these failures and improving performance overall. In this paper, we build on this case study to describe the automated pipeline needed for the construction of runtime monitors for AI/ML-based systems and illustrate the advantages and applicability of VerifAI in generating such monitors.

Outline. The rest of this paper is organized as follows. We start with a motivating example from the domain of autonomous aviation. In Section 3 we present the general architecture of VerifAI and its ability to model environments using the Scenic language. Section 4 introduces the process of generating data for learning runtime monitors. In Section 5 we discuss and compare the different types of data-driven monitor learning algorithms and discuss some of the monitors we learned with these algorithms. In Section 6 we integrate the monitors into an architecture for runtime assurance. We conclude with a discussion of important desiderata for runtime monitors and directions for future work.

2 Motivating Example: Autonomous Aircraft Taxiing

Consider the scenario of an airplane taxiing along a runway depicted in the images in Figure 1. This scenario is based on a challenge problem provided by Boeing in the DARPA Assured Autonomy program. The plane is equipped with a perception module, a deep neural network called TaxiNet, that based on images captured by a camera mounted on the plane estimates the *cross-track error (CTE)*, i.e., the left-right offset of the plane from the centerline of the runway. The estimated values are forwarded to a controller that adjusts the steering angle of the plane in order to track the centerline.

The deep neural network is a black box, with no further information provided about what images were used to train the network nor any knowledge about potential gaps in the training set and corresponding potential failure cases. Our goal is to construct and equip the system with a monitor that captures the conditions under which the deep neural network is expected to behave correctly

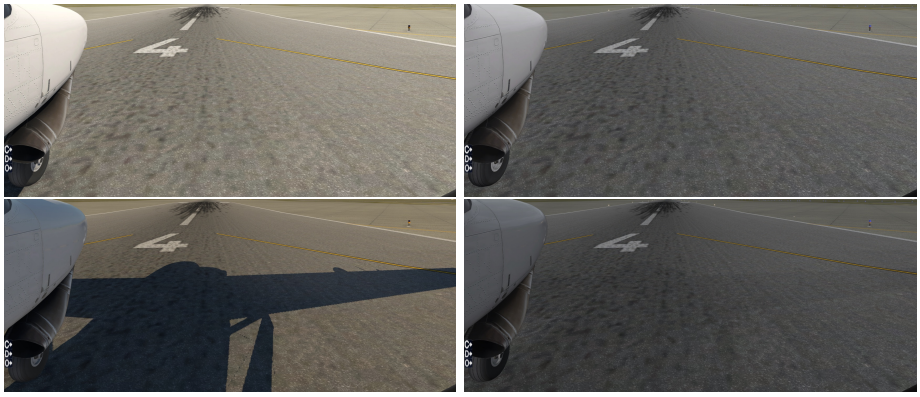

Fig. 1. Example input images to TaxiNet, rendered in X-Plane, showing a variety of lighting and weather conditions [16].

and alert the system about any violation to switch in time to more trustworthy safe components (such as human control).

Many factors can influence the behavior of the deep neural network, and often are not considered in the training process. For example, while TaxiNet was trained on images, its behavior may depend on high-level parameters such as weather conditions (like overcast or rain), time of day, the initial airplane position and heading on the runway, the frequency of skid marks on the ground, or the velocity of the airplane. We refer to these factors as *semantic features* (discussed further in the next section). For our goal, these features must be measurable and monitorable at run time.

Once we fix the semantic features which we intend to use to monitor the neural network, the next step is to establish a connection between values of these semantic features and the value of a system-level specification. For example, the system-level specification could be one requiring that the CTE value should not be larger than 2.5 meters over more than 10 time steps. A monitor for validating the performance of the deep neural network is one that based on the semantic features predicts whether the system-level specification will be violated. For example, under rainy weather conditions, the monitor might predict that the network will consistently yield poor CTE estimates and lead to the plane deviating too far from the centerline.

To establish such a connection, we need a systematic approach that allows us to find the environment conditions under which the aircraft significantly deviates from the centerline when using the deep neural network. We need to explore the diverse set of scenarios possible under different instantiations of the aforementioned semantic features. We need to analyze the executions of the system to identify distinct failure cases and diagnose potential root causes resulting in unsafe CTE values. Lastly, we need to deploy the right learning techniques that based on data generated by the exploration and analysis processes, learn a mon-

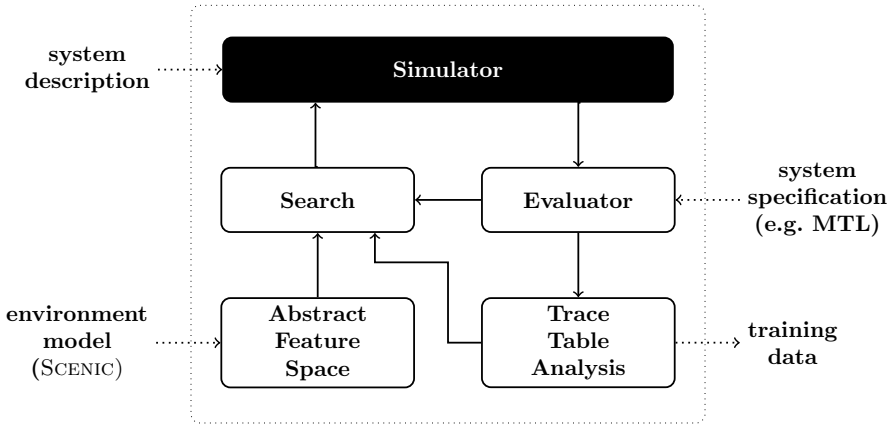

Fig. 2. The architecture of VERIFAI.

itor that predicts a faulty behavior of the system. With VERIFAI we provide a toolkit that includes all the necessary features for learning such a monitor.

3 The VerifAI Framework

VERIFAI follows a paradigm of *formally-driven simulation*, using formal models of a system, its environment, and its requirements to guide the generation of testing and training data [11]. The high-level architecture of VERIFAI is shown in Fig. 2. To use VERIFAI, one first writes an environment model which defines the space of environments that the system should be tested or trained against. Simple models can be specified by manually defining a set of environment parameters and their corresponding ranges; more sophisticated models can be built using the SCENIC probabilistic modeling language, as we will describe below. In either case, the environment model defines an abstract *semantic feature space*, representing environments as vectors in a space of *semantic* features such as object positions and colors. Such a space can have much lower dimension than the "concrete feature space" of inputs to the system (e.g. the space of images for our aircraft taxiing scenario), and it ensures that any counterexamples we find are semantically meaningful, unlike traditional adversarial machine learning [12]. On the other hand, actually testing the system requires turning abstract features into low-level sensor data, for which we depend on a simulator. In order to support a variety of application domains, VERIFAI provides a generic simulator interface allowing it to make use of any simulator which supports the desired domain of systems and environments.

Once the abstract feature space has been defined, VERIFAI can search the space using a variety of algorithms suited to different applications. These include passive samplers which seek to evenly cover the space, such as low-discrepancy (Halton) sampling, as well as active samplers which use the history of past tests

to identify parts of the space more likely to yield counterexamples. Each point sampled from the abstract feature space defines a concrete test case which we can execute in the simulator. During the simulation, VerifAI monitors whether the system has satisfied or violated its specification, which can be provided as a black-box monitor function or in a more structured representation such as a formula of Metric Temporal Logic [27]. VerifAI uses the quantitative semantics of MTL, allowing the search algorithms to distinguish between safe traces which are closer or farther from violating the specification. The results of each test can be used to guide future tests as mentioned above, and are also saved in a table for offline analysis, including monitor generation.

To enable modeling the complex, heterogeneous environments of cyber-physical systems, VerifAI accepts environment models written in the Scenic domain-specific probabilistic programming language [17]. A Scenic program defines a distribution over configurations of physical objects and their behaviors over time. For example, Fig. 3 shows a Scenic program for a runway taxiing scenario used in our previous case study [16]. This program specifies a variety of semantic features including time of day, weather, and the position and orientation of the airplane, giving distributions for all of them (with cloud type and rain percentage being correlated, for example). While this scenario only involves a single object, Scenic's convenient syntax for geometry and support for declarative constraints make it possible to define much more complex scenarios in a concise and readable way (see [17] for examples). Scenic also supports modeling dynamic behaviors of objects, with syntax for specifying temporal relationships between events and composing individual scenarios into more complex ones [18]. Finally, Scenic is also simulator- and application-agnostic, being successfully used in a variety of CPS domains besides aviation including autonomous driving [19], robotics [17], and reinforcement learning agents for simulated sports [3]. In all these applications, the formal semantics of Scenic programs allow them to serve as precise models of a system's environment.

4 Training Data Generation

In this section, we discuss the generation of training (and testing) data for the data-driven generation of monitors. The training data for a monitor M is provided as a table of labeled traces of the form (σ, ℓ), where σ is a sequence of events from the space of inputs over which the target monitor is defined, and ℓ is a truth value indicating whether σ should be accepted or rejected by M. Training data in this form is generated from the execution runs of several simulations through a process consisting of three phases, *mapping*, *segmentation*, and *disambiguation*, as depicted in Figure 4.

4.1 Mapping

The role of the mapper is to establish the connection between the sequence of events collected during a simulation and the inputs to the monitor. In general, the mapper consists of:

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

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

# Plane: up to 8m to left/right of the centerline,
# 2000m down the runway, and 30 degrees to left/right.
ego = Plane at Range(-8, 8) @ Range(0, 2000),
            facing Range(-30, 30) deg
```

Fig. 3. SCENIC runway taxiing scenario (updated slightly from [16]).

– **Projections**: *mapping a sequence of simulation events to a (sub)set of events that can be reliably observed at runtime.* A monitor must be defined over inputs that are observable by the system during runtime. Properties of other entities in the environment may be known during simulation, but not during runtime. Thus, the data collected at runtime must be projected to a stream of observable data. We especially want to project the data onto *reliable* and *trustable* data. Some data may be observable, but should not be used by a monitor because it is not based on reliable hardware or because it typically is unreliable in the particular scenario at hand. For example, a monitor for validating the confidence in the TaxiNet camera-based neural network can be based on the data of the weather condition and the time of day, whereas it might be better to refrain from using the images captured by the camera or the output of the neural network perception module.

– **Filters**: *mapping traces to other traces using transformation functions that may have an internal state (based on the history of events).* Beyond projecting, we may use the data available at runtime to estimate an unobservable system or environment state by means of filtering approaches and then use this system state (or statistics of this state) as an additional observable entity. For example, to validate the conditions for TaxiNet, we may want to use data computed based on an aggregate model that evaluates the change in the heading of the airplane. A filtering approach must (implicitly or explicitly) use a model that connects the observed traces with the notion of a system state based on the dynamics: in our example, the definition of the aggregate. The model does not need to be precise, although the quality of the monitor surely benefits from added precision. Furthermore, uncertainty in

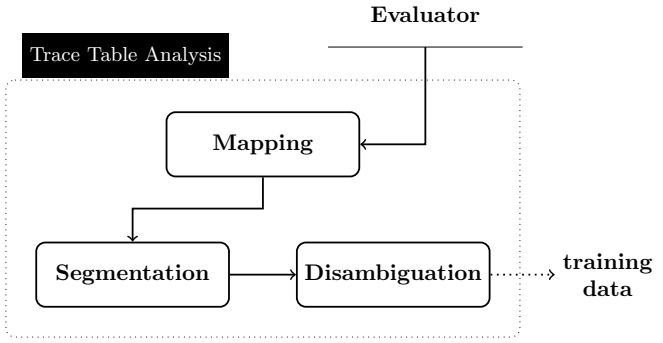

Fig. 4. Training data generation

the model may be made explicit, be it nondeterministic [22], stochastic [44], or both [26]. These filters can also be based on neural networks [7].

At all times, mappers should preserve the order of events as received from the evaluator.

4.2 Segmentation

Once the sequence of events from the simulator have been transformed to traces over adequate input data, they are forwarded to a segmentation process. The result of the segmentation process is a table of pairs (σ, ℓ) where σ is an infix of a trace received from the mapper and ℓ defines the behavior of the ideal monitor M_{true} upon observing a sequence of input data σ.

Extracting Segments (Windowing). Rather than considering traces from the initial (simulation) state, a sliding window approach can be used to generate traces σ of fixed length starting in any state encountered during the simulation. This approach is important to avoid generating monitors that overly depend on the initial situation or monitors that (artificially) depend on outdated events. For example, the behavior of TaxiNet may depend on the size of skid mark patches along the runway. Small patches may not cause major errors in the CTE values or perhaps only for a short recoverable period of time. Larger regions of skid marks may however cause a series of errors that could lead the airplane to leave the runway. A monitor for TaxiNet should check for continuous regions of skid marks over a fixed period of time. Therefore, the monitor does not need the entire history of data, as the plane will recover from small patches, but the monitor should switch from TaxiNet to manual control when the plane drives over a long region of skid marks. In general, the length of segments needs to be tuned based on the application at hand and the frequency in which data is received. We remark that the loss of information due to ignoring events earlier

in the history can be partially alleviated by adding a state estimate to the trace using an appropriate filter in the mapping phase.

Computing Labels. When determining the labels ℓ for each of the extracted segments σ, one needs to take into account the requirement that monitors for validating safe operation conditions must be predictive [9, 8], i.e., after observing σ, the verdict of the monitor is one that predicts whether the safe operation conditions will hold in the future. In our TaxiNet scenario, we want the monitor to alert the system in advance, such that the airplane does not deviate much from the centerline in the (near) future. Segments should thus be associated with the behavior of the plane a few steps into the future. The latter can be characterized with respect to a prediction horizon, usually defined based on the time needed for executing certain contingency plans to ensure the system stays safe. Once a prediction horizon of length n is determined, a segment is labeled with the truth value corresponding to evaluating some property φ up to n steps into the future (along the original simulation run). The property φ is defined over the events of traces received from the evaluator, and may be different from the system-level property used by the evaluator. In particular, it could be an aggregation over the labels computed by the evaluator, e.g., defining a threshold for the allowable number of deviations from the centerline over a period of time.

4.3 Disambiguation

After the table of training data is created by the segmentation process it can be forwarded to any learning algorithm that generates a suitable artifact for the monitor. In the next section, we elaborate on this generation process and discuss what features one might need to consider in choosing the artifact. In general, the data resulting from the mapping and segmentation process may be ambiguous, i.e., the table may include pairs (σ, ℓ) and (σ', ℓ') where $\sigma = \sigma'$ but $\ell \neq \ell'$. In this way, learned monitors can fail to be completely consistent with the data; depending on the learning method used, such inconsistencies can lead to undefined behavior. To further guide the learning process, one may want to apply a preprocessing step, a disambiguation algorithm, that resolves ambiguity in the data set. Such preprocessing can be either conservative and always label a trace as violating a safety property if any pair labels it as violating, or take a more quantitative approach (e.g. if labels are MTL robustness values, averaging all such values for the trace).

5 Monitor Generation

In this section, we discuss different ways to translate the available data from a simulation interface and high-level system description into a monitor M_{syn}. In particular, we aim to construct an efficiently-computable function that for any trace σ yields whether the monitor should issue an alert, i.e., $M_{\text{syn}}(\sigma) \in \mathbb{B}$. We wish to construct this monitor from the training data (and the simulation

interface) outlined in the previous section. Furthermore, the simulation-based environment described above allows us to compare against the ideal monitor M_{true} and define quality metrics in terms of false positives, false negatives, etc. with respect to a sampled set of traces.

5.1 Learning Methods

In general, we can distinguish between *exact* and *approximate* learning methods, on the one hand, and *passive* versus *active* learning on the other.

Exact and approximate learning. We start with a comparison between exact and approximate learning and explain why approximate learning is more suitable for our setting.

Exact learning. In exact learning [23, 48], the idea is to learn an artifact matching the labeled samples, i.e., $M_{\text{syn}} = M_{\text{true}}$. The guarantee that these approaches typically deliver is that they yield the simplest (and according to Ockham's razor, the best) explanation for the training data. If the training data were to be exhaustive, the monitor would be perfect. Using exact learning algorithms in our setting, however, comes with two main challenges. First, the training data is in most cases noisy, whereas many efficient exact learning algorithms require noise-free data. Such an assumption is unrealistic in our setting, even if the system is deterministic, due to quantization and nondeterminism in the simulator. While nondeterminism can be resolved by a disambiguation process (as outlined in the previous section) and using methods such as in [1], this will result in very conservative monitors that are not robust to noisy data. Second, the presence of multiple sensors may yield a large alphabet size, which poses a serious challenge even for state-of-the-art learning algorithms [1, 30]. For example, looking at the training data generated for TaxiNet, even if we further discretize the data by splitting the time of day into 16 different hours (counting night times as one), 6 types of clouds, 4 rain levels, 8 intervals for the initial position of the plane, and a simple binary flag for the skid marks, we already obtain an alphabet size of 6144. Finer discretizations necessary for high-quality monitors would have even larger alphabets.

Approximate learning. To construct monitors that are more robust to noisy data, we may rely on approximate learning methods that learn an optimal monitor for the training data with respect to a quantitative objective (e.g. the misclassification rate). The literature includes a plethora of techniques for learning optimal artifacts, including but not limited to, techniques for learning decision trees [34, 6], decision lists [36], or neural networks [5]. The typical guarantee for a monitor generated using these methods is a statistical one, often in the form of a *probably approximately correct* (PAC) monitor [49]. That is, assuming a sufficient amount of training data from the simulator, the generated monitor will with high probability be (optimally) correct on most of the traces observed at runtime. While

approximately correct monitors do admit false negatives and false positives, by adequately defining the weights of the quantitative objective we may bias the learning process towards false positives. In the TaxiNet example, we used an algorithm for learning decision trees and increased the weights corresponding to the false positives. This allowed for the learning of monitors that triggered more false alarms, but resulted in a fewer number of misclassifications of dangerous situations where the plane left the runway.

Active versus passive learning. Both exact and approximate learning can be either active or passive. In our setting, passive learning is simpler to apply than active learning. We briefly mention some of the challenges.

Passive learning. Passive learning starts from a data set collected *a priori*. This data would typically follow the distribution as specified in the environment model. However, the training set can be primed to include more negative examples (even if they rarely occur). In particular for passive learning, it is interesting to further prepare the data, e.g., by taking a windowed approach. The major downside of exact passive learning is its NP-hardness in the presence of a large alphabet (set of events) [33].

Active learning. In active learning, the learner adaptively runs the simulations. Thus, the data is no longer pre-selected: rather, we give the learner direct access to the simulator. The challenge here is that we then must produce a simulation that after mapping and segmentation yields a particular requested trace. A naive workaround for this problem is to run the simulator until such a trace is found and heavily rely on caching. While generally, such an approach is not feasible, there are cases where the observation trace contains enough information to control the simulation accordingly: e.g., consider a car where the observation is throttle and steering, or a setting where we only vary the initial (static) part of the environment. Furthermore, most learners select traces that accelerate learning (possibly ignoring properties of the trace such as its relative likelihood). This selection scheme can pose a challenge for creating monitors for rare events. Finally, while (approximate) active learners are typically more data-efficient than passive learners, their statistical guarantees are weaker.

5.2 Learning Monitors for TaxiNet

We used VerifAI in experiments implementing the aircraft taxiing example introduced in Section 2. For a given deep neural network implementing the perception module, we used VerifAI to learn a monitor which decides, based on the initial configuration of the airplane, weather conditions, time of day, and skid marks, whether to use an autopilot dependent on the perception module or switch to manual control. In the following, we provide some details on the experimental setup and results.

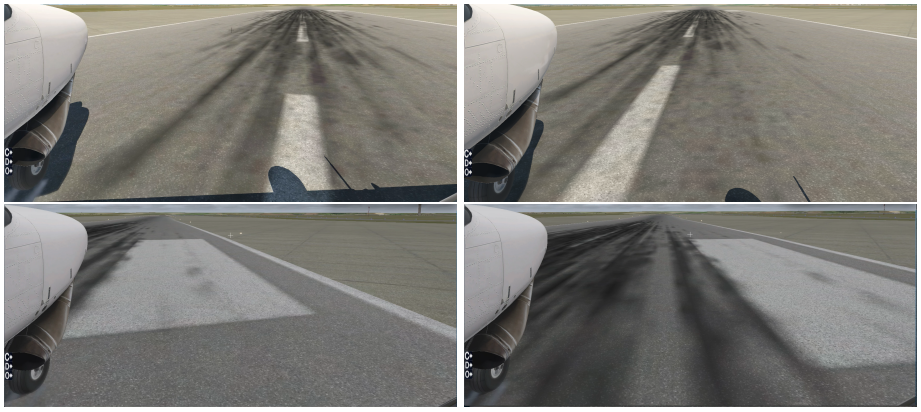

Fig. 5. Example runs using TaxiNet without and with learned monitor. On the left: TaxiNet without learned monitors. On the top right: TaxiNet with monitor learned using mapper 1. On the bottom right: TaxiNet with monitor learned using mapper 2. Note that that the deviation from the centerline is greater on the left than on the right.

Experimental Setup. Our setup uses VerifAI's interface to the X-Plane flight simulator [35]. The perception module was executed as part of a closed-loop system whose computations were sent to a client running inside X-Plane. As a client, we used X-Plane Connect [45], an X-Plane plugin providing access to X-Plane's "datarefs". These are named values which represent simulator state, such as the positions of aircraft and weather conditions, etc.

The deep neural network was trained on images collected from several X-Plane simulations, where each image was labeled with the CTE value observed in that image. The images were taken from a camera mounted on the right wing facing forward, as shown in Figure 1. The environment was modelled by the Scenic program depicted in Figure 3, originally used to falsify and retrain TaxiNet in [16]. The evaluator used the MTL specification $\varphi = \Diamond_{[0,10]} \Box(\text{CTE} < 2.5)$, requiring that the plane get within 2.5 m of the centerline within 10 s and maintain that maximum CTE for the entire simulation. We used the robust semantics for MTL, subsequently mapping positive robustness values (including 0) to true and negative values to false. To label the traces for monitor learning, we used a smoothed version of our specification designed to ignore short-term violations: specifically, we defined the label to be true when the property φ was true at least 8 out of 10 times in the 10 time steps following the prediction horizon of 5 time steps. Traces from 500 simulations were annotated with the truth value of the smoothed specification at each time step. We then applied a procedure for learning optimal decision trees over this training data.

Learned Monitors. We used two mappers for constructing two types of monitors. The first mapper used a segment length of 1 and filtered out all simulation data except for the weather conditions, the time of day, and the initial con-

figuration, thereby covering all static factors. Using this mapper, our approach resulted in a monitor that mainly issued alerts in the afternoon, between 12 pm and 6 pm, in clear weather conditions. This matched our observation (also made in [16]) that the perception module did not behave well under these conditions, leading the airplane to exit the runway in most simulations (see the top row of Figure 5 for an example). During these times we noticed that the shadow of the airplane confused the perception module, which was an indication that the neural network may have not been trained on data during these times and weather conditions. In fact, in very cloudy weather conditions during the afternoon, where no shadow can be observed, the number of alerts by the monitor decreased drastically. This result is in line with the manual analysis performed in [16]. There, the TaxiNet network was retrained on images during these times and weather conditions, which successfully eliminated the negative influence of shadows and yielded more robust performance.

Continuing our experiments, we further noticed that even after integration of the learned monitor, in many simulations the plane deviated from the centerline and left the runaway near markers indicating 1000 feet down the runway (shown in the bottom left image in Figure 5). This showed that the static features filtered by the first mapper were not enough to characterize the safe operating conditions of the perception module. To overcome this problem, we implemented a second mapper that additionally includes information about the skid marks on the runway (using a segment length of 10). Indeed, close to the 1000 foot markers, the runway had a long region of black skid marks that covered the white centerline, resulting in images on which TaxiNet produced wrong values. Using data that included information about the skid marks, our approach was able to learn a monitor that after driving for a period of 10 time steps over a skid mark region, issued an alert forcing a switch to the safe controller (manual control), which maneuvered the airplane back to the center (as shown in the bottom right image in Figure 5). As soon as the plane left this region, the monitor switched back to using TaxiNet.

6 Runtime Assurance

Runtime assurance (enforcement) techniques aim to ensure that a system meets its (safety) specification at runtime [9, 13, 38]. Abstractly speaking, a runtime assurance module modifies the behavior of the system when necessary so that the system remains in a safe state. The key ingredient of runtime assurance components is a monitor (also referred to as decision module, shield or mask) that triggers a modification in the system's outputs. Realizations of runtime assurance vary, and span from suppressing [2] to manipulating the system's executions [42].

In the setting where the goal is to evaluate the performance of a black-box component, as in the case of TaxiNet, runtime assurance is realized by switching to a provably-safe operating mode, i.e., switching to a verified controller that, although it may potentially be less performant than the black-box, is guaranteed to enforce the safety properties of the system. A prominent example of such a

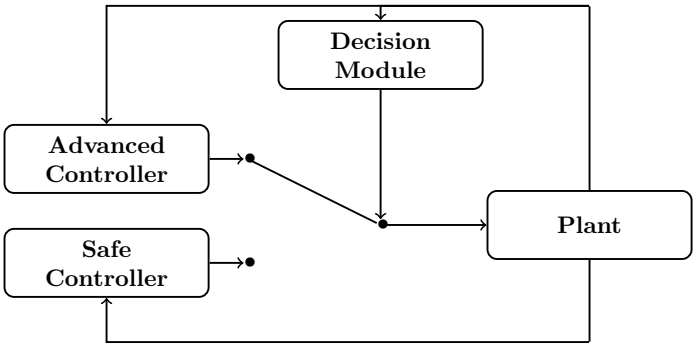

Fig. 6. The Simplex architecture

runtime assurance architecture is the *Simplex architecture* [42], which has been used in many domains, especially avionics [41] and robotics [31]. A runtime assurance module based on Simplex, depicted in Figure 6, typically consists of two controllers, an advanced controller (AC) and a safe controller (SC), and a decision module that implements a switching logic between the AC and SC. The AC is used for operating the system under nominal circumstances. The SC is a certified backup controller that takes over operating the system when anomalies in the behavior of the AC or its safe operating conditions are detected. An SC for TaxiNet could stop the plane and/or ask for human intervention. The decision module decides whether it is necessary to switch from the AC to the SC to keep the system in a safe state and when to switch back to the AC to utilize the high performance of the AC to optimally achieve the objectives of the system. In our TaxiNet example, the decision module is realized by the learned monitor, the AC is a controller that is built on top of the TaxiNet neural network, and the safe controller is a simple mock controller mimicking human control.

When realizing a runtime assurance architecture like Simplex, we should keep the following aspects of the implementation of the decision module in mind:

- the decision module should execute asynchronously, i.e., it should be able to trigger a switch whenever necessary.
- the decision module should aim to use the AC as much as possible without violating safety, i.e., it should switch from SC to AC (AC to SC) as often (little) as possible.
- the decision module needs to be efficient, i.e., it needs to provide, as early as possible, a correct assessment to switch between the AC and SC.

To this end, programming frameworks are needed to implement runtime assurance modules that are guaranteed to satisfy these criteria. An example of such a framework is Soter [9, 43], a runtime assurance framework for building safe distributed mobile robots. A Soter program is a collection of asynchronous processes that interact with each other using a publish-subscribe model of communication. A runtime assurance (RTA) module in Soter consists of a safe

controller, an advanced controller, and a decision module. SOTER allows programmers to construct RTA modules in a modular way with specified timing behavior, combining provably-safe operation with the feature of using AC whenever safe so as to achieve good performance. A key advantage of SOTER is that it also allows for straightforward integration of many monitoring frameworks.

The design and implementation of the decision module is critical to achieve principled switching between the AC and SC that keeps performance penalties to a minimum while retaining strong safety guarantees. Monitors can be defined and implemented using a variety of frameworks [21, 15, 29, 10, 14, 28, 32], ranging from automata and logics to very expressive programming languages with a trade-off between expressivity and efficiency guarantees. The choice of monitor language depends on the requirements of the application and constraints of the implementation platform.

7 Discussion

We have seen how to use VERIFAI and SCENIC to learn runtime monitors for autonomous systems, and how these monitors help to achieve runtime assurance. In this last section, we discuss a more general wish list for such monitors and prospects for additional learning methods to explore.

7.1 Desiderata for Effective Monitors

Runtime monitors for autonomous systems should satisfy several different criteria in order to be useful in practice. While our criteria align partially with those given in [37] for runtime monitors in general, we will see some differences. We also emphasize that our desiderata are not strict requirements: indeed, to the best of our knowledge, no current formalism or method achieves them fully.

Implementability. A monitor should be realizable on the target system and be executable during runtime. This requires that all data the monitor depends on is available and that the monitor can compute the resulting alerts with in a response time sufficient for the system to take corrective action.

Regarding the availability of the inputs, we remark that while we may evaluate monitor performance during simulation by taking into account the ground truth or unobservable data, the real monitor cannot. This concern necessitated the mapping and disambiguation phases in our discussion above.

On the real system, monitor performance is an additional factor. Promptness (i.e., the lag of the monitor) and memory-efficiency (are sufficient resources available) are well-known concerns [14]. In particular, complex numerical or iterative monitor definitions can be problematic, as are huge lookup tables. However, to put these concerns into perspective, we remark that many AI-based controllers are themselves more resource-consuming than traditional embedded controllers. To ensure performance, a first step is limiting the history-dependency through

segmentation. A second step is to limit the size of the artifact, which most learning formalisms support (e.g., limiting the depth of the tree or the size of the network during learning). While we provide monitors in an executable format, compilation onto a real system is currently beyond the scope of our tool.

Quantitative Correctness. Achieving absolute correctness, i.e., ensuring that $M_{\text{syn}}(\sigma) = M_{\text{true}}(\sigma)$ for all possible traces σ, is unrealistic as it would require an exhaustive search through the joint behaviors of the system and its environment. Indeed, AI-based components are most helpful in settings such as perception where correct behavior is difficult to capture in a formal specification, so that there is little hope for fully-correct monitors. A popular approach in verification when full correctness is out of reach is to generate an over-approximate (sound) monitor that only admits a one-sided error. However, such monitors are generally too conservative. Conservative monitors yield many alarms, which typically leads to their outputs being disregarded, defeating their purpose. Rather, we advocate to allow two-sided errors, but to discount false positives over false negatives. Thus, while our monitors do tend to almost always raise an alarm when necessary, they may still err on the conservative side.

To reduce quantitative error, one possibility is to increase the amount of simulation data, thereby covering corner cases with a higher probability. This is not the only option: in order to avoid indistinguishability of various traces, it can also be helpful to increase the variety of data that may be used by the monitor, as we saw in Section 5.2. A third possibility is to increase the expressivity of the monitor formalism: relatively simple models such as decision lists, trees, and finite automata may not be able to adequately capture the underlying dynamics determining system safety in a compact monitor. More elaborate frameworks such as RTLola [14, 4, 46] deliver more flexibility in defining a monitor, while still ensuring a performant implementation as described above.

Trustworthiness. Monitors for autonomous systems which are quantitatively correct in the sense above do not come with the same hard guarantees that many specification-based monitors provide. However, in order to alleviate weaknesses of the system, they must be *more* trustworthy than the system itself. For validation and certification, monitors can be inspected either manually or by verification and other tools. A level of explainability or (machine-)interpretability is thus a central aspect that must be considered when constructing monitors that come with statistical or empirical correctness claims [47]. One approach that VerifAI supports is to simulate the system including its monitor, generating concrete examples of monitor failures using falsification. However, we observe that to some extent, explainability trades off with increasing the quantitative correctness of the tool, as monitors based on a plethora of inputs can be harder to understand and analyze. To better understand the trade-offs, one can use techniques for exploring the Pareto-optimal space of monitors [47].

7.2 Monitor Refinement and Synthesis

Beyond classical learning approaches, an alternative is the use of (oracle-guided) inductive synthesis [24, 25], e.g., counterexample-guided inductive synthesis to learn a monitor by querying an oracle. Such an approach can be used as an extension of active or passive learning algorithms, or alone.

Inductive synthesis is heavily used in the context of programming languages but can also be used for perception modules and control [20]. Rather than learning a program, we learn a monitor. The main idea here is that rather than learning a complete monitor, we have a skeleton of the monitor that may be extracted from domain specific knowledge or learned. In our aviation example, we might search for a monitor that combines the forward speed and estimated CTE and compares this to some threshold. The question to find a monitor then is to find a function of forward speed, CTE and a threshold.

Rather than first collecting the training data offline, an inductive synthesis solver will typically assume some candidate monitor and then use an oracle (here, VerifAI) to evaluate the system with and without the monitor to find false positives and false negatives. These samples are counterexamples that can be used to refine the candidate monitor into a more accurate monitor. We remark that the criteria for the synthesis loop to accept a monitor can either be as in the exact learning case, or PAC-based.

Another direction for monitor synthesis is the paradigm of *introspective environment modeling* (IEM) [40, 39]. In IEM, one considers the situation where the agents and objects in the environment are substantially unknown, and thus the environment variables are not all known. In such cases, we cannot easily define a Scenic program for the environment. The only information one has is that the environment is sensed through a specified sensor interface. One seeks to synthesize an assumption on the environment, monitorable on this interface, under which the desired specification (e.g. safety property) is satisfied. While very preliminary steps on IEM have been taken [39], significant work remains to be done to make this practical, including efficient algorithms for monitor synthesis and the development of realistic sensor models that capture the monitorable interface.

Acknowledgments. The authors are grateful to Johnathan Chiu, Tommaso Dreossi, Shromona Ghosh, Francis Indaheng, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Kesav Viswanadha for their valuable contributions to the VerifAI project. We also thank the team at Boeing helping to define the TaxiNet challenge problem including especially Dragos D. Margineantu and Denis Osipychev.

References

1. Fides Aarts, Bengt Jonsson, Johan Uijen, and Frits W. Vaandrager. Generating models of infinite-state communication protocols using regular inference with abstraction. *Formal Methods Syst. Des.*, 46(1):1–41, 2015.

2. Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On Runtime Enforcement via Suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17, 2018.
3. Abdus Salam Azad, Edward Kim, Qiancheng Wu, Kimin Lee, Ion Stoica, Pieter Abbeel, and Sanjit A. Seshia. Scenic4rl: Programmatic modeling and generation of reinforcement learning environments. *CoRR*, abs/2106.10365, 2021.
4. Jan Baumeister, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. FPGA stream-monitoring of real-time properties. *ACM Trans. Embed. Comput. Syst.*, 18(5s):88:1–88:24, 2019.
5. Luca Bortolussi, Francesca Cairoli, Nicola Paoletti, Scott A. Smolka, and Scott D. Stoller. Neural predictive monitoring. In *RV*, volume 11757 of *LNCS*, pages 129–147. Springer, 2019.
6. Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.
7. Francesca Cairoli, Luca Bortolussi, and Nicola Paoletti. Neural predictive monitoring under partial observability. *CoRR*, abs/2108.07134, 2021.
8. Yi Chou, Hansol Yoon, and Sriram Sankaranarayanan. Predictive runtime monitoring of vehicle models using bayesian estimation and reachability analysis. In *IROS*, pages 2111–2118. IEEE, 2020.
9. Ankush Desai, Shromona Ghosh, Sanjit A. Seshia, Natarajan Shankar, and Ashish Tiwari. SOTER: A runtime assurance framework for programming safe robotics systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.
10. Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods Syst. Des.*, 51(1):5–30, 2017.
11. 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 *CAV*, 2019.
12. Tommaso Dreossi, Somesh Jha, and Sanjit A. Seshia. Semantic adversarial deep learning. In *CAV*, volume 10981 of *LNCS*, pages 3–26. Springer, 2018.
13. Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.*, 38(3):223–262, 2011.
14. Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In *CAV (1)*, volume 11561 of *LNCS*, pages 421–431. Springer, 2019.
15. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, 2004.
16. Daniel J. Fremont, Johnathan Chiu, Dragos D. Margineantu, Denis Osipychev, and Sanjit A. Seshia. Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI. In *CAV*, 2020.
17. 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 *PLDI*, 2019.
18. 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, 2020.

19. 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 *ITSC*, 2020.
20. Shromona Ghosh, Yash Vardhan Pant, Hadi Ravanbakhsh, and Sanjit A. Seshia. Counterexample-guided synthesis of perception models and control. In *American Control Conference (ACC)*, pages 3447–3454. IEEE, 2021.
21. Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *TACAS*, volume 2280 of *LNCS*, pages 342–356. Springer, 2002.
22. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Trans. Autom. Control.*, 43(4):540–554, 1998.
23. Malte Isberner, Bernhard Steffen, and Falk Howar. Learnlib tutorial - an open-source java library for active automata learning. In *RV*, volume 9333 of *LNCS*, pages 358–377. Springer, 2015.
24. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.
25. Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.
26. Sebastian Junges, Hazem Torfah, and Sanjit A. Seshia. Runtime monitors for markov decision processes. In *CAV (2)*, volume 12760 of *LNCS*, pages 553–576. Springer, 2021.
27. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
28. Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: runtime verification of non-synchronized real-time streams. In *SAC*, pages 1925–1933. ACM, 2018.
29. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FTRTFT*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.
30. Irini-Eleftheria Mens and Oded Maler. Learning regular languages over large ordered alphabets. *Log. Methods Comput. Sci.*, 11(3), 2015.
31. Dung Phan, Junxing Yang, Radu Grosu, Scott A. Smolka, and Scott D. Stoller. Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles. *Form. Methods Syst. Des.*, 51(1):62–86, August 2017.
32. Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *RV*, volume 6418 of *LNCS*, pages 345–359. Springer, 2010.
33. Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.
34. J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.
35. Laminar Research. X-Plane 11 (2019). `https://www.x-plane.com/`.
36. Ronald L. Rivest. Learning decision lists. *Mach. Learn.*, 2(3):229–246, 1987.
37. César Sánchez, Gerardo Schneider, Wolfgang Ahrendt, Ezio Bartocci, Domenico Bianculli, Christian Colombo, Yliès Falcone, Adrian Francalanza, Srdan Krstic, João M. Lourenço, Dejan Nickovic, Gordon J. Pace, José Rufino, Julien Signoles, Dmitriy Traytel, and Alexander Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.
38. Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
39. Sanjit A. Seshia. Introspective environment modeling. In *19th International Conference on Runtime Verification (RV)*, pages 15–26, 2019.

40. Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Towards Verified Artificial Intelligence. *ArXiv e-prints*, 2016.
41. Danbing Seto, Enrique Ferriera, and Theodore Marz. Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis). Technical Report CMU/SEI-99-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.
42. Lui Sha. Using simplicity to control complexity. *IEEE Softw.*, 18(4):20–28, 2001.
43. Sumukh Shivakumar, Hazem Torfah, Ankush Desai, and Sanjit A. Seshia. SOTER on ROS: A run-time assurance framework on the robot operating system. In *RV*, 2020.
44. Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *RV*, volume 7186 of *LNCS*, pages 193–207. Springer, 2011.
45. Christopher Teubert and Jason Watkins. The X-Plane Connect Toolbox (2019). `https://github.com/nasa/XPlaneConnect`.
46. Hazem Torfah. Stream-based monitors for real-time properties. In *RV*, volume 11757 of *LNCS*, pages 91–110. Springer, 2019.
47. Hazem Torfah, Shetal Shah, Supratik Chakraborty, S. Akshay, and Sanjit A. Seshia. Synthesizing pareto-optimal interpretations for black-box models. In *FM-CAD*. IEEE, 2021.
48. Frits W. Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, 2017.
49. Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.