Learning Monitorable Operational Design Domains for Assured Autonomy*

Hazem Torfah[1], Carol Xie[1], Sebastian Junges[2], Marcell Vazquez-Chanlatte[1], and Sanjit A. Seshia[1]

[1] University of California at Berkeley, USA
[2] Radboud University, Nijmegen, the Netherlands

Abstract. AI-based autonomous systems are increasingly relying on machine learning (ML) components to perform a variety of complex tasks in perception, prediction, and control. The use of ML components is projected to grow and with it the concern of using these components in systems that operate in safety-critical settings. To guarantee a safe operation of autonomous systems, it is important to run an ML component in its operational design domain (ODD), i.e., the conditions under which using the component does not endanger the safety of the system. Building safe and reliable autonomous systems which may use machine-learning-based components, calls therefore for automated techniques that allow to systematically capture the ODD of systems.

In this paper, we present a framework for learning runtime monitors that capture the ODDs of black-box systems. A runtime monitor of an ODD predicts based on a sequence of monitorable observations whether the system is about to exit the ODD. We particularly investigate the learning of optimal monitors based on counterexample-guided refinement and conformance testing. We evaluate the applicability of our approach on a case study from the domain of autonomous driving.

Keywords: AI-based autonomy, Runtime assurance, Operational design domains, Black-box models

1 Introduction

In recent years, there has been an increase in using autonomous systems in various safety-critical applications such as in transport, medicine, manufacturing, and space. Considering the complexity of the environments these systems are being deployed in, autonomous systems rely on machine learning (ML) techniques to solve complex tasks in perception, prediction and control. The use of

* 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 C3DTI, by the Toyota Research Institute, and by Toyota under the iCyPhy center.

complex, black-box ML components raises concerns regarding the safe operation of ML-based systems [2, 11, 41]. ML models such as deep neural networks are unpredictable; unanticipated changes in the environment may cause a neural network to produce faulty outcomes that could endanger the safety of the system [1, 23, 15]. To raise the level of assurance in autonomous systems, it is therefore crucial to provide designers with the necessary tools that help them understand and further capture the *operational design domain (ODD)* of such systems. In autonomous driving, the SAE J3016 standard for driving automation systems, [37], defines an ODD as the

> *operating conditions under which a given driving automation system or feature thereof is specifically designed to function, including, but not limited to, environmental, geographical, and time-of-day restrictions, and/or the requisite presence or absence of certain traffic or roadway characteristics.*

In general, from a user or authority perspective, the ODD can be seen as the operating environment in which a system should operate safely. To assure the safety of the system, the boundaries defined by an ODD must be monitored during system operation and the system should only operate (autonomously) when these boundaries are met.

However, not every ODD can be monitored at run time. First and foremost, some aspects of the ODD may not be reliably observable or are too expensive to observe. A monitor relying on these aspects is not (efficiently) implementable. Furthermore, we are interested in obtaining monitors that are *predictive* – we optimally want to raise an alarm *before* the system leaves its ODD. Lastly, we emphasize that a powerful ODD needs to be specified over runs of the system, as the history of observable features allows us to approximate hidden system states.

In this paper, we introduce a framework for learning *monitorable operational design domains* of black-box systems, in particular for systems with critical ML components. We define a monitorable ODD to be one that is defined over an observable feature space and that can be implemented as a runtime monitor that predicts whether the system will exit the ODD. This stands in contrast to general definitions of ODDs [37, 27] that do not assume their executability. A monitorable ODD in our framework is learned in terms of a system-level specification that defines a general ODD over a possible non-observable abstract feature space, and a desired class of programs defined over the observable feature space. For a system-level specification, our framework can be used to learn a monitorable ODD from the class of programs that predicts whether the system will violate the specification.

We are particularly interested in learning monitorable ODDs for *temporal system-level* specifications, i.e., where the safe operation conditions bounded by an ODD are defined in terms of timed sequences of observations. Compared to conditions that only rely on non-temporal features such as the type of a road, the state of the system, the weather conditions, or the current traffic situation, the ODDs in our setting incorporate the change of features over time. Consider a perception module in an autonomous vehicle used for lane keeping. While

the perception module accurately computes the distance from the edge of a lane, continuous interruptions of the side markings, for example, due to a line of parking cars or obstacles, may cause the module to produce values triggering a faulty steering behavior by the controller of the vehicle. In general, this should not endanger the vehicle if it happens at a low frequency. An occurrence over a large period of time, may, however, lead to steering the vehicle away from the lane. A runtime monitor learned by our framework may decide to switch to a more safe controller (or manual control) if the latter scenario is detected, and may issue a switch back to using the neural network when the line of obstacles is passed.

Our framework is based on a *quantitative* approach for learning monitors. Since monitors for ODDs are constricted to a specific class of programs and to an observable feature space possibly different from that of the system-level feature space, finding a monitor that exactly captures the ODD of a system may not be feasible: First, the program class may not include enough programs that can cover the entire concept class of functions defined over the observable feature space. Furthermore, a sequence of observations over the observable feature space may correspond to several executions over the system-level feature space. Some of these executions may satisfy the system-level specification and some may violate it. Depending on whether the corresponding sequence of observation is to be classified as part or not included in the ODD will result in a mismatch between the monitorable ODD and the system-level specification. In this case, given a quantitative measure over the system level executions, our approach will learn the optimal monitor over the observable feature space from the class of programs with respect to the given optimality objective. Particularly, our optimality objective is defined in terms of the quantitative measure and the rates of false positives and negatives.

The framework follows a data-driven counterexample-guided refinement approach for learning monitors. Data used for learning are generated via simulation-based runtime verification techniques. Specifically, we use VerifAI, an open-source toolkit for the formal design and analysis of systems that include AI or ML components [14]. VerifAI allows us to analyze 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 [20]. 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. Once the training data has been generated, it is forwarded to an algorithm for learning monitors from the class of interest (e.g., neural networks, decision trees, automata, etc.). The learned monitor is then checked by a conformance tester. The conformance tester relies again on the simulation-based testing techniques provided by VerifAI to check whether a monitor satisfies a given quantitative objective. If this is the case, a monitor is returned. Otherwise, counterexamples found during testing are used in the next learning cycle. We demonstrate the applicability of our framework using a case

study from the domain of autonomous driving. We show that our counterexample-guided approach can be used to learn a monitorable ODD for an image-based neural network used for lane keeping. Our example is inspired by several real case studies VerifAI has been applied in including with industrial partners (e.g., see [22, 19, 47]).

We summarize our contributions as follows:

- We formalize the notion of operational design domains by introducing the problem of finding monitorable operational design domains of systems with respect to system-level specifications.
- We present a framework for learning monitorable operational designs domains based on a quantitative counterexample-guided refinement approach.
- We present a case study that demonstrates the applicability of our framework and that points out the challenges in learning monitorable ODDs for black-box (ML) models.

2 Motivating Example: Autonomous Lane Keeping

Fig. 1. Example input images to the neural network, rendered in CARLA, showing a variety of orientation, weather, and road conditions.

Consider a scenario of an autonomous vehicle driving through a city. The car is equipped with a camera-based perception module using a convolutional neural network that based on images captured by a camera (cf. Figure 1) estimates the *cross-track error (CTE)*, i.e., the lateral offset of the car from the centerline of the road. The estimated values are forwarded to a controller that adjusts the steering angle of the car. The perception module is a black box. In particular, we do not have access to (any statistics of) the images used to train the network nor any knowledge about potential gaps in the training set.

Our goal is to learn a monitor that captures the conditions under which using the neural network does not result in large CTE values that endanger the safety of the car. The monitor should alert in time, to refrain from using the network and maybe switch to a more trustworthy safe controller, e.g. to human control or one that is less optimal but uses more trustworthy sensors.

The behavior of the neural network may be influenced by many factors, some of that may not have been sufficiently covered during training. For example, while the network was trained on images, its behavior may depend on other parameters not accounted for in the input to the network such as weather conditions (like precipitation or cloudiness), the sun angle, thus, determining the time of day and shadowing effects, the position and heading on the road, its velocity, and other objects on the road. We refer to these factors as *semantic features*. For our goal, these features must be observable and monitorable at run time.

Once we fix the observable semantic features, which we intend to use to monitor the system, the next step is to establish a connection between the values of these semantic features and a general system-level safety specification (e.g., leaving the lane). A monitor that implements this connection is one that captures the ODD of the neural network with respect to the above-mentioned semantic features and predicts whether the system will leave the ODD. For example, under rainy weather conditions, at certain turns, or after observing certain landmarks on the road, the monitor might predict that the system will likely deviate too far from the centerline or even exits its lane.

We present a systematic approach to capturing the connection between the system-level specification and the sequences of values of observable features. Our approach is based on exploring the diverse set of scenarios possible under different instantiations of the aforementioned semantic features and analyzing the executions of the system with respect to a system-level specification. Based on data generated by the exploration and analysis processes, our approach learns a monitor that predicts a faulty behavior of the system, or in other words, leaving the ODD of the neural network.

3 Optimal Monitors for Operational Design Domains

In this section, we introduce the problem of learning a monitorable operational design domain of a system. We first establish some key definitions. We then define the learning problem, and finally state some of the challenges in constructing matching monitors for operational design domains.

3.1 Learning monitors for ODDs

Notation. For an (possibly infinite) alphabet Σ, we define the set of traces over Σ by the set of finite words Σ^*. We define the set of traces of a fixed-length $d \in \mathbb{N}$ over Σ by the set Σ^d. A language over Σ is any set $L \subseteq \Sigma^*$. A language of d-length traces is any set $L \subseteq \Sigma^d$.

For a (discrete-time) black-box system with inputs $\mathcal{I}$ and outputs $\mathcal{O}$, we capture its behavior as a discrete sequence of input-output pairs. Formally, we use $\Sigma_{sys} = (\mathcal{I} \times \mathcal{O})$. The system behavior is then a language $C \subseteq \Sigma_{sys}^*$. We make no further assumptions over the system, in particular, we allow for the system to be nondeterministic, i.e., the system may provide different outputs for the same sequence of inputs. The system-level specification, encoding a correct system behavior, can be captured as set of traces over input-output pairs that the system's behavior should not deviate from. Formally, a system-level specification is a language $\varphi \subseteq \Sigma_{sys}^*$. A system $C \subseteq \Sigma_{sys}^*$ satisfies a specification $\varphi \subseteq \Sigma_{sys}^*$ if $C \subseteq \varphi$. We denote the satisfaction relation of systems and specifications by $C \models \varphi$.

For a specification φ and a system C, the *operational design domain* of C with respect to φ, captures the set of "behavioral conditions" where the system C is guaranteed to satisfy the specification φ. In a discrete-time model, we define a behavioral condition as a sequence of observations that can be observed off the system. Formally, we define the operational design domain D of C and φ as the tuple $D_{C,\varphi} = (\Sigma_{obs}, obs, d)$, where Σ_{obs} defines a set of observable inputs and actions, $obs \colon \Sigma_{sys}^* \to \Sigma_{obs}^*$ defines the relation between the system-level inputs and actions and the observations of interest, and $d \in \mathbb{R}^+$ is the prediction horizon. An operational design domain D defines a set $[\![D]\!] = \{\sigma \mid \forall \tau \in \Sigma_{sys}^*.\ obs(\tau) = \sigma \rightarrow \forall \tau' \in \Sigma_{sys}^d.\ \tau \cdot \tau' \notin \overline{\varphi}\}$. Intuitively, $[\![D]\!]$ defines the set of sequences of observations that cannot be mapped to a trace of the system C that violates the specification φ in d steps. We highlight that our definition of ODDs allows us to distinguish temporal interactions of the system with its environment, e.g. that driving over road marks for a short time is not problematic, but that driving over such an area for a prolonged time is problematic.

Our goal is synthesize a runtime monitor that captures the ODD of a system and a specification with respect to a set of observations. A runtime monitor M for an ODD D over observations Σ_{obs} is a program that implements a function $f_M \colon \Sigma_{obs}^* \to \mathbb{B}$, such that, for every trace $\tau \in \Sigma_{obs}^*$, $f_M(\tau)$ if and only if $\tau \in [\![D]\!]$[3]. In the rest of the paper will use f_M to also denote the set of traces τ for which $f_M(\tau) = \mathit{true}$. We formalize the monitor synthesis problem for ODD as follows.

Problem 1 (Synthesizing Monitors for ODDs) *For an operational design domain $D_{C,\varphi} = (\Sigma_{obs}, obs, d)$ and a class of monitors $\mathcal{M}$ over Σ_{obs}, find a monitor $M \in \mathcal{M}$, such that $f_M = [\![D]\!]$, or report that there does not exist such monitor.*

The ODD definition and the monitor extraction problem described above is idealized. In the following, we give some details on why this idealized problem statement is not well suited in practice and present a quantitative more practical version of the problem.

[3] We choose a Boolean codomain for monitors for simplicity reasons. Our approach can be extended easily to quantitative domains, i.e., monitors with a robustness semantics [9, 12].

3.2 Challenges in learning monitorable ODDs

The problem statement above yields monitors that are too conservative. In particular, it assumes the possibility of absolute safety: An observation trace is excluded from the ODD if any system-level traces that violates the specification may yield this observation trace. In line with safety standards, a practical formulation of the problem relaxes the safety requirements to a more quantitative setting. We observe that the occurrence of system-level traces which match a particular observation trace may be rare. In this case, including their corresponding observations in the operational design domain may be admissible, even in safety-critical domains. Furthermore, the class of monitors may not always include a monitor for the exact ODD. Semantically speaking, the monitors within a class typically cover only a subset of monitors over Σ_{obs}. In this case, our goal would be to search for an optimal monitor, e.g., one with the lowest misclassification rate. The optimality of a monitor can be defined in terms of a measure $\nu \colon \mathcal{P}(\Sigma_{obs}^*) \to \mathbb{R}^+$ over sets of observation traces. In this case, the monitor learning problem is converted to the following optimization problem. For a system C, a specification φ, and a class of monitors $\mathcal{M}$, find a monitor $M \in \mathcal{M}$, such that,

$$M \in \operatorname*{arg\,min}_{M' \in \mathcal{M}} \ \nu(f_{M'} \triangle [\![D]\!]),$$

where $\triangle$ denotes the symmetric difference.

While the latter formulation overcomes the mismatch in Problem 1, by searching for the optimal monitor, practically solving the problem is still faced with some issues. First, the usage of a *symmetric* difference treats *false positives* and *false negatives* equivalently. False positives are given as the set of traces of C that satisfy φ, but that are mistakenly identified by the monitor to be executions that lead to a violation of the ODD. False negatives are traces of C that violate φ but are not captured by the monitor as erroneous. In safety-critical settings, this is inadequate, and in general, we want the ability to find monitors that favor false positives over false negatives whenever possible. Another shortcoming of the formulation above is that it requires defining correctness/optimality on sets of traces over Σ_{obs}, which is often troublesome, as correctness/optimality is defined in our setting as a system-level specification, i.e., on traces over Σ_{sys}.

To address these challenges, in the next section, we present a quantitative variant of the monitor learning problem for ODDs. It is based on a correctness definition with respect to the traces over Σ_{sys}, and thus transforms the problem to minimizing a measure μ on languages over Σ_{sys}. This variant allows us to search for optimal monitors within a given class of monitors and with respect to given quantitative measure on the sets of system-level traces.

3.3 Quantitative monitor learning

Considering the challenges discussed above, a practical definition of the problem of learning optimal monitorable ODDs needs to define optimality with respect to

1. system-level traces, i.e., traces over Σ_{sys}

2. the rates of and biases towards false positives and negatives

One consequence of transforming the definition to measures over system-level traces is the matter of predictiveness. To remind the reader, an ODD is defined in terms of a prediction horizon d. The value of a monitor f_M for a trace τ_{obs}, depends the value of φ on system-level trace τ_{sys} of length $|\tau_{obs}|+d$. The problem definition should take this prediction horizon into account when defining the measure over system-level traces. To accommodate for this difference in length, we cut off all suffixes of length d of all traces in $(C \cap \varphi)$ and $(C \cap \overline{\varphi})$. In the problem definition, we will make use of the following notation: for a language L, we let L^{-d}, for $d \in \mathbb{N}$, denote $L^{-d} = \{\alpha_0\alpha_1 \dots \alpha_{k-d} \mid \alpha_0\alpha_1 \dots \alpha_k \in L, k \in \mathbb{N} \text{ s.t. } k - d \geq 0\}$.

Problem 2 (Optimal Monitor Synthesis for ODDs) *Given an operational design domain $D = (\Sigma_{obs}, obs, d)_{C,\varphi}$ of a system C and a specification φ over Σ^*_{sys}, a class of monitors $\mathcal{M}$ over Σ_{obs}, and a measure $\mu\colon \mathcal{P}(\Sigma^*_{sys}) \to \mathbb{R}^+$, find a monitor $M \in \mathcal{M}$, such that,*

$$M \in \operatorname*{arg\,min}_{M' \in \mathcal{M}} \ \mu(T_p \setminus obs^{-1}(f_{M'})) + w_{fn} \cdot \mu(T_n \cap obs^{-1}(f_{M'}))$$

for fixed values $w_{fn} \in \mathbb{R}^+$ and where $T_p = (C \cap \varphi)^{-d}$ and $T_n = (C \cap \overline{\varphi})^{-d}$.

The problem statement above defines a monitor as optimizing a kind of loss function with respect to system-level traces. The left side of the sum in the objective function defines a measure over the false positives. The false-negatives side of the objective function is weighted by w_{fn}, that allows us to bias the search towards false positives or false negatives.

Example 1. A system C we are interested in capturing its ODD, could be the image-based neural network from our motivating example. A monitor for the ODD in this case can be defined over values of the weather, time of the day, location, and road properties, representing a projection of general system-level values, such as the state of the car, or the images received by the neural network as well as its output. On top of system-level traces we define a measure μ that for a set Γ returns the ratio of Γ to the entire set of system-level traces.

Remark 1 (Relation between Problem 1 and Problem 2). In cases where the ODD can be captured by a monitor in a given class and where absolute safety is realizable, a solution to Problem 2 will indeed solve Problem 1.

3.4 Black-box vs. white-box settings

Problem 2 defines the optimality with respect to a system C, i.e., a set of traces. In a white-box setting, one can assume access to a model defining the entire set of traces and thus extract models for the sets T_p and T_n by evaluating φ over C. In a black-box setting, this is in general infeasible. Obtaining an exhaustive set of samples from a black-box model is not practical, considering the large

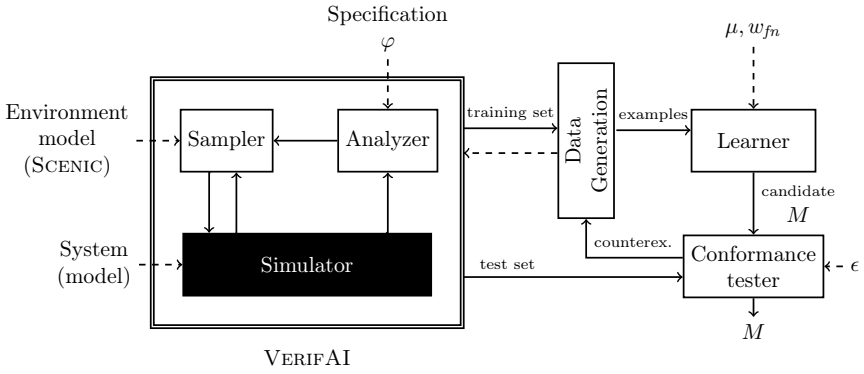

Fig. 2. Extension of VerifAI with the monitor learning framework

(potentially infinite) inputs domains autonomous systems are defined over. The question that we need to raise at this point is how to sample from the black box and how large this sample set must be to obtain monitors that do not overfit the set of samples. Depending on the class of monitors at hand, we can rely on theories from the field of *probably approximate correct learning (PAC)* [43] to construct monitors that are closest to optimal with high confidence. In practice this requires a large number of samples, considering that the class of monitors needed to obtain good monitors is usually very large. In this paper, we suggest a different approach based on conformance testing Here we rely on learning monitor from a small set of samples performing a conformance test to check the quality of the monitor (A testing PAC guarantee using theories such as the Hoeffding's inequality [25]). Relating this to Problem 2, the sets T_p and T_n are then defined with respect to the sample set and the monitors learned are optimal with respect to these sets. Conformance testing is done with respect to the measure μ and the sample sets are extended based on counterexamples obtained during testing. A framework implementing this workflow is given next.

4 Framework

We present of counterexample-guided learning framework. We sketch the overall architecture given in Figure 2 and give details on the individual components in separate sections.

4.1 Main workflow

We integrate three major components into a joint framework: *simulation-based analysis*, *data generation and learning*, and *conformance testing*. Given an executable model of the system with the black-box (ML) component, a model of the environment in which the system is to be executed, we use VerifAI [14] to run simulations and evaluate them according to a provided system-level

specification, cf. Section 4.2. The evaluated simulations are then forwarded to another component for data generation. The data generation component performs several operations on top of the simulation traces, applying certain filters, transformations, and slicing, cf. Section 4.3. Once the data has been prepared for learning, a learner of our choice runs on top of the data. The outcome is an (optimal) monitor implementing the ODD of the black-box component. Finally, a conformance tester checks the quality of the monitor, cf. Section 4.4. Here, the conformance tester may use further simulation runs, using VerifAI, to search for any counterexamples. If conformance testing succeeds, the framework terminates and returns the so-far learned monitor. Otherwise, counterexamples found during testing are passed to the data generating process to compute a new set of data over which a new monitor is learned.

4.2 Simulation-based analysis using VerifAI and Scenic

VerifAI is an open-source toolkit for the formal design and analysis of systems that include AI or ML components [14]. 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. The high-level architecture of VerifAI is shown in Fig. 2. To use VerifAI, one first provides an environment model which defines the space of environments that the system should be tested or trained against. Environment models can be specified using the Scenic probabilistic modeling language [20]. 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 the lane keeping scenario used in our case study. This program specifies a variety of semantic features including time of day, weather, and the position and orientation of the car, giving distributions for all of them. Scenic also supports modeling dynamic behaviors of objects, with syntax for specifying temporal relationships between events and composing individual scenarios into more complex ones [21]. Finally, Scenic is also simulator- and application-agnostic, being successfully used in a variety of CPS domains including autonomous driving [22], aviation [19], robotics [20], 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. For more examples we refer the reader to [20].

Once the abstract feature space has been defined, VerifAI can search the space using a variety of sampling algorithms suited to different applications (e.g., 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 [32, 35]. VerifAI uses the quantitative semantics of MTL [9, 12], allowing the search algorithms

```
param weather = Uniform('ClearNoon', 'CloudyNoon',
                        'WetNoon', 'MidRainyNoon'
                        'ClearSunSet')

lane = Uniform(*network.lanes)
start = OrientedPoint on lane.centerline

ego = Car at start,
    with visibleDistance 60,
    with behavior EgoBehavior(10)
```

Fig. 3. A SCENIC program specifying the environment for the lane keeping scenario

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.

4.3 Data generation

In this section, we discuss the training data generation process. Training data is generated from the execution runs of several simulations through a process consisting of two phases, *mapping* and *segmentation*.

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 a *projection* and a *filtering* phase.

Projection involves *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 simulation 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. For example, a monitor for validating the confidence in using the camera-based neural network, can be based on the data of the weather condition and the time of day, radar values, whereas it might be better to refrain from using the images captured by the camera.

Filtering involve *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 our neural network, we may want to use data

computed based on an aggregate model that evaluates the change in the heading of the car.

At all times, mappers should preserve the order of events as received from the evaluator and maintain the valuations of the system-level specification on the original system-level trace.

Segmentation 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 the car in our lane keeping example, may depend on the frequency of obstacles along the side of the road. Short-period occurrences may not cause major errors in the CTE values or perhaps only for a short recoverable period of time. Frequent occurrences may however cause a series of errors that could lead the car to exit the lane. Therefore, the monitor does not need the entire history of data, as the car will recover from small patches, but the monitor should switch from using the neural network-based controller to manual control when the a long series of obstacles on the side of the road is observed. 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.

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. We feed the traces that we obtain in a trace *warehouse*. From that warehouse, we select traces to feed into the learner.

4.4 Conformance testing

The goal of conformance testing is to test the quality of our learned monitors. This is done by testing the monitor on new independent simulation runs using VerifAI and checking whether a hypothesis with respect to the optimality objective is met. If we pass the hypothesis, we have found a monitor. If not, we augment our warehouse with the counterexamples found during testing. We particularly look for cases, where the monitor failed to issue an alert, and the specification was violated d steps later, where d is the prediction horizon, i.e, false negatives. We also look for cases, where the monitor issued false alerts, triggering unnecessary switches to manual control, i.e., false positives.

The result of the conformance tester is given relative to the set of sampled traces in VerifAI. For high confidence in the result of the conformance tester, we need to make sure to test the monitor on a sufficient number of simulations. For example, assuming that we sampled the simulation traces i.i.d. from the actual distribution, and assuming that we are using a quantitative measure over a σ-algebra over traces, then using Hoeffding's inequality, we can determine the number of samples for given error and confidence measures. For more details on this we refer the reader to [25].

5 Experiments

We used our framework in an experiment for learning a monitor for the ODD of the system with the image-based perception module used for lane keeping as described in Section 2. The perception module is a convolutional neural network (CNN) that for a given snapshot taken by a camera mounted at the front of the car returns the estimated cross-track error to the centerline of the road. We are interested in learning a monitor that based on features such as precipitation, cloudiness, the sun angle, and location determines whether the system will be safe in the presence of these conditions. In our experiment, we evaluate the latter based on whether the car exits its lane. In the following, we provide some details on the experimental setup and results.

5.1 Experimental Setup

Our setup uses VerifAI's interface to the CARLA simulator [13]. The perception module was executed as part of a closed-loop system whose computations were sent to a client running inside CARLA. These are named values that represent the simulator state, such as the position of the car, its velocity, heading, weather conditions, other objects on the road, etc.

The environment is modelled by the Scenic program depicted in Figure 3. The sampler was able to choose simulations in different weather conditions, different roads and initial positions on the road, and different sun angle, thus sampling different times of the day and their shadowing effects. The behavior of the ego car was implemented as a call to an external function `OncCarAction`, which depending on the setting either used the perception-based controller for steering or switched between perception-based control and a safe controller (mimicking manual control) if we were testing a learned monitor.

To evaluate simulation runs we used a built-in CARLA specification for detecting lane invasions. Initially, we started with 100 simulations. In each conformance testing round, we used ca. 160 i.i.d sampled scenes from Scenic. The number of samples were computed using Hoeffding's inequality [25] for confidence value $\alpha = 0.05$ and error-margin $\epsilon = 0.07$. Lastly, we fixed the class of decision trees as the class of our monitors and used a decision-tree learning procedure provided by the sci-kit learning library[4].

5.2 Results

We perform two experiments. The first is solely on static features, such as weather and time of the day (using the sun angle attribute of CARLA), The second additionally considers dynamic features such as the location and road information. The results show the importance of dynamic features in capturing adequate and monitorable ODDs.

[4] https://scikit-learn.org/

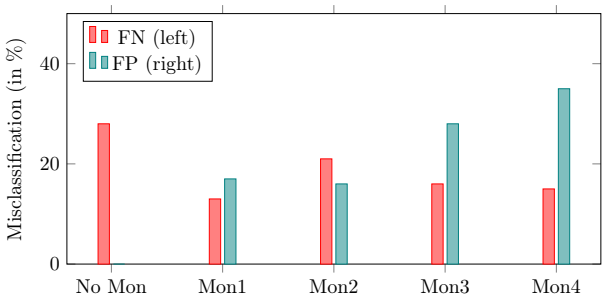

Fig. 4. Results using only static features

We executed our framework for several iterations. In the initial iteration, referred to by *No Mon* in Figure 4 and Figure 5, referring to one where we did not use a monitor (or using a monitor that does not issue alerts), we evaluated the performance of the network by calculating the rate of lane invasions over the number of steps performed in 163 simulation runs. Each run included 250 simulation steps. We use the value as a reference for later iterations to determine how the learned monitors in each iteration increase the safety of the system. The initial lane invasion rate was 21%.

In each iteration that follows, a new monitor is learned (indicated by Mon 1 to Mon 4). For each monitor, we calculated the false negatives rate and the false positives rate. The false negatives rate determines the lane invasion rate in the presence of the monitor. We compare this value to the initial reference rate to determine the increase in safety after using the monitor. To determine the quality of the monitor we also looked at the false positives cases where the monitor issued an unnecessary switch to the safe controller.

Results for static features In this first experiment, we only use values of the static features of precipitation, cloudiness, and sun angle to train the monitor.

In the first iteration, while the monitor can reduce to rate of lane invasions by 8%, the false positives rate of that monitor is very high. We apply another round of learning, this time amending the warehouse with counterexamples, both false positives, and negatives examples, collected during conformance testing. In the second iteration, the process managed to learn a monitor with a lower false positives rate, at the cost of increasing the false negatives rate. With further iterations, the misclassification rate increased, due to an increase in the false positives rate. From then on, the rates kept fluctuating aggressively. In Figure 4 we present the first five iterations.

The aggressive fluctuation is an indication that we have exhausted the role of the given semantic features in learning an optimal monitor with respect to these features. This in turn means that we need to extend the set of features with ones that allow us to construct monitors that can distinguish more cases than with the smaller set of features. For example, we noticed in some cases, that while the monitors constructed in the above experiments captured well the weather

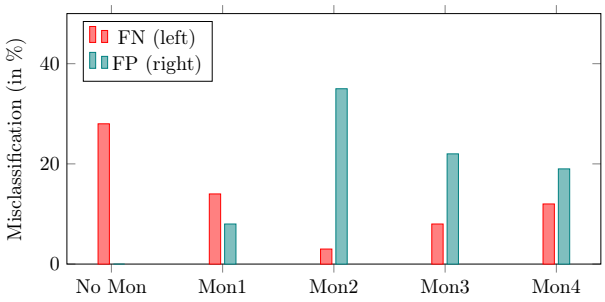

Fig. 5. Results using static and dynamic features

conditions where the CNN will mostly keep the system safe, in some corner cases such as entering a junction or a sharp turn, a lane invasion was occurring even when adequate weather conditions were present. In the next experiment, we show that we can improve on this, by adding location information, which will allow our framework to distinguish these cases from the general weather cases and return better quality monitors.

Results for dynamic features In this experiment, in addition to the features of precipitation, cloudiness, and sun angle, we used features defining the road id and the location of the car on this road. The latter two indirectly capture dynamic features such as being at the end of the road or passing by certain landmarks. Using the new additional features we were able to learn monitors with lower false negatives and false positives rates than monitors solely based on static features. While the rates fluctuate at the beginning, they start to stabilize after the third iteration.

By looking at simulations using some of the monitors above, we did indeed encounter situations where the monitor triggered an alert shortly before arriving at a junction or sharp turn. These scenarios would not have been able to be detected using the monitors from the previous experiment. Scenarios that could still be not handled by the new monitors, were cases where driveways had a similar texture and curvature as the roads. This emphasizes the importance of feature engineering in the learning process of monitors. In the future, we plan to build on our findings to further investigate this problem.

6 Related Work

Operational design domains A key aspect in assuring the safety of AI-based autonomous systems is to clearly understand their capabilities and limitations. It is therefore important to establish the operational design domains of the system and its components and to communicate this information to the different stakeholders [5, 30, 31]. Several works have been dedicated to investigating ways of describing ODDs. Some of them are textual and follow a structured natural

language format for describing ODDs [37, 48]. Others include a tabular description defining a checklist of rules and functional requirements that need to be checked to guarantee a safe operation of the system [45]. A generic taxonomy of the different ODD representation formats is presented in BSI PAS 1883 standard [26].

While the approaches above concentrate on the design of languages for describing ODDs, many works have concluded that there also is a necessity for ODDs to be executable, e.g., to enable the construction of monitors that can be used at runtime [7]. To this end, there has been a focus on developing machine-readable domain-specific languages for implementing ODD, to enable specification, verification, and validation of the ODD, both at design and runtime [27]. In contrast to previous work, we go one step further and present a framework for the automated construction of ODDs, i.e., for a given system component we learn the ODD which is initially unknown. We especially introduce a formal definition of monitorable ODD. Based on this definition we present a new quantitative formalization of the problem learning optimal ODDs for black-box models and solve the problem using a counterexample-guided learning approach.

Runtime verification Runtime verification and assurance techniques aim to ensure that a system meets its (safety) specification at runtime [8, 16, 39]. A large body of work in the runtime verification community has been dedicated to the development of specification languages for monitoring and investigating efficient monitoring algorithms for these languages. Most of the work on formal runtime monitoring is based on temporal logics [18, 38, 33]. The approaches vary between inline methods that realize a formal specification as assertions added to the code to be monitored [38], and outline approaches that separate the implementation of the monitor from the system under investigation [18]. Based on these approaches and with the rise of real-time temporal logics such as MTL [32] and STL [35], a series of works introduced new algorithms and tools for the monitoring of real-time properties [4, 10, 46, 17]. Neural networks themselves may be used as monitors [6]. All these monitoring can be adopted in our framework and can be used as monitoring tools for evaluating runtime properties during simulation. Runtime monitoring techniques can also be applied to investigate whether the input to a known neural network is within its support [34].

Furthermore, the literature also includes a list of frameworks for designing systems with integrated runtime assurance modules that are guaranteed to satisfy these criteria. An example of such a framework is Soter [8, 44], 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 module in Soter is based on the famous Simplex architecture [42] and consists of a safe controller, an advanced controller, and a decision module. A key advantage of Soter is that it also allows for straightforward integration of many monitoring frameworks. Another approach based on Simplex is the ModelPlex framework [36]. ModelPlex combines design-time verification of CPS models with runtime validation of system executions for compliance with the model to build correct by construction runtime monitors which validate at runtime any assumption on the

model collected at design time, i.e., whether or not the behavior of the system complies with the verified model and its assumptions. In case an error is detected, a fail-safe fallback procedure is initiated.

Counterexample-guided synthesis Our learning and conformance testing loop is a quantitative extension of the general line of work of inductive synthesis [28, 29]. We particularly use a quantitative extension of counterexample-guided synthesis to learn a monitorable ODD by querying an oracle, in our case the conformance tester. Inductive synthesis is heavily used in the context of programming languages but can also be used for perception modules and control [24]. 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.

Another direction for monitor synthesis is the paradigm of introspective environment modeling (IEM) [41, 40]. 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 is satisfied. While very preliminary steps on IEM have been taken [40], 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.

7 Conclusion

We presented a formal definition of monitorable operational design domains and a formalization of the problem of learning monitors for the operational design domains of black-box (ML) components. We discussed the need for a quantitative version of the problem and presented a quantitative counterexample-guided learning framework for solving the problem. Our experiments show, how the introduced framework can be used to learn monitors on a monitorable feature space that prevent the system from using a critical component when the system exits its ODD. Learning monitors of high quality requires a lot of effort on the feature engineering side. Furthermore, learning monitors may be subject to different objectives, e.g., accuracy vs efficiency. In the future we plan on investigating the latter problems further with the goal of providing the user with adequate feedback that helps in the selection process of monitors.

Acknowledgments. The authors are grateful to Daniel Fremont for his contributions to the VerifAI and Scenic projects, and assistance with these tools for this paper. The authors also want to thank Johnathan Chiu, Tommaso Dreossi, Shromona Ghosh, Francis Indaheng, Edward Kim, Hadi Ravanbakhsh, Ameesh Shah and Kesav Viswanadha for their valuable feedback and contributions to the VerifAI project.

References

1. Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul W. Fieguth, Xiaochun Cao, Abbas Khosravi, U. Rajendra Acharya, Vladimir Makarenkov, and Saeid Nahavandi. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Inf. Fusion*, 76:243–297, 2021.
2. Dario Amodei, Chris Olah, Jacob Steinhardt, Paul F. Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.
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. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. 62(2), 2015.
5. Marjory S. Blumenthal, Laura Fraade-Blanar, Ryan Best, and J. Luke Irwin. *Safe Enough: Approaches to Assessing Acceptable Safety for Automated Vehicles*. RAND Corporation, Santa Monica, CA, 2020.
6. Luca Bortolussi, Francesca Cairoli, Nicola Paoletti, Scott A. Smolka, and Scott D. Stoller. Neural predictive monitoring and a comparison of frequentist and bayesian approaches. *Int. J. Softw. Tools Technol. Transf.*, 23(4):615–640, 2021.
7. Ian Colwell, Buu Phan, Shahwar Saleem, Rick Salay, and Krzysztof Czarnecki. An automated vehicle safety concept based on runtime restriction of the operational design domain. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1910–1917, 2018.
8. 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.
9. Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, 2017.
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. Thomas G. Dietterich and Eric Horvitz. Rise of concerns about AI: reflections and directions. *Commun. ACM*, 58(10):38–40, 2015.
12. Alexandre Donzé and Oded Maler. Robust satisfaction of temporal logic over real-valued signals. In *FORMATS*, volume 6246 of *LNCS*, pages 92–106. Springer, 2010.
13. 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*, pages 1–16, 2017.
14. 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 (1)*, volume 11561 of *LNCS*, pages 432–442. Springer, 2019.
15. Tommaso Dreossi, Somesh Jha, and Sanjit A. Seshia. Semantic adversarial deep learning. In *CAV*, volume 10981 of *LNCS*, pages 3–26. Springer, 2018.
16. 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.

17. 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.
18. Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods Syst. Des.*, 24(2):101–127, 2004.
19. 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.
20. 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*, pages 63–78. ACM, 2019.
21. 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.
22. 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.
23. Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna M. Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, Muhammad Shahzad, Wen Yang, Richard Bamler, and Xiao Xiang Zhu. A survey of uncertainty in deep neural networks. *CoRR*, abs/2107.03342, 2021.
24. 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.
25. Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
26. The British Standards Institution. Operational design domain (odd) taxonomy for an automated driving system (ads) – specification. *BSI PAS 1883.*, 2020.
27. Patrick Irvine, Xizhe Zhang, Siddartha Khastgir, Edward Schwalb, and Paul Jennings. A two-level abstraction ODDdefinition language: Part i*. In *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, page 2614–2621. IEEE Press, 2021.
28. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224. ACM, 2010.
29. Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.
30. Siddartha Khastgir, Stewart A. Birrell, Gunwant Dhadyalla, and Paul A. Jennings. Calibrating trust through knowledge: Introducing the concept of informed safety for automation in vehicles. *Transportation Research Part C: Emerging Technologies*, 2018.
31. Siddartha Khastgir, Simon Brewerton, John Thomas, and Paul Jennings. Systems approach to creating test scenarios for automated driving systems. *Reliability Engineering and System Safety*, 215:107610, 2021.
32. Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.
33. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Runtime assurance based on formal specifications. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed*

Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA, pages 279–287. CSREA Press, 1999.
34. Anna Lukina, Christian Schilling, and Thomas A. Henzinger. Into the unknown: Active monitoring of neural networks. In *RV*, volume 12974 of *LNCS*, pages 42–61. Springer, 2021.
35. Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *LNCS*, pages 152–166. Springer, 2004.
36. Stefan Mitsch and André Platzer. Modelplex: verified runtime validation of verified cyber-physical system models. *Formal Methods Syst. Des.*, 49(1-2):33–74, 2016.
37. SAE on Road Automated Driving Committee et al. SAE J3016. taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. Technical report, Technical Report.
38. Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties: This time with calls and returns. In Martin Leucker, editor, *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers*, volume 5289 of *LNCS*, pages 51–68. Springer, 2008.
39. 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.
40. Sanjit A. Seshia. Introspective environment modeling. In *19th International Conference on Runtime Verification (RV)*, pages 15–26, 2019.
41. Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016.
42. Lui Sha. Using simplicity to control complexity. *IEEE Softw.*, 18(4):20–28, 2001.
43. Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, USA, 2014.
44. 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.
45. Eric Thorn, Shawn C. Kimmel, and Michelle Chaka. A framework for automated driving system testable cases and scenarios. 2018.
46. Hazem Torfah. Stream-based monitors for real-time properties. In *RV*, volume 11757 of *LNCS*, pages 91–110. Springer, 2019.
47. Hazem Torfah, Sebastian Junges, Daniel J. Fremont, and Sanjit A. Seshia. Formal analysis of ai-based autonomy: From modeling to runtime assurance. In Lu Feng and Dana Fisman, editors, *Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings*, volume 12974 of *LNCS*, pages 311–330. Springer, 2021.
48. Xizhe Zhang, Siddartha Khastgir, and Paul Jennings. Scenario description language for automated driving systems: A two level abstraction approach. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 973–980, 2020.