Radboud University Nijmegen

Faculty of Science

Learning Predictive Monitors for Autonomous Vehicles Using Binary Decision Trees

A Case Study

Thesis BSc Computing Science

Supervisor:
Dr. Sebastian Junges

Author:
Jord Cluitmans

Daily supervisor:
Msc. Luko van der Maas

Second reader:
Dr. Jana Wagemaker

May 2025

Abstract

In safety-critical domains, like autonomous driving, it is a crucial challenge to ensure the safety of autonomous systems. To achieve this, runtime monitors can be used to detect when a system's operations do not adhere to its intended specifications. However, reacting only when a violation is detected, is often too late to prevent unsafe situations. This thesis investigates the use of predictive monitors that raise an alarm before a violation occurs. More specifically, we explore how such monitors can be learned using binary decision trees and how well they perform in different scenarios. Using the Scenic programming language we simulate three different driving scenarios, ranging in complexity, where a driving car must avoid a collision with a parked car. With the results of these simulations, we iteratively train monitors using a variation of the framework proposed by Torfah et al. We define three different measures to evaluate the performance of the monitors: the alarm rate, violation rate, and late alarm rate. The results show that binary decision trees can effectively learn to predict collisions in simple scenarios, achieving a very low violation rate and a stable alarm rate. As complexity of the scenarios increases, however, the monitors tend to overfit and become overly sensitive, leading to very high alarm rates compared to the violation rate. This suggests that while decision trees are interpretable and easy to train, more complex scenarios require alternative classifiers to maintain performance.

Contents

1 Introduction **2**
- 1.1 Outline . . . 3
- 1.2 Background . . . 3

2 Problem Statement **5**
- 2.1 Preliminaries . . . 5
- 2.2 Formal definitions . . . 5
- 2.3 Problem Specification . . . 7
  - 2.3.1 Monitor Quality . . . 7
- 2.4 Proposed solution . . . 9
- 2.5 Scenario specification . . . 10
  - 2.5.1 Observations of Interest . . . 12

3 Experiments **14**
- 3.1 Methods . . . 14
- 3.2 Results . . . 15
  - 3.2.1 Results scenario I . . . 15
  - 3.2.2 Results scenario II . . . 15
  - 3.2.3 Results scenario III . . . 17
  - 3.2.4 Result comparison . . . 18

4 Discussion **19**
- 4.1 Limitations and future work . . . 19
  - 4.1.1 System definition . . . 19
  - 4.1.2 Classifier . . . 19
- 4.2 Conclusion . . . 20

Chapter 1

Introduction

Autonomous systems increasingly rely on machine learning (ML) techniques to solve complex tasks. ML components are particularly well suited for these problems due to their ability to generalize and work with incomplete knowledge [10, 13]. However, the use of these ML components, especially in environments that are critical to safety, has also raised concerns about the robustness and reliability of ML systems [10, 18]. A notable challenge for these ML systems, is that while sets of training data might be large, it is often impossible to cover all possible situations that the systems might encounter. This is especially true for systems that operate in highly dynamic environments, such as autonomous vehicles. Because of this, the systems are very sensitive to sudden changes in input data that are not accounted for during training, like different road markings or weather conditions.

As a consequence, such systems might take actions that do not match their system specification. The system specification is the set of rules that the system should adhere to in order to operate safely. In other words, it specifies the behaviour that the system should exhibit to stay safe. As mentioned, not all situations can be covered during training, because of the dynamic nature of the environments in which these systems operate, which can lead to the system not adhering to its specification. We thus need to find a method to monitor the system's behaviour at runtime, preventing it from taking unwanted or unsafe actions.

In this thesis, we consider the problem above in the context of systems in autonomous vehicles. For instance, consider an autonomous lane-keeping component in a car, trained on large amounts of image data. Changes in factors such as weather conditions or road marking colours, that were not present in the training data can lead to unwanted or even unsafe behaviour. Standards like ISO 26262 [14] have been developed to ensure safety in the automotive domain. However, these standards are often not designed with technologies such as ML-based systems in mind [17]. Thus, alternative approaches are needed to ensure the safe operation of autonomous ML systems in the automotive domain.

Ideally, for safety, these systems should only operate (autonomously) within their so-called operational design domain (ODD). The ODD of a system is the environment in which a system must operate safely [21]. In other words, for a system to operate safely, it must be prevented from exceeding its ODD. Preventing a system from going outside its ODD requires monitoring the boundaries of this ODD at runtime and intervening in the system's operations when these boundaries are met. We thus want to monitor if a

system adheres to its specification at all time. However, reacting only when the system exits its ODD might be too late, as it often means it is too late to intervene and prevent unsafe situations. We therefore want the monitor to be predictive, i.e. raising an alarm when the monitor anticipates the system will exit its ODD before it actually happens and intervening to return the system to a safe state.

Since monitors need to decide based on current and past observations, if something will happen in the future, they can be seen as a form of classifiers. Therefore, one approach to model such a monitor is by using a binary decision tree classifier. An advantage of such a classifier is that it is very interpretable and easy to understand. It is also relatively easy and time-efficient to train, compared to other classifiers such as neural networks.

In this thesis, we apply a variation of the framework developed by Torfah et al. [21] to learn a predictive runtime monitor for a specific scenario. More specifically, we look at how well binary decision tree classifiers can be used to monitor the driving behaviour of a car in specific scenarios. We consider three different scenarios in a range of complexity to evaluate the performance of the monitors. In each scenario, there are two cars present: a driving car and a parked car. The monitor should predict if the driving car will collide with the parked car some prediction horizon d steps in the future. For simplicity, we assume that these cars are the only two vehicles present. To model the scenarios, we use the probabilistic programming language Scenic [7].

1.1 Outline

In this thesis, we start in Chapter 2 by discussing the theoretical background and framework that we will apply in this thesis. We go into what is needed to learn a predictive runtime monitor and how to measure its performance. We then propose our method for learning a monitor based on the framework developed by Torfah et al. [21] in Sections 2.4 and 2.5, and discuss the implementation we will use for our experiments. After that, in Chapter 3 we present the practical implementation of our method and the setup of our experiments. Finally, we present and discuss the results of our experiments in Chapter 3, by comparing the performance of the monitors in different scenarios. Based on these results, we try to draw conclusions and give suggestions for future work.

1.2 Background

Operational Design domains For autonomous systems, especially in safety-critical domains, it is important to clearly establish the operational design domain of the system and communicate the capabilities and limitations of the system to its users [12]. On top of this, Colwell et al. have also concluded the importance of ODDs to be executable, in order to be able to create monitors that can be used during runtime [4]. This is also recommended for on-road motor vehicles by the Society of Automotive Engineers (SAE) [5].

Monitoring ODDs Multiple works have been investigating methods for learning monitors for ODDs of autonomous systems [2, 3, 21]. Some try to find monitors using ensemble methods [20] making use of majority voting or multi-armed bandits. In this study by Torfah et al., ensembles of monitors are generated in the form of decision tree classifiers, from which a final decision is obtained by either a majority vote, or by choosing the action with the least risk (multi-armed bandits). Monitors are generated by training the classifiers on data generated by simulating scenarios in Scenic, similar

to how it is done in this thesis. Other work focus more on finding monitors in cases where input measurements have uncertainties [2]. This is done by using a logic-based approach that tries to measure how well the current state falls within the ODD. Finding a monitor then becomes more a matter of finding the proper logical and mathematical boundaries, than training a model based on generated data.

The literature on monitors also presents monitors in multiple shapes, like logic based systems [2], as mentioned above, and domain-specific frameworks [19]. In the paper by Torens et al., the authors present a monitoring framework based on a formal specification language called RTLola. The method uses a translation from the ODD specification, given in a domain specific language, into RTLola which acts as a real-time stream-based monitoring framework based on certain trigger conditions.

However, these require the ODD to be known and monitorable at runtime, which might not always be the case. Firstly, some aspects of the ODD might be hard or expensive to monitor at runtime, as described by Torfah et al. [21]. Furthermore, we ultimately want our monitor to be predictive, i.e., raising an alarm before a violation occurs. Work by Torfah et al. tackles this problem of learning predictive monitors, using classification and neural network approaches [20, 21].

Learning predictive monitors In this paper, we focus on the method developed by Torfah et al. [21], which uses a counter-example guided approach to find a monitor in the shape of a classifier. Their method uses Scenic in combination with verifai [6] to generate training and testing data and aims to find a monitor in the shape of a decision tree classifier. As a slight variation to this, we do not use verifai, and instead rely solely on Scenic for generating data. We apply the framework to specific scenarios in the automotive domain, where we monitor the driving behaviour of a car. As a classifier, we use binary decision trees to try and see how suitable they are for this task.

Chapter 2

Problem Statement

2.1 Preliminaries

In order to define the problem at hand, we first define some notation on formal languages.

An alphabet Σ is a set, whose elements are referred to as symbols. Words, also referred to as traces, are a sequence of elements of Σ. For a (possibly infinite) alphabet Σ, we define the set of traces over Σ as the set of finite words Σ^*. The set of words of fixed length $d \in \mathbb{N}$ over Σ is defined as Σ^d. A language over Σ is then any set $L \subseteq \Sigma^*$. A language of d-length traces over Σ is then any language $L \subseteq \Sigma^d$. To concatenate two words w and w' we use the notation $w \cdot w'$.

2.2 Formal definitions

In this section we introduce some key definitions. Using those definitions, we formalise the problem statement and specify the scenario on which we are going to apply the problem. Since our work is heavily based on the paper by Torfah et al. [21], we restate most definitions from there.

System behaviour The behaviour of a system with inputs $\mathcal{I}$ and outputs $\mathcal{O}$ can be captured as a sequence of input-output pairs. These pairs capture how the system behaves in response to inputs from its environment, enabling a complete reconstruction of the steps taken. We model this by using the alphabet $\Sigma_{sys} = \mathcal{I} \times \mathcal{O}$ where each symbol represents an input-output interaction. We denote traces over this alphabet with the symbol τ. The system behaviour C is then defined as a language $C \subseteq \Sigma_{sys}^*$, capturing all possible sequences of input-output pairs the system can exhibit.

System specification In order to differentiate a correct system behaviour from unwanted or unintended behaviour, we define a system specification as a language φ. This system specification can be seen as the ground rules for how the system is supposed to operate, i.e., defining the correct system behaviour. It can be captured as a set of traces of input-output pairs that the system behaviour must adhere to. Not all input-output pairs of the system can be observed or observed exactly, e.g., distance sensors may not always give the true distance, or in the simulation only to limited precision. We therefore define an alphabet Σ_{obs} consisting of only the observable part of the input-output pairs. Traces over this alphabet are considered the observable system traces and are

denoted as σ. Note that this implies $\Sigma_{obs} \subseteq \Sigma_{sys}$. We then formally define a system specification as a language $\varphi \subseteq \Sigma^*_{obs}$. The system behaviour C then satisfies our system specification φ if $C \subseteq \varphi$. This is also denoted as $C \vDash \varphi$.

Defining the ODD An ODD D of a system C with respect to φ can be seen as the set of conditions where C is guaranteed to satisfy φ. To define this ODD, we will need three important components. Firstly, the ODD defines guarantees on system behaviour and the system specification; therefore, we require a clear notion of the observable input-output pairs Σ_{obs} upon which φ is defined. Secondly, not all observations are of interest or realistic for defining the ODD. For example, the absolute positions of the cars can be observed during runs of the simulation, but are not realistic features for real life scenarios. Other features like the state of car headlights can be useful for the general functionality of the car, but have little to no impact on collisions during an automated driving procedure. It is therefore needed to have some mapping $obs : \Sigma^*_{sys} \to \Sigma^*_{obs}$ between the system inputs and actions and the observations of interest. Lastly, we want the ODD to guarantee $C \vDash \varphi$ for some amount of time in the future, since only guaranteeing $C \vDash \varphi$ at the current moment is not of much use. For this we use a prediction horizon d.

Definition 1 (Operational Design Domain) *An ODD is defined as a tuple* $D_{C,\varphi} = (\Sigma_{obs}, obs, d)$. *An ODD* D *then defines a set* $[\![D]\!] \subseteq \Sigma^*_{obs}$ *where* $[\![D]\!] = \{\sigma \mid \forall \tau \in C, obs(\tau) = \sigma \to \forall \tau' \in \Sigma^d_{sys}, \tau \cdot \tau' \in \varphi\}$.

Torfah et al. [20] clarifies this definition by saying the ODD D thus defines a set of traces $[\![D]\!]$ where for any system trace τ over Σ_{sys} that induces σ, i.e. $obs(\tau) = \sigma$, a continuation of τ for another d steps, i.e. extending τ with τ' of length d, will result in an execution $\tau \cdot \tau'$, that does not violate the system-level specification φ.

Monitors A predictive monitor M for an ODD D needs to predict if an observation trace σ will fall within $[\![D]\!]$ some steps in the future. The problem is that $[\![D]\!]$ is not known in advance, which we will discuss in Section 2.3. Practically, a monitor would not be able to classify infinite traces. Because of this, we define a monitor on traces of a predefined length l. We will refer to the l-length input traces as the input window for the monitor.

Definition 2 (Monitor) *A runtime monitor* M *is a program that implements a function* $f_M : \Sigma^l_{obs} \to \{0, 1\}$, *for* $l \in \mathbb{Z}^+$.

Thus, a monitor essentially classifies observation traces in two classes; 0 and 1. Monitors can therefore be considered *binary* classifiers, classifying observation traces. Using the definition above, for a perfect monitor, this means that $\forall \sigma \in \Sigma^*_{obs}$, $f_M(\sigma) = 1$ if and only if $\sigma \in [\![D]\!]$. The definition of a monitor, however, does not give any guarantees on this. This is because finding a perfect monitor is often computationally difficult. Using this definition of a monitor also allows for defining measures of monitor quality in ways that work best for the specific application. For a perfect monitor, this would thus imply that $[\![D]\!] = \{\sigma | f_M(\sigma) = 1\}$.

2.3 Problem Specification

The ultimate goal is to find or 'learn' a monitor for a specific scenario. As mentioned in the introduction, we consider three different scenarios. In short, every scenario defines a driving car and a parked car, where the driving car has some autonomous driving behaviour. We elaborate the scenarios in Section 2.5. We aim to find a monitor that predicts if traces of the system will violate its system specification φ after d steps. In this section, we define the problem at hand and specify a quality measure on monitors. After this, we discuss a way to solve this problem.

Classification Perfect monitors provide a function that can classify if observation traces violate the system specification. Taking a step back, we see that by finding such a monitor for a system, we are therefore actually identifying $[\![D]\!]$. Since we saw that monitors act as binary classifiers, the problem now becomes determining whether an observation trace $\sigma \in \Sigma^*_{obs}$ belongs to the language $[\![D]\!]$. However, our definition of a monitor does not provide any guarantees on how well it is at classifying traces. Because of this, the first problem is to find any monitor at all.

Finding a monitor The problem of finding any monitor at all is of course trivial, since the definition only requires a monitor to be a function from traces to $\{0, 1\}$. This means that simply providing a function that always returns 0, regardless of the trace, would be a valid monitor. Such a monitor however would obviously not be useful. The initial monitor should therefore try to classify traces as well as possible. To do so, we will try and train a classifier on labelled traces of the system. Although this still gives few guarantees on the quality of the monitor, it at least provides a starting point that can be improved upon by providing it with more training data.

Perfect monitors Optimally, we would like to find a monitor that perfectly captures the ODD of a system. This however has many practical complications. Firstly, it is often computationally hard to learn a perfect classifier. This is because finding a perfect monitor would require the monitor to see every possible trace of the system, which is often infeasible. Secondly, the systems we consider, like other real-world systems, have intricate temporal dependencies that lead to non-Markovian properties [9], that can be difficult to learn.

2.3.1 Monitor Quality

Monitor implementation In order to assess the quality of a monitor, we evaluate the monitor in the context of the system. This means we implement the monitor in the system simulation. Implementing the monitor is done by giving real-time inputs to the monitor during simulation executions, and implementing some behaviour that will be executed once the monitor raises an alarm. We call this the fall-back behaviour of the system. In the scenarios we define in Section 2.5, this behaviour is implemented by simply hitting the brakes of the driving car. Using this implemented monitor, we can assess the performance of the monitor using different measures that are defined below.

Misclassifications Misclassifications by the monitor are unavoidable. The misclassifications of a monitor can be either false negative or false positive. False negatives are traces, where the monitor does not raise an alarm but the specification φ is violated d steps later, with d the prediction horizon. False positives, on the other hand, are traces of C where the monitor does raise an alarm, but d steps later the system specification is not violated. We denote both false negatives and false positives as counterexamples,

since they are examples where the monitor does not act as we would like it to. In safety critical domains, false negatives mean that critical failures are missed, potentially leading to serious consequences. False positives on the other hand, while disruptive, often lead to less severe outcomes. Because of this, monitors should favour false positives over false negatives [21].

Optimal monitors Given the limitations of monitors and the potential for misclassifications, we prefer to find a monitor with *optimal* quality. To do so, we first need to define what the quality of a monitor entails. There are multiple ways to go about this. Intuitively, a good monitor is one which makes the system more 'safe', i.e., that leads to fewer violations of the system specification. Using this definition, however, a monitor that always raises an alarm would be the most optimal, since it would never allow the system to execute and thus, the system will never violate its specification. Such a monitor however would be very disruptive, and not very useful in practice. We thus want to find proper measures that encompass both the safety of the system and the disruptiveness of the monitor. To do so, we look at the two different measures below. By combining these two measures and finding a balance between them, we can assess the quality of a monitor.

Violation rate The percentage of system runs where a system specification violations occurs, provides a measure on the safety of the system. We call this measure the *violation-rate*. In contrast to the number of false negatives, the violation-rate also provides information on how safe a monitor is, given its implementation. Consider again a scenario of two cars driving on a road, with a predictive monitor trying to prevent collisions. When a monitor in this scenario raises an alarm, a reasonable response might be to simply hit the brakes to try and stop the car and prevent a collision. When a moving car brakes, it does not stop instantaneously, but rather slows down over time. It would therefore be possible that a monitor raises an alarm, tries to stop the car as fast as possible, but the car still ends up crashing. Such a situation would not necessarily be a false negative of the monitor, since it did raise an alarm d steps before the specification was violated. Given the implementation of the monitor however, this situation should still be accounted for when measuring the safety of the overall system, which the violation-rate takes care of. The lower the violation-rate, the safer the system is and the higher the quality of the monitor.

Alarm rate In order to make sure the monitor does not just always raise an alarm, we also measure the percentage of system runs where an alarm gets raised, i.e., the *alarm-rate*. We prefer that alarms get raised as little as possible, since false positives can be troublesome. From a practical standpoint however, checking for false positive alarms in our simulation is not possible. It would require running the exact same simulation where the monitor is implemented and where it is not, and then comparing the results. The problem is that we can only set the initial seed for the scenario, from which all scenes will be sampled. The driving behaviour we use in our simulation is partially dependent on random sampling. Since the car stops driving when an alarm is raised, fewer numbers will have been sampled after a simulation where an alarm was raised, meaning the next sampled scene will start at a different point than if the monitor had not been active. Because of this, the results of the two simulations cannot be compared, since from the second scene onwards, there is no guarantee that they will indeed be the same exact simulation run.

Another, more theoretical, reason to not only look at the number of false positives, is that even true positive alarms can be disruptive when raised in low-risk situations.

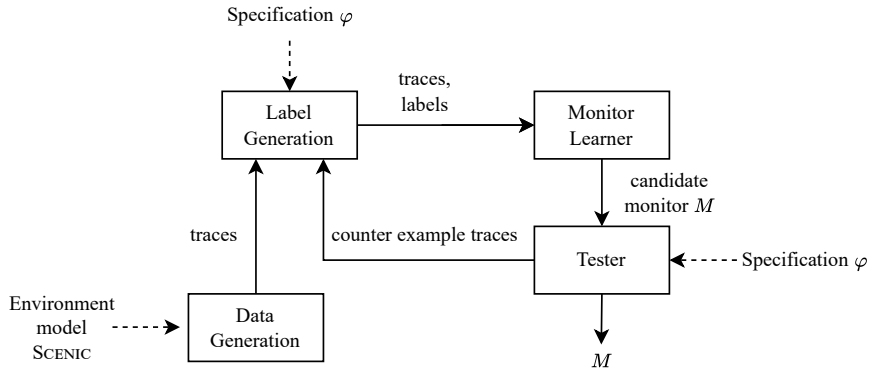


Figure 2.1: Workflow for finding a runtime monitor. In the first iteration of the loop, traces are generated from a SCENIC scenario, labels are given to the traces and both are then sent to a monitor learning algorithm. The monitor is then applied to new runs of the system and the traces obtained from this are again compared to the system level specification in order to filter counterexamples. From there the cycle keeps going until the monitor meets pre-specified quality requirements.

Our definition of the ODD says that a trace for which any continuation of d steps leads to a violation of the system should be excluded from the ODD. If for a certain trace only 0.1% of all continuations end up in a violation, the monitor should still raise an alarm according to the definition of the ODD. In practice however, it might be acceptable to allow the system to execute in such situations, in order to prevent unnecessary disruptions. The alarm-rate thus gives more insight in the disruptiveness of the monitor to the system. Together with the number of specification violations, this measure gives a representation of the quality of the monitor, that is also easily interpretable for humans.

Monitor quality We ultimately aim to find a monitor with the lowest alarm-rate possible, without compromising the safety of the system, as measured by the violation-rate. It is important to note that the optimal values and the ratio between them for these measures might differ depending on the system, the system specification and the implementation of the monitor. For some systems, the disruptiveness is not that important compared to the safety of a system, while in other cases system violations are more acceptable than unnecessary disruptions. Because of this, it is hard to define when the quality of a monitor is optimal. Instead, we aim to first find any monitor at all, and then improve the quality of the monitor until the quality is satisfactory, or until we are no longer able to improve the quality. Using this method, we cannot give any guarantees on if the monitor is optimal, but we can say something about how good the monitor is in general.

2.4 Proposed solution

In the following section, we discuss our proposed way to solve the problems described above. We first look at how we go about finding a monitor in the first place and what type of classifier could be used for this. We then discuss a method to evaluate and iteratively improve the monitor using a counterexample guided refinement approach.

Combined, this method allows us to find a monitor that is of sufficient quality for the system at hand.

Finding a monitor Like explained previously, the monitors we try to find will act as a binary classifier. In order to find an initial monitor, we first need to generate a dataset on which we can train a classifier. This will be done by modelling the system in a simulation environment and generating multiple runs of the system. The resulting system traces will be split up into input windows of a fixed size, as explained in Section 2.2. Every input window is labelled with a binary value, indicating whether the original system trace satisfies the system specification φ d steps after the last time step of the input window. The resulting classifier will thus try to predict, based on input windows, if a system violation will occur d steps later.

Classifier In this paper we will use a binary decision tree (BDT) as our classifier. Decision trees are suited tools for classification, especially since they are non-parametric. This means that the model does not make any assumptions about the distribution of the data. This is useful in our case, since there are no guarantees on the distribution of the system traces we generate. Another useful feature of decision trees is that they are easy to interpret. This is useful to quickly identify why a monitor might be making certain decisions, and to debug the monitor if necessary.

Quality evaluation Once an initial monitor has been found, we will try to assess and improve its quality. To do so, we implement the monitor in the simulation environment as explained in Section 2.3.1. We then run the simulation with the monitor active, and again record the resulting system traces. We can measure the violation- and alarm-rate of these newly obtained traces in order to evaluate the quality of the monitor.

Counterexample guided refinement We improve upon the initial monitor, we will use a counterexample guided refinement approach. First, the new system traces obtained by running the simulation with the monitor active are analysed. We then extract the counterexamples from these traces, according to the definition in Section 2.3.1. These counterexamples are labelled according to the same system as before and added to the initial dataset. The new dataset is then used to train a new classifier. This leads to a new classifier that can again be evaluated using the violation- and alarm-rate. This learning cycle is repeated until the monitor is either of sufficient quality, or when the monitor does not improve any more.

2.5 Scenario specification

In this paper we cover three specific cases of the monitor problem. The difference in the scenarios lies mainly in how difficult it is to find a suitable monitor for the system. In all three scenarios, we define a parked car, which does not move at all, and a so-called ego car, also referred to as the driving car. The driving car is given an autonomous lane keeping and driving behaviour, which is the system we aim to monitor. To learn the monitors for these scenarios, we define certain observations of interest, which will act as the input features for our classifier. In this section we define the scenarios and observations of interest in more detail.

Definition of scenario I The first and simplest scenario sets the ego car on the centreline of a random lane. The parked car is put directly in front of the driving car, with a random offset to the left or right. We define an additional constraint to this

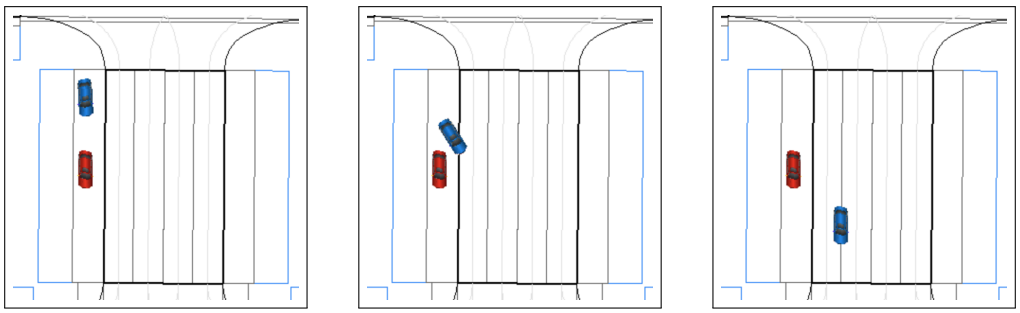
scenario, where if the two cars are in the same lane, they always eventually crash. The other way around, we also define a constraint that if the two cars are in different lanes, they will never crash. Furthermore, for this scenario we also add a feature to the driving car that represent if the driving car and the parked car are in the same lane or not. This feature will be used on top of the observations of interest defined in below Section 2.5.1 This thus defines a very simple and relatively unrealistic scenario. This scenario is mainly used as a proof of concept, to see if the algorithm is able to find a monitor for a very simple system. We would expect the decision trees for this scenario to be very simple, consisting of only 1 split, based on the same lane feature. Such a monitor would perfectly capture all traces of the system, and we thus expect no violations to occur when the monitor for this scenario is applied.

Definition of scenario II The second scenario is similar to scenario I, but a little less restrictive. We randomly place both cars on the centreline of a road, with the parked car with in visible distance of the driving car. Note that both starting lanes are sampled separately, hence they are not necessarily the same. We remove the constraints about the lanes of the cars, which means it is possible that the driving car accidentally switches lanes during a run of the simulation. On top of that, we remove the same lane feature from the inputs to the monitor. This means the monitor is only able to use the observations of interest defined below to make decisions.

Definition of scenario III In the third and most complex scenario, the two cars are parked on the same side of the road, with the ego car behind the parked car. The ego car will autonomously try to drive away from the parking spot, onto the road. For this scenario we again only use the observations of interest defined below. The difficulty in this scenario lies mostly in the fact that the driving car is trying to move out of the parking spot behind the parked car. This means the car is exhibiting more steering actions than in the other two scenarios. The monitor therefore needs to take the rotation and steering angle of the car more into account when making decisions.

System-level specification The autonomous driving component of the driving car is considered to be the system C, of which we want to monitor the ODD $[\![D]\!]$. For simplicity, we define the system-level specification φ such that the only rule for the system is not to crash into another object on the road. This definition is of course much simplified compared to real-world scenarios, but it is sufficient for the purpose of this thesis. Furthermore, by defining the system-level specification in this way, checking if the system satisfies the specification is relatively easy. For more complicated scenarios, this might be harder to do in the simulation. The algorithm for finding the monitor however, does not change when the system-level specification is more complex. It should therefore be possible to extend our method of finding a monitor to more complex specifications, without much alteration.

Operational design domain Using the system C and its specification φ as defined above, we can define the operational design domain $D_{C,\varphi} = (\Sigma_{obs}, obs, d)$. The alphabet Σ_{obs} in our scenario consists of the observable in- and output features of the system. Executions of the system will induce traces $\tau \in \Sigma^*_{sys}$. The function obs is then the function that extracts the observations of interest as defined in Section 2.5.1 from these traces and turns traces $\tau \in \Sigma^*_{sys}$ into traces $\sigma \in \Sigma^*_{obs}$.



(a) $t = 0$ (b) $t = 15$ (c) $t = 40$

Figure 2.2: A render of one simulation of scenario III at different time steps using the Newtonian simulator from SCENIC. The ego car is blue, the parked car is red. The thicker black lines surround a road segment in the network, where the thinner lines inside indicate the different lanes and their centreline. The road section in this scenario ends in a crossing on each end, as can be seen by the curving lines at the top and bottom of the image. The black lines on the left and right of the road section indicate denote the shoulders of the road, with the blue boxes next to them indicating the pavement.

2.5.1 Observations of Interest

Observations of interest The observable system traces consisting of the observations of interest are captured by traces $\sigma \in \Sigma^*_{obs}$. We will use a total of 5 observable features that will be used to try and learn the monitor; d_{left}, d_{right}, θ, dv_{left}, and dv_{right}, as detailed below.

Distance Because of our definition of φ, an obvious first useful observation is the distance between the driving car and any other object. For this purpose, cars often use parking sensors, fitted at multiple locations on the car. In this scenario, we will fit two 'parking sensors' on the front-left and front-right of the driving car. The simulation of our scenario is an idealised version of a realistic scenario. The sensors are therefore able to track distance 360 degrees around them. As can be seen in figure 2.2, the front of the driving car is most likely to collide with the parked car when driving, hence fitting the parking sensors there makes the most sense. These sensors will, for every step of the simulation, return the absolute distance between the sensor and the parked car. We will call these observations d_{left} and d_{right} for the distances measured from the front left and the front right of the car respectively.

Steering angle Another observation that is important for the monitor is the steering angle of the driving car. Consider the situation in figure 2.2a. It looks like the car is headed straight towards the parked car, and might crash into it. As seen in figure 2.2b however, the car steers away and avoids a collision. The steering angle of the car at any given point, thus gives insight into the heading of the car and can therefore partly predict if a collision will happen. We will denote this steering angle as θ.

Relative speed Distance and steering angle alone however, do not paint the full picture A car might steer directly towards the parked car, with a very small distance between the driving car and the parked car, but if the car is stationary, there is still no risk at collision. We therefore also need the speed of the car. More specifically, we consider the relative speed dv of the driving car, compared to the parked car. Relative speed can be calculated as the difference in distance between two objects over a given

time span. In this case the two objects are the two cars in the scenario. Since the distance between the cars is taken from two points on the car, the front left and front right, we can calculate the relative speed from both of these points as well. We denote the relative speed from the front left and front right of the driving car compared to the parked car as dv_{left} and dv_{right} respectively. For both of these points, the relative speed can be computed as follows:

$$dv = \frac{d_t - d_{t-n}}{n}, \text{ with } t, n > 0 \text{ and } t \geq n$$

Since we are interested in the relative speed at every step of the simulation, we take $n = 1$. This allows us to ignore the division by n leaving us with $dv = d_t - d_{t-1}$. This represents the difference in distance between two consecutive time steps. The relative speed in our case can therefore also be seen as the closing-rate of the driving car in comparison to the parked car. The reason for taking relative speed over the absolute speed, is because it more accurately describes how much closer the two cars are getting over time. It tells us not only something about the speed of the car, but also about its heading and the distance between the driving and parked car. The driving car can have a high velocity, but if it is not heading towards the parked car, the risk of collision might be very low. Relative speed is therefore a more precise and relevant measure for predicting collisions.

Chapter 3

Experiments

3.1 Methods

Scenic programming language The experiments are run using the probabilistic programming language Scenic [7]. Scenic is a domain-specific language for modelling and simulating specific scenarios. Scenic programs define distributions over spatial and temporal configurations of objects that are generated as scenes. We use this in order to generate many different runs of the scenarios we define, without having to manually define every new run of the system.

Scenario implementation For our definition of scenario III, we create two versions of this scenario: one where the cars start on the left side of the road and one where they start on the right. This way, we ensure that the monitor is not exclusively trained on a single method of driving out of parking spots. We then implement the counterexample guided refinement method as described in section 2.4. To do so, we create a new version of all the scenarios in Scenic, but now with the monitor implemented. The scenarios will thus be identical, however now the driving car will start braking when an alarm is raised. In Scenic we make use of the built-in Newtonian simulator. In this simulator, two cars that collide will not interact with each other at all, and simply pass through each other. This is of course not very realistic, but more importantly, it leads to situations where the two cars are intersecting at a certain timestamp, but d steps later, they are not intersecting anymore. This means some windows where the cars are intersecting get labelled as windows where nothing is wrong, which is a bit counterintuitive. To counteract this, we implement a simple behaviour for the driving car, that instantly brings it to a standstill once its bounding box intersects with that of the parked car.

Labelling traces In order to train the classifier, we split the system traces into input windows of a fixed size l, which we take to be 5. The first l steps of the system therefore do not have a complete input window. For simplicity, and in order to provide somewhat relevant input data to the monitor also at the start of the simulation, we decide to duplicate the first observation of the system l times, thus filling the entire input-window with the same observation. Every input window is then labelled with a binary value, indicating whether the original system trace satisfies the system specification φ d steps after the last time step of the input window. These input windows act as the input data for the classifier. For our experiments we set d to be 10, which corresponds to 1 second in the simulation.

Binary Decision Tree For our experiment, we used the binary decision tree (BDT) classifier from the *Scikit-learn* module for Python [15]. In order to get the best results, we optimize the hyperparameters of the classifier using built-in functions from the *Scikit-learn* module once at the start of the program.

Gini impurity The decision tree classifier in our experiment uses the Gini impurity [16] as a splitting criterion. The Gini impurity can intuitively be seen as the probability that two randomly chosen samples from one dataset have different labels [23]. This criterion is thus defined over the distribution of an entire dataset. During every learning cycle of the algorithm used in our experiment, new data is generated and added to the training data for this classifier. This newly generated data will influence the distribution of the dataset and therefore the values of the Gini impurity. To counter this, instead of using an incremental learning strategy [22], where new data is added to the existing classifier and the classifier is tuned based on the new data, we simply retrain the entire model. Retraining with all the data also has the benefit that it prevents catastrophic interference [8], which is the tendency of neural networks to forget old information upon learning new information.

System specifications All experiments are run on a MacBook Air M3 with 8 cores and 16 GB RAM. The macOS version used for running the experiments is Sequoia 15.4.1. All code can be found on GitLab[1].

3.2 Results

We ran the algorithm on the three scenarios we defined. Per iteration of the learning cycle, we generated 2000 new traces of the system to be used for training the monitor. We ran the learning cycle for 50 cycles each, and tracked the quality measures of the monitor over time. Next to the alarm- and violation-rate, we also tracked the late alarm-rate. We define a trace to have a late alarm, if there is a violation, but no alarm is raised by the monitor d steps before the violation occurs. The late alarm-rate is then the percentage of system traces where a late alarm occurs. This measure thus tells something about how well the fall-back behaviour of the system is implemented.

3.2.1 Results scenario I

In figure 3.1, we see that no violations occur during the entire training loop for scenario I. This is very well explained by looking at the scenario definition, and the shape of the decision tree in 3.2. The decision tree simply splits on the 'same lane' feature, which according to the scenario definition should indeed always yield the correct result. The fact that the alarm rate is not completely stable, is due to the fact that SCENIC randomly samples scenes that adhere to the scenario definition.

3.2.2 Results scenario II

Monitor quality For scenario II, the results are less straight forward. We again see that the alarm rate is relatively stable, apart from the expected variation due to random sampling of scenes. Interestingly, after about 20 iterations of the learning cycle, the violation rate suddenly drops to almost 0%. A possible explanation for this is described below. The fact that the alarm rate is relatively constant, but the violation rate seems to decrease over time, implies that the monitor learns to be less sensitive over time.

[1] https://gitlab.science.ru.nl/jcluitmans/Bachelor-Thesis.git

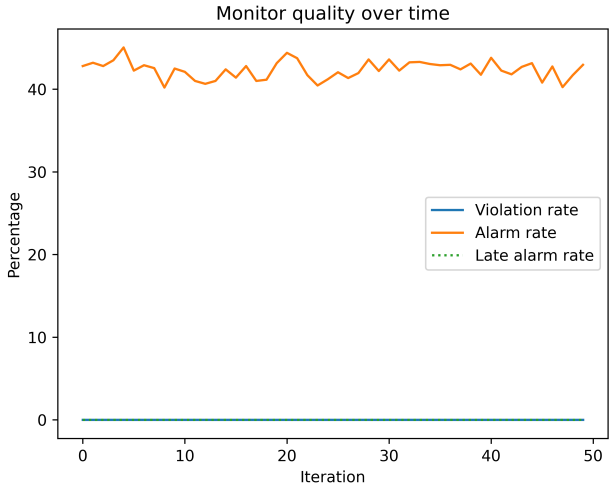
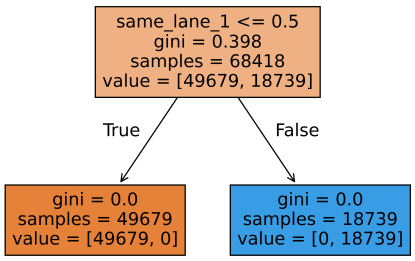
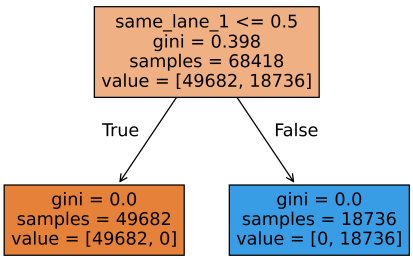


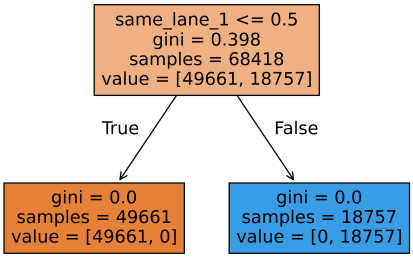
Figure 3.1: Results of the learning cycle for scenario I. The alarm-rate, violation-rate and late alarm-rate are plotted per iteration of the learning cycle.



(a) Decision tree for scenario I at the start of the learning cycle.



(b) Decision tree for scenario I after 25 cycles of the learning cycle.



(c) Decision tree for scenario I after 50 cycles of the learning cycle.

Figure 3.2: Decision trees for scenario I at different points in the learning cycle.

Late alarms We find that during the entire learning cycle, every violation is the result of a late alarm. This implies that the fall-back behaviour of the monitor works as intended, and ensures that no violations occur if the monitor raises an alarm in time.

Decision trees The decision trees for this scenario are too complex to legibly be represented in this document. The initial tree consists of 9 layers, and 133 nodes, while

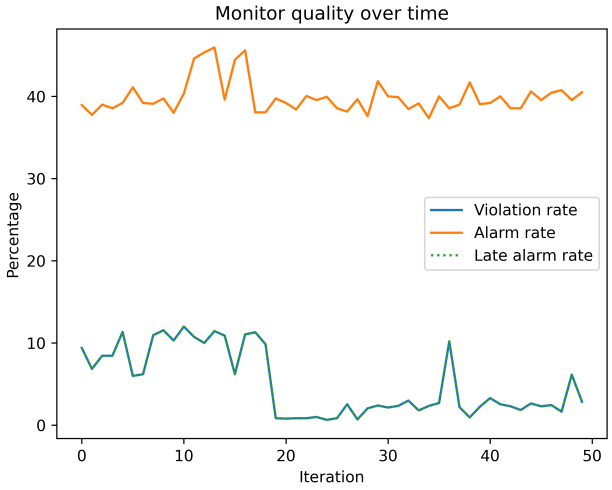


Figure 3.3: Results of the learning cycle for scenario II. The alarm-rate, violation-rate and late alarm-rate are plotted per iteration of the learning cycle.

the final tree consists of 9 layers and 213 nodes. When we look at the size of the trees over time, we notice that at iteration 19, the tree suddenly grows by about 70 nodes. This probably explains the sudden dip in the violation rate that can be seen in figure 3.3. Looking at the data distribution in the trees, in the lower layers of the tree, both classes still seem to be very much present. This can be explained by the fact that the scenario is a little more complex than scenario I, and the fact that the monitor does not get the same lane feature any more. It thus needs to look at more different features in the input windows to make decisions.

3.2.3 Results scenario III

Monitor quality For scenario III, the monitor again leads to relatively few violations. The monitor does however seem to be overfitting a lot to the violations in the training data. This can be seen due to the fact that the alarm rate, especially towards the end of the learning cycle, is very high, almost reaching 100% at certain iterations. This probably means that the monitor is getting too sensitive, and simply always raises an alarm. As explained before in section 2.3.1, such a monitor is obviously very safe, but also not very useful. Interestingly, during the start of the learning cycle, both the violation rate, and the alarm rate seem to be relatively low. Because of this we could consider the overall quality of the monitor better at the start of the learning cycle, than towards the end.

Decision Trees The decision trees for this scenario again are too complex to be represented in this document. The trees do again grow over time, with the initial tree consisting of 9 layers and 213 nodes, and the final tree consisting of 9 layers and 415 nodes. Interestingly, the sudden dip in violation rate around iteration 23 this time does not seem to be caused by an increase in tree size, since the trees between iteration 20 and 30 all consist of around 380 nodes. This implies the dip instead is caused by the monitor learning a new distinction between the two classes.

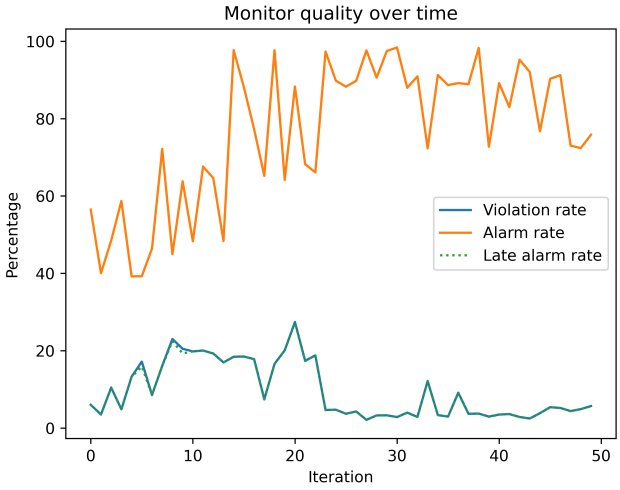


Figure 3.4: Results of the learning cycle for scenario III. The alarm-rate, violation-rate and late alarm-rate are plotted per iteration of the learning cycle. The late alarm rate entirely overlaps with the violation rate, which makes hard to see in the figure.

3.2.4 Result comparison

In table 3.1, we compare the initial and final violation rates for the three scenarios. The final violation rates are obtained by running the system with the final monitor at the end of the learning cycle active on the same set of traces that were used to train the initial monitor, i.e. the system without the monitor active. This way, we can fairly compare how well the monitor prevents collisions in the system. We also include the final alarm and late alarm rate from these traces in the table in order to give a more complete overview of the quality of the monitor.

	Initial violation rate (%)	Final violation rate (%)	Final alarm rate (%)	Final late alarm rate (%)
Scenario I	27.40	0.00	27.40	0.00
Scenario II	38.40	2.50	40.15	2.50
Scenario III	35.48	5.40	74.13	5.40

Table 3.1: Comparison of the initial and final violation rates for the three scenarios.

Overall we can conclude that the monitor is perfectly able to learn scenario I, which is also what we expected. For scenario II we find that monitor is relatively accurately able to prevent violations. Most importantly for this scenario, we find that the final alarm rate is very similar to the initial violation rate, which implies a very low number of false positives. For scenario III, the monitor is overfitting a lot. This is visible in the extremely high alarm rate, which is about double the initial violation rate.

Chapter 4

Discussion

4.1 Limitations and future work

The experiment setup used has some shortcomings. In the following section we will discuss the limitations and imperfections of the experiments conducted in this paper.

4.1.1 System definition

Driving behaviour The behaviour that we monitor in this paper is a relatively straight-forward lane following behaviour. It is a simple algorithm that tries to keep the driving car in the middle of the lane. It would be interesting to see if the method we applied would work similar for more complex behaviours, like more advanced machine learning systems.

System specification Similarly, the system behaviour we defined was an overly simplified version of a real-world scenario. In reality, the system specification would need to take much more factors into account, like speed-limits, other traffic, lighting and weather conditions. For future work it would be interesting to explore how more complex system specifications would affect the quality of the monitor using similar methods.

4.1.2 Classifier

In this paper, we represent a monitor as a binary decision tree (BDT). The BDT implementation of *Scikit-learn* uses univariate feature selection based on the Gini impurity function. Univariate feature selection means that splits in the tree are always based on one feature at a time. In the context of system monitoring, this can be problematic. In some instances, violation of the system specification by a trace σ might be dependent on multiple features at the same time, like distance and speed. Furthermore, the univariate splitting of decision trees can be problematic when some input variables are numerically related [1]. In the experiment described in this paper, the distance and relative speed attributes, as defined in 2.5.1, have such a numerically dependent relationship. A potential consequence of this is that in order to capture complex relationships between features, univariate splits will need multiple splits to distinguish certain patterns. This can lead to overfitting and unnecessarily complex trees. Furthermore, the numeric relationship of some input variables can make it difficult for a univariate splitting tree to capture complex patterns. As a result, the classifier might only capture shallow patterns in the data.

Multivariate feature selection Multivariate splitting could be used as a solution to these problems. The downside to this is that such models are computationally way more expensive [1]. Besides, multivariate splitting can also lead to overfitting, when the classifier focusses too much on the relationship between input attributes when this is not necessary. On top of that, pruning multivariate decision trees to prevent overfitting is often also more complex compared to univariate decision trees [1]. To find if multivariate splits perform better than the binary decision trees to solve the monitor problem remains a question for future research.

Types of classifiers We saw in our experiments that the BDT classifier quickly has more difficulty when scenarios get more complex. BDTs are also known to get overly complex by e.g. creating duplicate subtrees [11]. This can often lead to overfitting, which is a problem we also encountered in our experiments. For future work, it would be interesting to see if other classifier methods, like random forests and neural networks could improve the performance of the monitor.

4.2 Conclusion

In this paper we applied a variation of the method developed by Torfah et al. [21] to learn a predictive monitor for a simple driving behaviour in three different scenarios, using a binary decision tree classifier as a monitor. We found that for very simplified scenarios, the decision tree classifier is very suitable for monitoring these type of systems. Once the scenarios start becoming a little more realistic, and therefore complex, decision tree classifiers are found to be less suitable. For these scenarios, we found that binary decision trees start to overfit a lot which results in very disruptive monitors. To be able to properly monitor such scenarios, more research is needed to determine if other classifiers are more suitable.

Bibliography

[1] Carla E. Brodley and Paul E. Utgoff. "Multivariate Decision Trees". In: *Mach. Learn.* 19.1 (1995), pp. 45–77.

[2] Thibault Charmet et al. "Operational Design Domain Monitoring with Uncertain Measurements". In: *27th IEEE International Conference on Intelligent Transportation Systems (ITSC 2024)*. 2024.

[3] Thibault Charmet et al. "Overview of the Operational Design Domain Monitoring for Safe Intelligent Vehicle Navigation". In: *ITSC*. IEEE, 2023, pp. 5363–5370.

[4] Ian Colwell et al. "An Automated Vehicle Safety Concept Based on Runtime Restriction of the Operational Design Domain". In: *Intelligent Vehicles Symposium*. IEEE, 2018, pp. 1910–1917.

[5] SAE On-Road Automated Driving Committee and ISO TC204/WG14. *SURFACE VEHICLE RECOMMENDED PRACTICE Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. 2021.

[6] Tommaso Dreossi et al. "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems". In: *CAV (1)*. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 432–442.

[7] Daniel J. Fremont et al. "Scenic: a language for scenario specification and data generation". In: *Mach. Learn.* 112.10 (2023), pp. 3805–3849.

[8] Atith Gandhi et al. "Natural Mitigation of Catastrophic Interference: Continual Learning in Power-Law Learning Environments". In: *ECAI*. Vol. 392. Frontiers in Artificial Intelligence and Applications. IOS Press, 2024, pp. 2886–2893.

[9] Maor Gaon and Ronen I. Brafman. "Reinforcement Learning with Non-Markovian Rewards". In: *AAAI*. AAAI Press, 2020, pp. 3980–3987.

[10] Mohamad Gharib et al. "On the Safety of Automotive Systems Incorporating Machine Learning Based Components: A Position Paper". In: *DSN Workshops*. IEEE Computer Society, 2018, pp. 271–274.

[11] Lisandro Arturo Jimenez-Roa, Tom Heskes, and Mariëlle Stoelinga. "Fault Trees, Decision Trees, And Binary Decision Diagrams: A Systematic Comparison". In: *CoRR* abs/2310.04448 (2023).

[12] Siddartha Khastgir et al. "Calibrating trust through knowledge: Introducing the concept of informed safety for automation in vehicles". In: *Transportation Research Part C: Emerging Technologies* 96 (2018), pp. 290–303.

[13] Zeshan Kurd, Tim Kelly, and Jim Austin. "Developing artificial neural networks for safety critical systems". In: *Neural Comput. Appl.* 16.1 (2007), pp. 11–19.

[14] R. Palin et al. "ISO 26262 Safety Cases: Compliance and Assurance". In: *6th IET International Conference on System Safety 2011*. IET, 2011, B12.

[15] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830.

[16] Lior Rokach and Oded Maimon. “Top-down induction of decision trees classifiers - a survey”. In: *IEEE Trans. Syst. Man Cybern. Part C* 35.4 (2005), pp. 476–487.

[17] Rick Salay, Rodrigo Queiroz, and Krzysztof Czarnecki. “An Analysis of ISO 26262: Using Machine Learning Safely in Automotive Software”. In: *CoRR* abs/1709.02435 (2017).

[18] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. “Toward verified artificial intelligence”. In: *Commun. ACM* 65.7 (2022), pp. 46–55.

[19] Christoph Torens et al. “From Operational Design Domain to Runtime Monitoring of AI-Based Aviation Systems”. In: *2024 AIAA DATC/IEEE 43rd Digital Avionics Systems Conference (DASC)*. IEEE, 2024, pp. 1–9.

[20] Hazem Torfah et al. “Learning Monitor Ensembles for Operational Design Domains”. In: *RV*. Vol. 14245. Lecture Notes in Computer Science. Springer, 2023, pp. 271–290.

[21] Hazem Torfah et al. “Learning Monitorable Operational Design Domains for Assured Autonomy”. In: *ATVA*. Vol. 13505. Lecture Notes in Computer Science. Springer, 2022, pp. 3–22.

[22] Gido M. van de Ven, Tinne Tuytelaars, and Andreas S. Tolias. “Three types of incremental learning”. In: *Nat. Mac. Intell.* 4.12 (2022), pp. 1185–1197.

[23] Ye Yuan, Liji Wu, and Xiangmin Zhang. “Gini-Impurity Index Analysis”. In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 3154–3169.