Safety and Reliability of DRL Agents Through Testing and Safety Monitoring

by

Amirhossein Zolfagharian

Thesis submitted to the Faculty of Engineering
In partial fulfillment of the requirements
For the Doctorate of Philosophy degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Deep Reinforcement Learning (DRL) agents have shown significant promise across various domains, including autonomous driving, healthcare, and robotics. However, their deployment in safety-critical applications presents substantial concerns regarding their safe and reliable behavior. The complexity and unpredictability of DRL environments, combined with their objective of maximizing long-term rewards, can lead to unintended safety violations.

This thesis proposes two complementary methods aimed at improving the safety and reliability of DRL agents: (1) a pre-deployment testing approach called *STARLA* (Search-based Testing Approach for Reinforcement Learning Agents) and (2) a runtime safety monitoring approach known as *SMARLA* (Safety Monitoring Approach for Reinforcement Learning Agents).

The first component, *STARLA*, addresses the challenges of systematically testing DRL agents by employing a search-based strategy that aims to reveal functional faults—situations where the agent may encounter an unsafe state. *STARLA* uses state abstraction, machine learning models, and evolutionary algorithms to efficiently generate test episodes that expose functional faults, within a limited simulation budget.

The second component, *SMARLA*, focuses on runtime safety by predicting potential safety violations at runtime. *SMARLA* is agnostic to the inputs of the DRL agent and is a black-box approach. By continuously observing the agent's behavior through the analysis of Q-values and leveraging state abstraction, *SMARLA* enables timely predictions before safety violations occur.

Together, *STARLA* and *SMARLA* form a comprehensive framework for improving both pre-deployment quality assurance and runtime safety of DRL agents.

Finally, the proposed approaches have been extensively evaluated on complex case studies and through large-scale experiments. Empirical results demonstrate the effectiveness of these approaches in identifying and mitigating safety risks.

Acknowledgements

This research was supported by a grant from General Motors, along with funding from the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC). Lionel Briand's contribution was partially funded by Science Foundation Ireland under grant 13/RC/2094-2. We would like to thank Haq *et al.* [51] for kindly sharing the implementation of their search algorithm, which significantly aided our study.

Additionally, we are grateful to GM R&D scientists, including Arun Adiththan, Prakash Mohan Peranandam, Md Mhafuzul Islam, Ramesh Sethu, and Rouhollah Jafari, for their invaluable feedback and assistance in implementing part of our work in a large-scale case study. Their expert insights and support were instrumental in enhancing the quality and scope of this research.

Dedication

First and foremost, I would like to express my deepest gratitude to Prof. Lionel Briand for his invaluable guidance, mentorship, and support throughout the course of my research. His insights and feedback have significantly shaped both the direction and quality of this work, making the completion of this thesis possible.

I would like to extend my sincere appreciation to Dr. Manel Abdellatif, for her collaboration and continuous support in the two studies presented in this thesis: A Search-Based Testing Approach for Deep Reinforcement Learning Agents and SMARLA: A Safety Monitoring Approach for Deep Reinforcement Learning Agents. Her contributions and dedication have been instrumental in advancing these research efforts. Also, thank you for enduring my countless "What if we try this instead?" messages at odd hours.

I am also grateful to my collaborators, including Mojtaba Bagherzadeh, for his help in developing the search-based testing approach and his valuable guidance during my first year of PhD.

A special thanks to Ramesh S, and General Motors R&D team for his insightful contributions and for providing me the opportunity to intern and learn about the challenges faced in industry.

Lastly, I would like to express my deepest love and gratitude to my beloved wife, Zohreh. Her unwavering patience, encouragement, and belief in me have been a constant source of strength throughout this challenging journey. She has been my greatest supporter, offering me not only emotional comfort but also her wisdom and understanding during the most difficult times. Her sacrifices and enduring support behind the scenes have been immeasurable, and I cannot thank her enough for her love, companionship, and resilience. Without her by my side, this journey would have been far more difficult, and her presence has made the completion of this thesis all the more meaningful. She also deserves extra credit for tolerating my late-night "Eureka" moments that were followed by me pacing around the house like a mad scientist.

I am also deeply thankful to my family and friends for their constant encouragement, belief in me, and steadfast support, without which this work would not have been possible. To my family: thanks for frequently asking, "So, are you a doctor now or what?"

Finally, I would like to dedicate a heartfelt message to my younger brother, whose curiosity, determination, and energy have been a constant source of inspiration for me. His belief in my abilities, coupled with his own achievements and potential, reminds me of the importance of perseverance and passion. I hope this thesis shows you that persistence (and a good Wi-Fi connection) can help achieve almost anything. Just remember: when it's your turn, I'll be here to remind you to write your acknowledgments — and to proofread them with the same love and mischief you've shown me.

AI Artificial Intelligence

AV Autonomous Vehicle

DNN Deep Neural Network

DRL Deep Reinforcement Learning

MDP Markov Decision Process

ML Machine Learning

Q-values Action-Value Function in Reinforcement Learning

RL Reinforcement Learning

SMARLA Safety Monitoring Approach for Reinforcement Learning Agents

STARLA Search-based Testing Approach for Reinforcement Learning Agents

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Reinforcement Learning (RL) has seen significant advancements in recent years across various domains such as autonomous driving, healthcare, and robotics, driven by its potential to solve complex real-world problems. RL algorithms enable agents to learn optimal policies through interaction with their environments, aiming to optimize long-term cumulative rewards. The advent of Deep Reinforcement Learning (DRL), which integrates Deep Neural Networks (DNNs) to learn optimal policies, has further expanded the capabilities of RL systems [73, 116, 139]. DRL agents can handle high-dimensional inputs and learn directly from raw data, making them suitable for a wide range of real-world applications [11, 88]. However, the increasing complexity and deployment of these agents in safety-critical areas raise substantial concerns about their safety and reliability.

One of the primary challenges in deploying DRL agents is the lack of guarantees regarding their safe behavior [33, 138]. DRL agents, driven by the goal of maximizing long-term rewards, may inadvertently violate safety requirements [7], especially in environments where the learned policies may conflict with safety requirements. For instance, a DRL agent controlling a self-driving car might prioritize speed over adherence to traffic rules, posing significant risks to passengers and pedestrians. This issue is exacerbated by the fact that traditional testing methods are often inadequate for thoroughly evaluating these agents due to the vast state spaces and the probabilistic nature of their environments.

To address these challenges, there is a critical need for comprehensive approaches towards ensuring the safety of DRL agents. This thesis explores two complementary methods to achieve this goal, (1) a pre-deployment testing approach and (2) a runtime safety monitoring solution.

The first component focuses on testing DRL policies. DRL systems requires effective and systematic testing prior deployment, especially when used in safety-critical applications [117]. Specifically, deploying a reinforcement learning agent in autonomous driving systems raises major safety concerns as we must pay attention not only to the extent to which the agent's objectives are met but also to damage avoidance [34]. Thus, two types of

faults can be defined in a DRL context: functional and reward faults. The former happens when an RL agent takes an action that leads to an unsafe state such as running a red light. The latter occurs when an agent does not reach the desired reward (e.g., reaching destination late). Functional and reward faults are often in tension, as we can obtain a high reward while observing a functional fault. For instance, an agent might perform an unsafe action, such as ignoring a stop sign, to reach a destination faster, thereby achieving a higher reward. This tension complicates the detection and resolution of faults, especially in well-trained agents where failures are rare. Given the critical nature of safety in applications like autonomous driving, our focus is on detecting functional faults.

Another challenge in this domain is the cost associated with simulations and real-time testing. High-fidelity simulations that accurately replicate real-world environments can be computationally expensive and time-consuming. Meanwhile, real-time testing, especially in safety-critical applications like autonomous driving, poses significant risks and logistical challenges. These factors necessitate more efficient and realistic testing methodologies. Moreover, traditional testing approaches, such as adversarial attacks, often fail to realistically simulate the operational environment of DRL agents and primarily aim to test robustness rather than functionality.

Therefore, the motivation behind this research is to develop a more efficient and effective approach to testing DRL systems, specifically focusing on identifying functional faults prior to deployment.

Thus, the first thesis objective (TO_1) is defined as follows:

TO_1 (Search-based Testing of Deep Reinforcement Learning Agents): The goal is to develop a method that effectively searches for failing episodes within a limited testing budget. This approach focuses on identifying functional faults in DRL agents to improve their safety before deployment.

1.2 Runtime Safety Monitoring

While pre-deployment testing is critical [100], it cannot anticipate all possible scenarios that DRL agents may encounter once deployed. Real-world environments are often dynamic and unpredictable, which makes it difficult to detecting all safety problems in advance. As a result, there is a need for continuous safety monitoring during runtime to ensure that the agent behaves safely in unforeseen situations.

This need is especially critical in safety-sensitive domains like autonomous driving, where undetected unsafe behaviors can have severe consequences. No testing method can guarantee that all potential safety violations will be identified, making it essential to monitor the agent's behavior in real-time to detect and mitigate potential risks as they arise.

The motivation for this aspect of the research is to ensure that DRL agents can continue to operate safely after deployment by providing an early warning system that predicts safety violations in real-time.

Thus, the second thesis objective (TO_2) is defined as follows:

TO_2 (Runtime Safety Monitoring of Deep Reinforcement Learning Agents): The objective is to develop a black-box runtime safety monitoring approach capable of predicting safety violations early, providing sufficient time for corrective actions to be taken during operation.

1.3 Research Contributions

To address the challenges related to the pre-deployment testing of DRL agents, we propose *STARLA*, a Search-based Testing Approach for Reinforcement Learning Agents. *STARLA* is a data-box[1] testing method, utilizing state abstraction and machine learning to efficiently identify and generate episodes that reveal functional faults in DRL agents. This approach is particularly beneficial in scenarios where access to the internal workings of the DRL system is limited, but training data is available. *STARLA* demonstrates its effectiveness by significantly improving fault detection over random testing, as shown in well-known RL benchmarks like *Cart-Pole* and *Mountain-Car*.

To address the challenges related to the runtime safety monitoring of DRL agents, we introduce *SMARLA*, a black-box Safety Monitoring Approach for Reinforcement Learning Agents. Even with rigorous pre-deployment testing, unsafe scenarios can arise during real-world operation that were not considered during the development phase. *SMARLA* provides continuous safety monitoring by predicting potential safety violations in real-time, without requiring access to the internal mechanisms of the agent. It uses Q-values, which reflect the potential rewards and penalties of actions, to detect when the agent may violate safety boundaries. *SMARLA* is highly versatile and applicable to a wide range of problems with different input data types. It offers timely predictions, ensuring that interventions can be made before any real harm occurs.

STARLA and *SMARLA* contribute to assuring the safety of DRL agents, addressing both pre-deployment testing and runtime safety monitoring. This thesis thus makes a substantial contribution to the broader field of artificial intelligence safety, providing tools and methods that improve the reliability of intelligent systems operating in complex environments.

1.3.1 Summary of Contributions

- ***STARLA* (A Search-based Testing Approach for Reinforcement Learning Agents)**:
  - A novel search-based testing approach that reveals functional faults in DRL agents.

[1]Data-box testing requires access only to the training dataset, without needing access to the model's internals.

  - Focuses on identifying unsafe actions (functional faults) rather than simply reward faults.
  - Uses data-box testing, state abstraction, and machine learning to efficiently generate fault-revealing episodes.
  - Outperforms random testing on well-known RL benchmarks (Cart-Pole and Mountain Car).

- **SMARLA (A Safety Monitoring Approach for Reinforcement Learning Agents)**:
  - A runtime safety monitoring approach that predicts safety violations in real-time.
  - Operates as a black-box system using Q-values, without requiring internal access to the agent.
  - Agnostic to the type of input data, making it suitable for a wide range of applications, such as image, voice and textual data.
  - Offers early predictions of potential safety violations to enable timely interventions.

These contributions collectively advance the field of trustworthy AI, offering both practical tools and theoretical insights into the safe deployment of DRL systems.

1.4 Dissemination of Work

The findings and contributions of this thesis have been disseminated through several channels, including:

- **Conference Presentations:** The search-based testing approach results were presented at the *International Conference on Software Engineering (ICSE 2024)* in the Journal First track, where the novel contributions to testing DRL agents were discussed and positively received by the community. Additionally, preliminary results on runtime safety monitoring were showcased at the *ICSE 2024* [155] poster track.

- **Journal Publications:** The search-based testing approach was published in *2023 IEEE Transactions on Software Engineering (TSE)* [156], which is widely seen as the top journal in software engineering, providing a comprehensive analysis of the proposed methodology and its effectiveness in real-world applications. The work on runtime safety monitoring has been accepted for publication in *IEEE Transactions on Software Engineering (TSE)* [157].

- **Open-Source Tools:** Both the testing and runtime monitoring frameworks have been made publicly available as open-source tools on Github, including the datasets and configurations necessary for replicating the experiments presented in this thesis. The type of work presented in this thesis required the design of complex tooling and experiments.

1.5 Thesis Outline

The rest of this thesis is structured as follows:

- Chapter 2 addresses TO_1 by providing:
  - Introduction and motivation for testing DRL agents.
  - Background and definitions to establish the context of the thesis.
  - Proposal of our search-based testing approach.
  - A thorough empirical evaluation of the proposed testing approach.
  - Discussion of practical implications recommendations, and threats to validity.
  - Review of related work and comparison with our proposed technique.
  - Conclusion and suggestions for future research directions.
- Chapter 3 starts addressing TO_2 by providing:
  - Introduction and motivation for runtime safety monitoring of DRL.
  - Proposal of our black-box Safety monitoring approach.
  - Qualitative and quantitative analysis of the proposed safety monitoring approach.
  - Potential threats to the validity of the study.
  - Review of the related work and comparison with our proposed technique.
  - Conclusion of our work and suggestions for future work.
- Finally, Chapter 4 provides reflections on the outcomes of this thesis and outlines potential future research directions. It also discusses the broader impact of the proposed approaches, *STARLA* and *SMARLA*, in enhancing the safety and reliability of DRL agents in real-world applications.

Chapter 2

Search-based Testing of Deep Reinforcement Learning Agents

This chapter focuses on Search-based Testing of Deep Reinforcement Learning Agents (TO_1). The content of this chapter has been published in *2023 IEEE Transactions on Software Engineering (TSE)* [156] under the title of *A Search-Based Testing Approach for Deep Reinforcement Learning Agents*.

2.1 Overview

As discussed in chapter 1, safety of DRL systems should be assessed prior deployment, especially when used in safety-critical applications. One of the ways to assess the safety of DRL agents is to test them to detect possible faults leading to critical failures during their execution. By definition, a fault in DRL-based systems corresponds to a problem in the RL policy that may lead to the agent's failure during execution. Since DRL techniques use DNNs, they inherit the advantages and drawbacks of such models, making their testing challenging and time-consuming. Furthermore, DRL-based systems are based on a Markov Decision Process (MDP) [111] that makes them stateful. They embed several components including the agent, the environment, and the ML-based policy network. Testing a stateful system that consists of several components is by itself a challenging problem. It becomes even more challenging when ML components and the probabilistic nature of the real-world environments are considered.

There are three types of testing approaches for deep learning systems that depend on the required access levels to the system under test: white-box, black-box, and data-box testing [77, 137]. White-box testing requires access to the internals of the DL systems and its training dataset. Black-box testing does not require access to these elements and considers the DL model as a black box. Data-box testing requires access only to the training dataset.

Prior testing approaches for DL systems (including DNNs) have focused on black-box and white-box testing [58, 60, 101, 105, 136, 149], depending on the required access level to

the system under test. However, limited work has been done on testing DRL-based systems in general and using data-box testing methods in particular [60, 101, 105, 129]. Relying on such types of testing is practically important, as testers often do not have full access to the internals of RL-based systems but do have access to the training dataset of the RL agent [24].

Most existing works on testing DRL agents are based on adversarial attacks that aim to perturb states of the DRL environment [79]. However, adversarial attacks lead to unrealistic states and episodes, and their main objective is to test the RL agents' robustness rather than test the agents' functionality (e.g., functional safety). In addition, a white-box testing approach for DRL agents has been proposed that focuses on fault localization in the source code of DRL-based systems [101]. However, this testing approach requires full access to the internals of the DRL model, which are often not available to testers, especially when the DRL model is proprietary or provided by a third party. Also, localizing and fixing faults in the DRL source code do not prevent agent failures due to imperfect policies and the probabilistic nature of the RL environment. Furthermore, because of the huge state space, the high cost of test execution, and the black-box nature of DNN models (policy networks), exhaustive testing of DRL agents is impossible.

In this research, we focus on testing the policies of DRL-based systems using a data-box testing approach, and thus address the needs of many practical situations. We should remind that we focus our analysis on functional faults since they are more critical. Thus, the detection of reward faults is left for future work and is out of the scope of this research. We propose STARLA, a Search-based Testing Approach for Reinforcement Learning Agents, that is focused on testing the agent's policy by searching for faulty episodes as effectively as possible. An episode is a sequence of states and actions that results from executing the RL agent. To create these episodes, we leverage evolutionary testing methods and rely on a dedicated genetic algorithm to identify and generate functional-faulty episodes [57]. We rely on state abstraction techniques [1, 5] to group similar states of the agent and significantly reduce the state space. We also make use of ML models to predict faults in episodes and guide the search toward faulty episodes. We applied our testing approach on two Deep-Q-Learning agents trained for the widely known *Cart-Pole* and *Mountain Car* problems in the OpenAI Gym environment [48]. We show that our testing approach outperforms Random Testing, as we find significantly more faults.

Overall, the main contributions of this research are as follows:

- We propose STARLA, a data-box search-based approach to test DRL agents' policies by detecting functional-faulty episodes.
- We propose a highly accurate machine learning-based classification of RL episodes to predict functional-faulty episodes, which we use to improve the guidance of the search. This is based in part on defining and applying the notion of abstract state to increase learnability.
- We applied STARLA on two well-known RL problems. We show that STARLA outperforms the Random Testing of DRL agents as we detect significantly more faults

when considering the same testing budget (i.e., the same number of the generated episodes).

- We provide online[1] a prototype tool for our search-based testing approach as well as all the needed data and configurations to replicate our experiments and results.

The remainder of the chapter is structured as follows. Section 2.2 presents the required background and establishes the context of our research. Section 2.3 describes our research problem. Section 2.4 presents our testing approach. Section 2.5 reports our empirical evaluation and results. Section 2.6 discusses the practical implications of our results. Section 2.7 analyzes the threats to validity. Finally, sections 2.8 and 2.9 contrast our work with related work and conclude the research, respectively.

2.2 Background

Reinforcement Learning trains a model or an agent to make a sequence of decisions to reach a final goal. It is therefore a technique gaining increased interest in complex autonomous systems. RL uses trial and error to explore the environment by selecting an action from a set of possible actions. The actions are selected to optimize the obtained reward. In the following, we will describe an example application of RL in autonomous vehicles.

2.2.1 Definitions

To formally define our testing framework, we rely on a running example and several key concepts that are introduced in the following sections.

A running example. Assuming an Autonomous Vehicle (AV) cruising on a highway, an RL agent (i.e., AV) receives observations from an RGB camera (placed in front of the car) and attempts to maximize its reward during the highway driving task, which is calculated based on the vehicle's speed. The agent is given one negative/positive reward per time step when the vehicle's speed is below/above 60 MPH (miles per hour). Available actions are turning right, turning left, going straight, and no-action. The cruising continues until one of the termination criteria is met: (1) the time budget of 10 seconds is consumed, or (2) a collision has occurred.

Definition 1. *(RL Agent Behavior.)*

The behavior of an RL agent can be captured as a Markov Decision Process [111] $\langle S, A, T, R, \gamma \rangle$ where S and A denote a set of possible states and actions accordingly, $T : S \times A \times S \rightarrow [0, 1]$ refers to the transitions function, such that $T(s', a, s)$ determines the probability of reaching state s' by performing action a in state s, $R : S \times A \rightarrow [0, R_{max}]$ is a reward function that determines the immediate reward for a pair of an action and a state,

[1] https://github.com/amirhosseinzlf/STARLA

and $\gamma \in [0, 1]$ is the discount factor indicating the difference of short-term and long-term reward [126].

The solution of an MDP is a policy $\pi : S \rightarrow A$ that denotes the selected action given the state. The agent starts from the initial state ($s_0 \in S$) at time step $t = 0$ and then, at each time step (t_i, $i \geq 0$), it takes an action ($a_i \in A$) according to the policy π that results in moving to a new state s_{i+1}. Also, r_i refers to the reward corresponding to the action a_i and state s_i that is obtained at the end of the step t_i. Note that there may not be a reward at each step, which in that case is considered to be zero. Finally, $\sum^i r_i$ refers to the accumulative reward until step t_i.

Definition 2. *(Episodes.)* An episode e is a finite sequence of pairs of states and actions, i.e., $[(s_j, a_j)|s_j \in S, a_j \in A, 0 \leq j \leq n, n \in \mathbb{N}]$, where the state of the first pair is an initial state, and the state of the last pair is an end state. An end state is, by definition, a state in which the agent can take no more action. The accumulative reward of episode e is $\sum^{|e|} r$, where $|e|$ denotes the length of the episode. We refer to the accumulated reward of episode e with r'_e. A valid episode is an episode where each state is reachable from the initial state with respect to the transition function presented in definition 1. Moreover, the episode is executable (i.e., consistent with the policy of an agent) if starting from the same initial state and in each state, the selected action of the agent is consistent with the action in the episode that we want to execute.

Definition 3. *(Faulty state.)* A faulty state is a state in which one of the defined requirements (e.g., the autonomous vehicle must not hit obstacles) does not hold, regardless of the accumulated reward in that state. A faulty state is often an end state. In the context of the running example, a faulty state is a state where a collision occurs.

Definition 4. *(Faulty Episode.)* We define two types of faulty episodes:

- ***Functional fault:*** If an episode e contains a faulty state, it is considered as a faulty episode of type functional. A functional fault may lead to an unsafe situation in the context of safety-critical systems (e.g., hitting an obstacle in our running example).
- ***Reward fault:*** If the accumulative reward of episode e is less than a predefined threshold ($r'_e \leq \tau$), it is considered a faulty episode of type reward (i.e., the agent failed to reach the expected reward in the episode). Intuitively, regarding our running example, if we assume a reward fault threshold of $\tau = 100$, then each episode with a reward below 100 is considered to contain a reward fault. In our running example, this occurs when the AV agent drives at 25 MPH all the time. As we mentioned earlier, the detection of this type of fault is out of the scope of this research and is left for future work.

2.2.2 State Abstraction

State abstraction is a means to reduce the size of the state space by clustering similar states to reduce the complexity of the investigated problem [1, 5]. State abstraction can be

defined as a mapping from an original state $s \in S$ to an abstract state $s^\phi \in S^\phi$

$$\phi : S \rightarrow S^\phi \tag{2.1}$$

where the abstract state space is often much smaller than the original state space. Generally, there are three different classes of abstraction methods in the RL context [64,75]:

1. π^*-irrelevance abstraction: s_1 and s_2 are in the same abstraction state $\phi(s_1) = \phi(s_2)$, if $\pi^*(s_1) = \pi^*(s_2)$, where π^* represents the optimal policy.

2. Q^*-irrelevance abstraction: $\phi(s_1) = \phi(s_2)$ if for all available actions $a \in A$, $Q^*(s_1, a) = Q^*(s_2, a)$, where $Q^*(s, a)$ is the optimal state-action function that returns the maximum expected reward from state s up to the final state when selecting action a in state s.

3. Model-irrelevance abstraction: $\phi(s_1) = \phi(s_2)$ if for any action $a \in A$ and any abstract state $s^\phi \in S_\phi$, $R(s_1, a) = R(s_2, a)$ and the transition dynamics of the environment are also similar, meaning that $\sum_{s' \in \phi^{-1}(s^\phi)} T(s', a, s_1) = \sum_{s' \in \phi^{-1}(s^\phi)} T(s', a, s_2)$ where $T(s', a, s)$ returns the probability of going to state s' from state s performing action a, as defined in definition 1.

As we are testing RL agents in our work, we use the Q^*-irrelevance abstraction method in this study because it represents the agent's perception. We also choose this abstraction method because it is more precise than π^*-irrelevance. Indeed, π^*-irrelevance only relies on the predicted action (i.e., the action with the highest Q^*-value) to compare two different states, which makes it coarse. In contrast, Q^*-irrelevance relies on Q^*-values for all possible actions.

To clarify, assume two different states in the real world for which our trained agent has the same Q-values. Given the objective to test the agent, it is logical to assume these states to be similar, as the agent has learned to predict identical state-action values for both states (i.e., the agent perceives both states to be the same).

Further, abstraction methods can be strict or approximate. Strict abstraction methods use a strict equality condition when comparing the states of state-action pairs, as presented above. Although they are more precise, they bring limited benefits in terms of state space reduction. In contrast, more lenient abstraction methods can significantly reduce the state space, but they may yield inadequate precision. Approximate abstractions relax the equality condition in strict abstraction methods to achieve a balance between state space reduction and precision. For example, instead of $Q^*(s_1, a) = Q^*(s_2, a)$, approximate abstraction methods use the condition $|Q^*(s_1, a) - Q^*(s_2, a)| < \epsilon$, where ϵ is a parameter to control the trade-off between abstraction precision and state space reduction.

Another important property is transitivity, as transitive abstractions use a transitive predicate. For example, assume that two states, s_1 and s_2, are similar based on an abstraction predicate and the same is true for s_2 and s_3. Then, we should be able to conclude that s_1 and s_3 are similar. Transitive abstractions are efficient to compute and preserve the

near-optimal behavior of RL agents [1]. Moreover, this property helps to create abstract states more effectively.

Considering the properties that we explained previously, we use the following abstraction predicate ϕ_d that is transitive and approximates the Q^*-irrelevance abstraction:

$$\phi_d(s_1) = \phi_d(s_2) \equiv \forall a \in A : \left\lceil \frac{Q^*(s_1, a)}{d} \right\rceil = \left\lceil \frac{Q^*(s_2, a)}{d} \right\rceil \tag{2.2}$$

where d is a control parameter (abstraction level) that can squeeze more states together when increasing and thus reduce the state space significantly. Intuitively, this method discretizes the Q^*-values with buckets of size d.

2.3 Problem Definition

In this research, we propose a systematic and automated approach to test a DRL agent. In other words, considering a limited testing budget, we aim to exercise the agent in a way that results in detecting faulty episodes, if possible. This requires finding faulty episodes in a large space of possible episodes while satisfying a given testing budget, defined as the number of executed episodes.

2.3.1 RL Agent Testing Challenges

Since DRL techniques use DNNs, they inherit the advantages and drawbacks of DNNs, making their testing challenging and time-consuming [80, 122–124]. In addition, RL techniques raise specific challenges for testing:

- **Functional faults.** The detection of functional faults in DRL systems is challenging because relying only on the agent's reward is not always sufficient to detect such faults. Indeed, an episode with a functional fault can reach a high reward. For example, by not stopping at stop signs, the car may reach its destination sooner and get a higher reward if the reward is defined based on the arrival time. Even if we consider a penalty for unsafe actions, we can still have an acceptable reward for functional-faulty episodes. Relying only on the agent's reward makes it challenging to identify functional faults.

- **State-based testing with uncertainty.** Most traditional ML models, including DNNs, are stateless. However, DRL techniques are based on an MDP that makes them stateful and more difficult to test. Also, an RL agent's output is the result of an interaction between the environment (possibly consisting of several components, including ML components) and the agent. Testing a system with several components and many states is by itself a challenging problem. Accounting for ML components and the probabilistic nature of real-world environments makes such testing even more difficult [33, 137].

- **Cost of test execution.** According to the previous discussion, testing an RL agent requires the execution of test cases by either relying on a simulator or by replaying the captured logs of real systems. The latter is often limited since recording a sufficient number of logs that can exhaustively include the real system's behavior is impossible, especially in the context of safety-critical systems, for which logs of unsafe states cannot (easily) be captured. Thus, using a simulator for testing DRL agents, specifically in the context of safety-critical domains, is often inevitable. Despite significant progress made in simulation technology, high-fidelity simulators often require high-computational resources. Thus, testing DRL agents tends to be computationally expensive [28, 33, 60].
- **Focus on adversarial attacks.** Most existing works on testing DRL agents use adversarial attacks that are focused on perturbing states [58]. However, such attacks lead to unrealistic states and episodes [45]. The main goal of such attacks is to test the robustness of RL policies rather than the agents' functionality [26, 60, 153].

The exhaustive testing of DRL agents is impossible due to the large state space, the black-box nature of DNN models (policy networks), and the high cost of test execution. To address these challenges, we propose a dedicated search-based testing approach for RL agents that aims to generate as many diverse faulty episodes as possible. To create the corresponding test cases, we leverage meta-heuristics and most particularly genetic algorithms that we tailor to the specific RL context.

2.3.2 Assumptions

In this work, we focus on testing RL agents with discrete actions and a deterministic policy interacting with a stochastic environment. A discrete action setting reduces the complexity of the problem in defining genetic search operators, as we will see in the following sections. It also reduces the space of possible episodes. Moreover, assuming a deterministic policy and stochastic environment is realistic because in many application domains (specifically in safety-critical domains), randomized actions are not acceptable and environments tend to be complex [35]. We further assume that we have neither binary nor noisy rewards [154] [112] (i.e, where an adversary manipulates the reward to mislead the agent) since the reward function should provide guidance in our search process. We build our work on model-free RL algorithms since they are more popular in practice and have been extensively researched [104, 127].

2.4 Approach

Genetic Algorithms (GA) are evolutionary search techniques that imitate the process of evolution to solve optimization problems, especially when traditional approaches are ineffective or inefficient [8]. In this research, as for many other test automation problems, we use genetic algorithms to test RL agents. This is accomplished by analyzing the episodes

performed by an RL agent to generate and execute new episodes with high fault probabilities from a large search space.

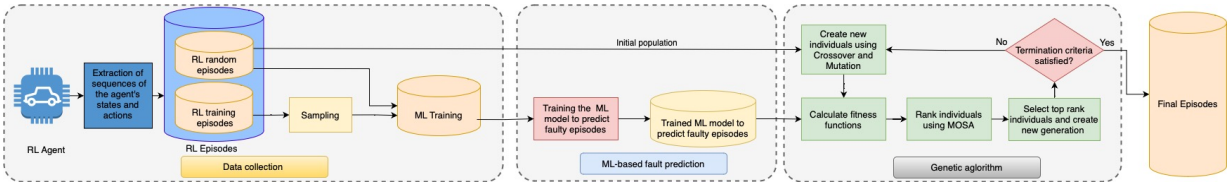

Figure 2.1: Overview of STARLA

2.4.1 Reformulation as a Search Problem

We are dealing with a high number of episodes represented as sequences of pairs (Definition 2), which are executed to test an RL agent. To properly translate the process into a search problem using a genetic algorithm, we need to define the following elements.

- **Individuals.** Individuals consist of a set of elements called genes. These genes connect together and form an individual that is a solution. Here, individuals are episodes complying with Definition 2, which should ideally have a high probability of failure. Naturally, each gene is represented by a pair of state and action.

- **Initial population.** A set of individuals forms a population. In our context, a population is a set of episodes. However, it is imperative for the search to select a diverse set of individuals to use as the initial population. The sampling process is detailed in section 2.4.3.

- **Operators.** Genetic operators include crossover, mutation, and selection [147], which are used to create the next generation of episodes. In the crossover, we use two episodes as input and create a new offspring that hopefully has a higher fault probability. We use the current population as the input and select an episode for the crossover (using tournament selection). The selection of such an episode is in relation to its fitness. We then select a crossover point randomly, search for a matching episode, and join it with the selected episode.

  Mutation is an operator that adds diversity to our solutions. An episode is selected using the tournament selection again. Then, one pair is randomly selected as the mutation point, which is altered according to a defined policy that is detailed in section 2.4.5.2.

  Selection is the last operator used in each generation. It combines the episode from the last generation with the newly created episode in a way that does not eliminate good solutions from previous generations. More detailed explanations are provided in section 2.4.5.3.

- **Fitness function.** The fitness function should indirectly capture how likely an episode is to be faulty. To that end, we define a multi-objective fitness [98] function

that we use to compare episodes and select the fittest ones. As further explained in section 2.4.4, we consider different ways to indirectly capture an episode's faultiness: (1) the reward loss, (2) the predicted probability of observing functional faults in an episode based on machine learning, and (3) the certainty level for the actions taken in an episode.

- **Termination criteria.** This determines when the search process should end. Different termination criteria can be used, such as the number of generations or iterations, the search time, and convergence in population fitness. For the latter, the search stops when there is no improvement above a certain threshold over a number of newly generated episodes.

2.4.2 Overview of the Approach

As depicted in Figure 2.1, the main objective of STARLA is to generate and find episodes with high fault probabilities to assess whether an RL agent can be safely deployed.

To apply a genetic algorithm to our problem, we first need to sample a diverse subset of episodes to use as the initial population. In the next step, we use dedicated genetic operators to create offspring to form the new population. Finally, using a selection method, we transfer individuals from the old population to the new one while preserving the diversity of the latter. For each fitness function, we have a threshold value, and a fitness function is satisfied if an episode has a fitness value beyond that threshold. We repeat this process until all fitness functions are satisfied, or until the maximum number of generations is reached.

Algorithm 1 shows a high-level algorithm for the process previously described, based on a genetic algorithm. Assuming P is the initial population, that is, a set of *episodes* containing both faulty and non-faulty episodes, the algorithm starts an iterative process, taking the following actions at each generation until the termination criteria are met (lines 3-16). The search process is as follows:

1. We create a new empty population P_{new} (line 5).
2. We create offspring using crossover and mutation, and add newly created individuals to P_{new} (lines 6-11).
3. We calculate the fitness of the new population (line 12) and update the archive α and the condition $FitSatisfied$ capturing whether all fitness functions are satisfied (line 13). The archive contains all solutions that satisfy at least one of our three fitness functions (line 14).
4. If all fitness functions are satisfied, we stop the process and return the archive (i.e., the set of solutions that satisfy at least one fitness function). Otherwise, the population P is updated using the selection function, and then we move to the next generation (lines 15-16).

Algorithm 1: High-level genetic algorithm

Input: A set of episodes (P) as the initial population, solution archive α
Output: The updated archive α containing faulty episodes

$gen \leftarrow 0$
$\alpha \leftarrow \emptyset$
while $FitSatisfied = False \quad and \quad gen \leq g$ **do**
    $Fitness = Fit(P)$
    $P_{new} \leftarrow \emptyset$
    $rand = Random(0, 1)$
    if $rand < c$ **then**
        $t_1, t_2 \leftarrow Cross(P)$
        $P_{new} \leftarrow P_{new} \cup \{t_1, t_2\}$
    $t_m \leftarrow Mut(P \cup \{t_1, t_2\}, m)$
    $P_{new} \leftarrow P_{new} \cup t_m$
    $Fitness = Fit(P_{new})$
    $Update(FitSatisfied)$
    $Update(\alpha)$
    $P \leftarrow Select(P, P_{new})$
    $gen \leftarrow gen + 1$
Return α

Furthermore, in our genetic search algorithm, we set the crossover rate c to 75% and the mutation rate m to the $\frac{1}{V}$, where V is the length of the selected episodes for mutation based on the suggested parameters for genetic algorithms in the literature [106]. In the following, we discuss each step of the search in detail.

2.4.3 Initial Population

The initial population of our search problem is a set containing $|P|$ episodes. We use random executions of the agent to build the initial population of the generic search. Consequently, we initiate the environment with different initial states in which we randomly change the alterable parameters of the environment when available (e.g., changing the weather or time of the day in the running example, or changing the starting position of the car). We execute the RL agent starting from the randomly selected initial states and store the generated episodes as the initial population of the search. This increases the diversity of episodes and helps the genetic algorithm explore the solution space more effectively.

2.4.4 Fitness Computations

A fitness function quantitatively assesses the extent to which an individual fits the search objectives and is meant to effectively guide the search. Recall that our objective is to find faulty episodes that can exhibit functional faults. We therefore define the following three fitness functions that complement one another to capture the extent to which an episode

is close to being faulty. We consider (1) the agent reward as a fitness function, as it guides the search toward low-reward episodes, which are more likely to lead to functional faults, if designed properly, as the reward may capture some of the agent's unsafe behavior; (2) the probability of functional faults, which complement the reward and can provide additional guidance toward functional faults if the estimation of such probabilities is accurate (see section 2.3.1 for more explanation); and (3) the certainty level that guides the search toward episodes where the agent is highly uncertain about the selected action.

Next, we define our fitness functions and illustrate their relevance through scenario examples (sections 2.4.4.1, 2.4.4.2, and 2.4.4.3). Specifically, we explain how each fitness function contributes to guiding the search toward faulty episodes. We then explain in sections 2.4.4.4, 2.4.4.5, 2.4.4.6, and 2.4.4.7 the different steps of applying machine learning to estimate the probability of functional faults, which is one of our fitness functions. Finally, we explain our search objectives in section 2.4.4.8.

2.4.4.1 Reward

The first fitness function in our search is meant to drive the search toward episodes with low reward. The reward fitness function of an episode is defined as follows.

$$f_1(e) = r'_e \tag{2.3}$$

where r'_e is the accumulated reward of an episode e.

In the initial population, the rewards of selected episodes are known. Also, when genetic operators are applied, we calculate the reward of new individuals using the reward function according to Definition 1. The search aims to minimize the reward of episodes over generations to guide the search toward finding faulty episodes.

Example: A well-defined reward function which is correlated to functional faults can provide valuable guidance toward the identification of such faults. For instance, in our running example, if the reward function incorporates a penalty based on the car's proximity to the border of the lane or other vehicles, minimizing the reward function would result in the car being driven dangerously close to other lanes, thereby increasing the risk of accidents. Such low-reward episodes are of particular interest, as applying crossover and mutation (section 2.4.5) to them can reveal potential functional faults in the later generations. However, it is worth mentioning that designing a reward function that is capable of capturing all possible functional faults remains a challenging task in most real-world scenarios. We therefore rely on other complementary fitness functions (certainty level and the probability of functional faults) that we describe in the following sections to further guide the search toward functional faulty episodes.

2.4.4.2 Probability of Functional Fault

The second fitness function captures the probability of an episode to contain a functional fault. Such probability is predicted using an ML model. The fitness function is defined as follows:

$$f_2(e) = 1 - Pr_f[e] \tag{2.4}$$

where $Pr_f[e]$ is the probability of having a functional fault in an episode $e \in E$, and E is the space of all possible episodes. In the context of the running example, driving very close to obstacles has a higher probability of revealing a functional fault (high probability of collision) than an episode that maintains a safe distance from them. We therefore want the first episode to be favored by our search over the second one. The ML model that we use takes an episode as input, uses the presence and absence of abstract states in the episode as features, and returns the probability of functional fault for that episode. A detailed explanation of the probability prediction method using ML is provided in section 2.4.4.4. In the search process, instead of maximizing $Pr_f[e]$, we minimize its negation $f_2(e)$ to (1) have a consistent minimization problem across all fitness functions, and (2) guide the search toward finding episodes with a high probability of functional faults.

Example: In the context of our running example, it is possible to encounter a scenario where the speed of the vehicle is above 60 MPH (getting a positive reward at each time step) but excessively high, resulting in challenges to control the car, thus increasing the risk of collision. In such a case, using the reward fitness function is inadequate to guide the search toward such faulty episodes since they get high rewards. To address such situations, we can leverage the fitness function based on the probability of functional faults, which is predicted based on the presence of abstract states. With this approach, we can identify episodes that have a high probability of resulting in functional faults, even when the reward function cannot guide the search process effectively.

2.4.4.3 Certainty Level

This fitness function captures the level of certainty associated with the actions taken in each state within an episode. It is calculated as the average difference in each state-action pair between the probability of the chosen action, assigned according to the learned policy, and the second-highest probability assigned to an action [78]. This is also referred to as the action preference margin in the literature.

A higher accumulated certainty level across the sequence of actions in an episode suggests that the agent is more confident overall about the selected actions. On the other hand, a lower accumulated certainty level can guide our search toward situations in which the agent is highly uncertain of the selected action. Thus, it is relatively easier to lure the agent to take another action, which makes these episodes suitable for applying search operators.

The certainty level is calculated as shown in Equation 2.5, where e is the given episode, $|e|$ is its length, a_i is the selected action in state s_i (i.e., a_i is the action with the highest selection probability), A_i is the set of possible actions in state s_i, and $P_r(a_i|s_i)$ is the probability of selecting a_i in state s_i.

$$f_3(e) = \frac{\sum_{i=1}^{|e|} \left(P_r(a_i|s_i) - \max_{a_j \in A_i \ \& \ j \neq i} P_r(a_j|s_i)\right)}{|e|} \tag{2.5}$$

In our search algorithm, we aim to minimize this fitness function to guide the search toward finding episodes with high uncertainty levels.

Example: Suppose that in our running example, we have episodes where (1) the reward is high (i.e., driving with a speed above 60 MPH); (2) the probability of functional fault is low (i.e., the speed is not too high and the risk of collision is low); and (3) the uncertainty of the agent in selecting the optimal action is high. Such episodes are of particular interest, as they are promising candidates for our mutation operator (section 2.4.5.2). Indeed, by applying a small realistic transformation to these episodes, we can easily change the optimal action and explore new search directions that have the potential to reveal functional faults. This fitness function is especially useful when the reward function and probability of functional fault cannot guide the search process effectively, and we need to explore different actions and states to identify potential functional faults. Incorporating the certainty level metric into our fitness functions can thus help us to identify such episodes more effectively. In other words, by targeting episodes with high uncertainty levels, we can increase our chances of discovering new functional faults and improve the overall performance of our search process.

2.4.4.4 Machine Learning for Estimating Probabilities of Functional Faults

A machine learning algorithm is used to learn functional faults and estimate their probabilities in episodes without executing them. This model is expected to take episodes as input and predict the probabilities of functional faults. The labels of each episode are functional-faulty or not faulty. We choose *Random Forest* as a candidate modeling technique because (1) it can scale to numerous features, and (2) its robustness to overfitting has been well studied in the literature [22, 41]. We also tried several other ML models to predict functional faults, such as *K-Nearest Neighbor*, *Support Vector Machine*, and *Decision Trees*. However, *Random Forest* led to the most accurate prediction model. Since this is not a crucial or central aspect of the work, we do not include the results of these experiments in this chapter.

2.4.4.5 Preparation of Training Data

To build the above-mentioned machine learning model, the training data are collected from training episodes and random executions of the RL agent. More precisely, our ML training dataset contains both faulty and non-faulty episodes generated through the training and random executions of the agent.

Episodes from RL training. We sample episodes from the agent's training phase to increase the diversity of the dataset. We also include such episodes in case we do not find enough faulty episodes based on random executions. Providing data with different

types of episodes (i.e., functional-faulty and non-faulty) makes training the ML models possible. Since the training phase of the RL agents is exploratory, it contains a diverse set of faulty and non-faulty episodes, which helps learning and increases model accuracy. One issue with sampling from the training episodes is that they may not be consistent with the final policy of the trained agent. The agent may execute a faulty episode during training because of (1) randomness in action selection, due to the exploratory nature of the training process, and (2) incomplete agent training. To alleviate this issue, when sampling to form the training dataset of the ML model, we give a higher selection probability to the episodes executed in the later stages of the training, since they are more likely to be consistent with the final behavior of the trained agent.

Assuming a sequence of n episodes ($[E_i : 1 \leqslant i \leqslant n]$) that are explored during the training of the RL agent, the probability of selecting episode Ei ($P_r[E_i]$) is calculated as follows.

$$P_r[E_i] = \frac{i}{\Sigma_{j=1}^{n} j} \tag{2.6}$$

We thus give a higher selection probability to episodes executed in the later stage of the training phase of the agent ($P_r[E_1] < P_r[E_2] < ... < P_r[E_n]$).

Episodes from random executions. To build the training dataset of our ML model, we also include episodes generated through random executions of the agent to further diversify the training dataset with episodes that are consistent with the final policy of the agent. In practice, we use the episodes of the initial population of the generic search since they have been already created with random executions of the RL agent (section 2.4.3), thus minimizing the number of simulations (and therefore the testing budget).

2.4.4.6 State Abstraction for Training Data

After collecting the training episodes, we need to map each concrete state to its corresponding abstract state to reduce the state space and thus enable effective learning. Indeed, this is meant to facilitate the use of machine learning with more abstract features. To do so, we rely on the transitive Q^*-irrelevance abstraction method, which was described in section 2.2.2.

The state abstraction process is defined in Algorithm 2. The algorithm takes the concrete states as input and finds abstract states $s^{\phi_d} \in S_\phi$ considering the abstraction function of ϕ_d where d is the abstraction level. For each concrete state, we try to find the abstract state that corresponds to the concrete state by calculating the Q^*-values of all available actions, as described in section 2.2.2. If a match with an abstract state of a previous concrete state that was already processed is found, we assign the abstract state to the concrete state. Otherwise, we create a new abstract state.

Algorithm 2: High-level algorithm to create abstract states

Input: Set of states S, abstraction level d
Output: Abstract states S^{ϕ_d}

```
S^{φ_d} ← ∅
len ← 0
for s_i ∈ S do
    if S^{φ_d} = ∅ then
        len ← len + 1
        append s_i to S_1^{φ_d}
    Found = False
    for j in range(1, len) do
        if φ(s_i) = S_j^{φ_d} then
            append s_i to S_j^{φ_d}
            Found = True
    if Found = False then
        len ← len + 1
        append s_i to S_len^{φ_d}
Return S^{φ_d}
```

2.4.4.7 Feature Representation: Presence and Absence of Abstract States

To enable effective learning, each episode consists of state-action pairs, where the states are abstract states instead of concrete states. To train the ML model, we determine whether abstract states are present in episodes and use this information as features. As described in the following, each episode is encoded with a feature vector of binary values denoting the presence (1) or absence (0) of an abstract state S_i^{ϕ} in the episode and n is the total number of abstract states.

	S_1^{ϕ}	S_2^{ϕ}	$\cdots$	S_i^{ϕ}	$\cdots$	S_n^{ϕ}
$episode_i$	0	1	$\cdots$	0	$\cdots$	1

The main advantage of this representation is that it is amenable to the training of standard machine learning classification models. Furthermore, we were able to significantly reduce the feature space by grouping similar concrete states through state abstraction, where the selected action of the agent is the same for all concrete states within one abstract state. As a result, considering n different abstract states, the feature space of this representation is 2^n. Note that we only consider the abstract states that have been observed in the training dataset of the ML model, which we expect to be rather complete. Further, in this feature representation, the order of the abstract states in the episodes is not accounted for, which might be a weakness if we are not able to predict functional faults as a result.

Empirical results will tell whether the two above-mentioned potential problems materialize in practice.

2.4.4.8 Multi-Objective Search

We need to minimize the above-mentioned fitness functions to achieve our goal, and this is therefore a multi-objective search problem. More specifically, our multi-objective optimization problem can be formalized as follows:

$$\min_{x \in E} F(x) = (f_1(x), f_2(x), f_3(x)) \tag{2.7}$$

where E is the set of possible episodes in the search space, $F : E \rightarrow \mathbf{R}^3$ consists of three real-value objective functions $f_1(x), f_2(x), f_3(x)$, and $\mathbf{R}^3$ is the objective space of our optimization problem.

2.4.5 Search Operators

We describe below three genetic operators. The first operator is crossover, which generates new offspring using slicing and joining high-fitness, selected individuals. The second operator is mutation, which introduces small changes in individuals to add diversity to the population, thus making the search more exploratory. Finally, the selection operator determines which individuals survive to the next generation. We provide a detailed description of how we defined these operators.

2.4.5.1 Crossover

The crossover process is described in Algorithm 3. It uses the population as input and creates offspring as output. It begins by sampling an episode (line 2) with the *sample* function. This function draws an episode using tournament selection [91]. In a K-way tournament selection, K individuals are selected, and we run a tournament between the selected individuals where fitter individuals are more likely to be selected for reproduction. Then, we randomly select a crossover point (line 3) using the uniform distribution.

After finding the crossover point, we must find a matching pair (line 4). We do so by considering individuals in the population containing the abstract state of the pair selected as a crossover point. The *search* function tries to find a matching pair for the crossover point based on the Q^*-irrelevance abstraction method (section 2.2.2). If no matching pair is found (line 5), we repeat the process from the beginning (lines 1-5). Otherwise, offspring are created on lines 6-9. Whether a match can be found for the crossover point highly depends on the abstraction level. Therefore, this can be controlled by changing the abstraction level to prevent bottlenecks.

The crossover process is illustrated in Figure 2.2. Let us assume that the selected parent is as follows:

$$Parent = [(s_1, a_1), (s_2, a_2), ..., (s_{f-1}, a_{f-1}), \mathbf{(s_f, a_f)}, (s_{f+1}, a_{f+1}), ..., (s_m, a_m)] \tag{2.8}$$

where (s_f, a_f) is the pair selected as a crossover point.

The matching function tries to find an episode containing a pair that has a concrete state that belongs to the same abstract class as state (s_f) to ensure the validity of the new episode. Recall that all states in the same abstraction class are perceived to be the same by the RL agent. Also, since they have the same Q^*-values, their certainty level is the same.

$$Match = [(s'_1, a'_1), (s'_2, a'_2), ..., (s'_{v-1}, a'_{v-1}), \mathbf{(s'_v, a'_v)}, ..., (s'_n, a'_n)] \tag{2.9}$$

where s'_v and s_f result into the same abstract state based on our abstraction method (i.e., $\phi_d(s'_v) = \phi_d(s_f)$). As a result, the selected actions are also the same.

The newly created offspring are:

$$Offspring_1 = [(s_1, a_1), ..., (s_{f-1}, a_{f-1}), (s'_v, a'_v), ..., (s'_n, a'_n)] \tag{2.10}$$

$$Offspring_2 = [(s'_1, a'_1), ..., (s'_{v-1}, a'_{v-1}), (s_f, a_f), ..., (s_m, a_m)] \tag{2.11}$$

The first offspring contains the first part of the matching individual up to the crossover point with state s_{f-1} and the second part is taken from the parent and vice versa for the second offspring.

Based on the selected state abstraction method (section 2.2.2), we create episodes that are more likely to be valid, though this is not guaranteed. Also, we may get inconsistent episodes (i.e., episodes that cannot be executed by the RL agent). Furthermore, due to the high simulation cost of the RL environment, we are not executing episodes after applying crossover during the search. The validity of the episodes in the final archive is therefore checked by executing the final high-fitness episodes. The execution process is described in detail in section 2.4.6.

Algorithm 3: High-Level Crossover Algorithm

Input: Population P
Output: Offspring B_1 and B_2

do
  $parent \leftarrow sample(P)$
  $l \leftarrow CrossoverPoint(parent)$
  $match \leftarrow search(P, parent[l])$
while $match = \emptyset$
$B_1[0:l] \leftarrow match[0:l]$
$B_1[l:end] \leftarrow parent[l:end]$
$B_2[0:l] \leftarrow parent[0:l]$
$B_2[l:end] \leftarrow match[l:end]$
return B_1, B_2

2.4.5.2 Mutation

The mutation operator starts by selecting an episode using a K-way tournament selection. Then a mutation point is randomly selected using the uniform distribution. To ensure the exploratory aspect of the mutation operator, we alter the state of the mutation point using some image transformation methods that are selected by considering the environment and the learning task to produce realistic and feasible states [109, 136]. These transformations are context-dependent. They should be realistic and representative of situations with imprecise sensors or actuators, as well as external factors that are not observable. In our running example, transformations matching such situations include changing the brightness and contrast of the image, adding tiny black rectangles to simulate dust on the camera lens, or changing the weather. Another example is the *Cart-Pole* problem (section 2.5.2), where the task is to balance a pole on a moving cart. For this type of environment, we rely on other transformations, such as slightly changing the position, the velocity, and the angle of the pole.

After mutating the gene, we run the episode. Although executing episodes is computationally costly, mutation is infrequent and helps create valid and consistent episodes exploring unseen parts of the search space. Then, the updated episode is added to the population. Also, if we find any failure during the execution of the mutated episodes, we mark such episodes as failing and exhibiting a functional fault.

Assume that (1) the selected episode for mutation is e_h, (2) we select (s_c, a_c) from e_h as a candidate pair, and (3) the mutated/transformed state for s_c is s_c^t. The mutated episode e_h^m is then as follows:

$$e_h = [(s_1, a_1), ..., \boldsymbol{(s_c, a_c)}, (s_{c+1}, a_{c+1}), ..., (s_m, a_m)] \tag{2.12}$$

$$e_h^m = [(s_1, a_1), ..., \boldsymbol{(s_c^t, a_c^t)}, ...] \tag{2.13}$$

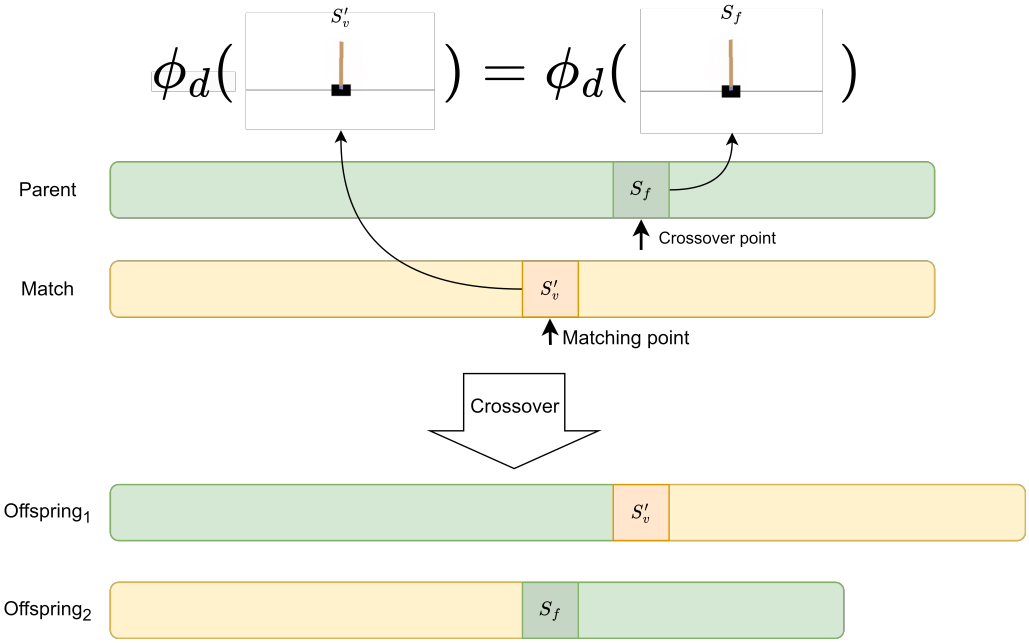

Figure 2.2: The crossover operator

where the states after the mutation point are determined from executing episode e_h^m.

2.4.5.3 Selection

Given that our search involves optimizing multiple fitness functions and the possibility of incorporating additional fitness functions to address different types of faults, it is imperative to employ a search algorithm that can handle more than three fitness functions. Additionally, it is worth noting that our focus is not on Pareto front optimization or the generation of a well-distributed set of solutions that capture the trade-off between objectives. Rather, we need a search algorithm that can optimize all the fitness functions concurrently and separately.

Considering these search requirements, we do not rely on traditional dominance-based algorithms like NSGA-II, particle swarm optimization (PSO) [142], or SPEA-II [9] because they show poor performance in problems with many objectives [130] and they target Pareto front optimization. Additionally, to build a generalizable and extensible testing approach, we opt for the Many Objective Sorting Algorithm (MOSA) [106] to select the best individuals that minimize our fitness functions. This is because we anticipate that, in many cases, we will need to consider more than three fitness functions, and MOSA is specifically tailored to our application context, software test automation. MOSA can indeed accommodate a large number of objectives and is better at generating diverse solutions to address all of the fitness functions. MOSA is therefore an optimal choice for our problem domain because no other search algorithm fulfills our requirements.

MOSA is a dominance-based multi-objective search algorithm based on NSGA-II [31]. Although traditional dominance-based algorithms like NSGA-II and SPEA-II [9] show poor performance in problems with many objectives [130], MOSA performs well even for a large number of objectives. It is widely used in the literature and tries to generate solutions that cover the fitness functions separately, instead of finding a well-distributed set of solutions (i.e., diverse trade-offs between fitness functions). More specifically, in our context, MOSA's purpose is to generate faulty episodes that separately satisfy at least one of the search objectives rather than to find episodes that capture diverse trade-offs among them. It drives the search toward the yet-uncovered search objectives and stores in an archive all solutions that satisfy at least one search objective.

MOSA works as follows. Similar to NSGA-II, it starts from an initial population and generates new offspring at each generation using genetic operators (i.e., mutation and crossover). We calculate the fitness value of each individual in the population based on the three fitness functions that we described in section 2.4.4. MOSA then uses a novel preference method to rank the non-dominant solutions. In this preference method, the best solutions according to each fitness function are rewarded with the $rank = 0$, and the other solutions are ranked based on the traditional non-dominated sorting in NSGA-II. During the transition to a new population, we select the highest-ranked individuals using MOSA and add them to the new population without any changes. We also transfer a subset of the individuals with the highest fitness from the previous population to avoid losing the best solutions. Finally, an archive is used to store the best individuals for each individual fitness function.

2.4.6 Execution of Final Results

After completing the execution of the genetic algorithm, we obtain a population that contains episodes with high fault probability. We need to execute these final episodes to check their validity, their consistency with the policy of the agent, and whether they actually trigger failures. We assume that an episode is consistent if the RL agent can execute it. We retain failing episodes that are both valid and consistent.

During the execution process, we may observe deviations where the agent selects an action other than the action in the episode. To deal with such deviations, we replace the state observed by the agent with the corresponding state from the episode and observe the selected action. For example, let us assume that we want to execute an episode e' produced by STARLA where $e' = [(s'_i, a'_i)|s'_i \in S, a'_i \in A, 0 \leq i \leq n, n \in \mathbb{N}]$. We set the state of the simulator to the initial state of the episodes s'_0. Then we use the states from the environment as input to the agent and we check the action selected by the agent. If during the execution, the agent selects an unexpected action $a_i \neq a'_i$ at state s_i, we replace s_i with state s'_i from episode e' to drive the agent to select action a'_i.

If the action selected by the agent is not a'_i, we consider that episode e' is invalid and we remove it from the final results.

Replacing states in this situation is acceptable since (1) we assume that the environment is stochastic, (2) states in episode e' are real concrete states generated in the environment,

and (3) we noticed that the states of the environment and in the episodes where deviations occur are very similar. The latter is likely due to the selection of the crossover point based on identical abstract states. Indeed, we observed that 94% of the environment and episode states where deviations occur in the *Cart-Pole* environment, which we will describe in detail in section 2.5.2.1, have a cosine distance lower than 0.25. Similarly, we observed that 99% of the deviations in the *Mountain Car* environment (see section 2.5.2.2 for more details) have a cosine distance lower than 0.25. Replacing similar states where deviations of the agent occur is therefore a sensible way to execute such episodes (if possible) because, in real-world environments, we may have incomplete or noisy observations due to imperfect sensors.

2.5 Empirical Evaluation

This section describes the empirical evaluation of our approach, including the research questions, the case study, the experiments, and the results.

2.5.1 Research Questions

Our empirical evaluation is designed to answer the following research questions.

2.5.1.1 RQ1. Do we find more faults than Random Testing with the same testing budget?

We aim to study the effectiveness of our testing approach in terms of the number of detected faults compared to Random Testing. We want to compare the two approaches with the same testing budget, which is defined as the number of executed episodes during the testing phase. Given that the cost of real-world RL simulations can be high (e.g., autonomous driving systems), this is the main cost factor.

2.5.1.2 RQ2. Can we rely on ML models to predict faulty episodes?

In this research question, we want to investigate whether it is possible to predict faulty episodes using an ML classifier. We do not execute all episodes during the search; therefore, we want to use the probabilities of functional faults that are estimated by an ML classifier as a fitness function to guide our search toward finding faulty episodes.

2.5.1.3 RQ3. Can we learn accurate rules to characterize the faulty episodes of RL agents?

One of the goals of testing an RL agent is to understand the conditions under which the agent fails. This can help developers assess the risks of deploying the RL agent and focus

its retraining. Therefore, we aim to investigate the learning of interpretable rules that characterize faulty episodes from the final episodes that are executed once the search is complete.

2.5.2 Case Studies

In our study, we consider two Deep-Q-Learning (DQN) agents on the *Cart-Pole*[2] balancing problem and *Mountain Car* [3], both from the *OpenAI Gym* environment.

We have chosen these RL case studies because they are open source and widely used as benchmark problems in the RL literature [4, 99, 107]. We have also considered these benchmarks as they include a large number of concrete states. Furthermore, the simulations in such environments are fast enough to enable large-scale experimentation.

2.5.2.1 Cart-Pole Balancing Problem

In the *Cart-Pole* balancing problem, a pole is attached to a cart, which moves along a track. The movement of the cart is bidirectional and restricted to a horizontal axis with a defined range. The goal is to balance the pole by moving the cart left or right and changing its velocity.

As depicted in Figure 2.3, the state of the agent is characterized by four variables:

- The position of the cart.
- The velocity of the cart.
- The angle of the pole.
- The angular velocity of the pole.

We provide a reward of +1 for each time step when the pole is still upright. The episodes end in three cases: (1) the cart is away from the center with a distance more than 2.4 units, (2) the pole's angle is more than 12 degrees from vertical, or (3) the pole remains upright during 200 time steps. We define functional faults in the *Cart-Pole* balancing problem as follows.

If, in a given episode, the cart moves away from the center by a distance above 2.4 units, regardless of the accumulated reward, we consider that there is a functional fault in that episode. Note that termination based on the pole's angle is an expected behavior of the agent and thus a normal execution, whereas termination based on passing the borders of the track can cause damage and is therefore considered a safety violation.

[2] gymlibrary.dev/environments/classic_control/cart_pole/
[3] gymlibrary.dev/environments/classic_control/mountain_car/

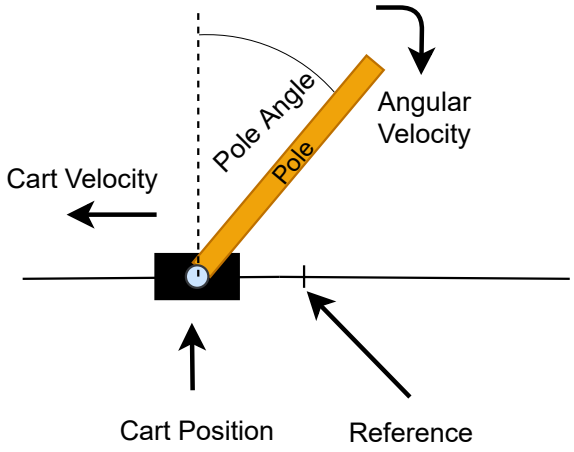

Figure 2.3: *Cart-Pole* balancing problem

2.5.2.2 Mountain Car Problem

In the *Mountain Car* problem, an under-powered car is located in a valley between two hills. The objective is to control the car and strategically use its momentum to reach the goal state on top of the right hill as soon as possible. The agent is penalized by -1 for each time step until termination. As illustrated in Figure 2.4, the state of the agent is defined based on (1) the location of the car along the x-axis, and (2) the velocity of the car.

There are three discrete actions that can be used to control the car:

- Accelerate to the left.
- Accelerate to the right.
- Do not accelerate.

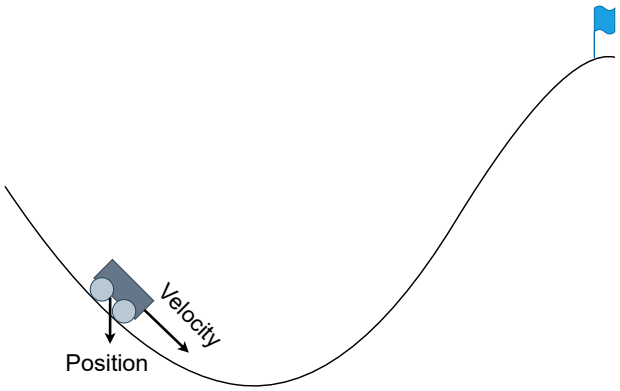

Figure 2.4: *Mountain Car* problem

Episodes end in three cases: (1) reaching the goal state, (2) crossing the left border, or (3) exceeding the limit of 200 time steps. In our custom version of the *Mountain Car*

(Figure 2.4), climbing the left hill is considered an unsafe situation. Consequently, reaching the leftmost position in the environment results in a termination with the lowest reward. Therefore, if in a given episode, the car crosses the left border of the environment, we consider that there is a functional fault in that episode.

2.5.3 Implementation

We used Google Colab and stable baselines [56] to implement RL agents for both case studies (see section 2.5.2). Our RL agents are based on a DQN policy network [94] using standard setting of stable baselines (i.e., Double Q-learning [139], and dueling DQN [144]). Our *Cart-Pole* RL agent has been trained for 50 000 time steps. The average reward of the trained agent is 124 (which is also equal to the average length of the episodes). In general, the pole is upright over 124 time steps out of a maximum of 200. The *Mountain Car* agent has been trained for 90 000 time steps. The average reward is -125 and the average length of the episodes is 112.

Finally, we execute the search algorithm for a maximum of 10 generations for both case studies. The mean execution time of STARLA on Google Colab was 89 minutes for the *Cart-Pole* problem and 65 minutes for the *Mountain Car* problem.

2.5.4 Evaluation and Results

2.5.4.1 RQ1. Do we find more faults than Random Testing with the same testing budget?

In this research question, we want to study STARLA's effectiveness in finding more faults than Random Testing when we consider the same testing budget B, measured as the number of executed episodes. To do so, we consider two practical testing scenarios:

- **Randomly executed episodes are available or inexpensive**: In the first scenario, we assume that we want to further test a DRL agent provided by a third-party organization. We assume that both training episodes and some randomly executed episodes of the RL agent, used for testing the agent, are provided by the third party. Therefore, we can extract ML training data and an initial population from such episodes without using our testing budget.

  We can also consider another situation where the RL agent is trained and tested using both a simulator and hardware in the loop [84]. Such two-stage learning of RL agents has been widely studied in the literature, where an agent is trained and tested on a simulator to "*warm-start*" the learning on real hardware [68, 84]. Since STARLA produces episodes with a high fault probability, we can use it to test the agent when executed on real hardware to further assess the reliability of the agent. In this situation, STARLA uses prior episodes that have been generated on the simulator to build the initial population and executes the newly generated episodes on the hardware.

In this case, randomly executed episodes using a simulator become relatively inexpensive. Therefore, only episodes that are executed with hardware in the loop and in the real environment are accounted for in the testing budget.

To summarize, when randomly executed episodes are available or inexpensive, the testing budget B is equal to the sum of (1) the number of mutated episodes that have been executed during the search, and (2) the number of faulty episodes generated by STARLA that have been executed after the search.

- **Randomly executed episodes are generated with STARLA and should be accounted for in the testing budget:** In the second scenario, we assume that the agent is trained and then tested by the same organization using STARLA. Therefore, we have access to the training dataset but need to use part of our testing budget, using random executions, to generate the initial population. More precisely, the total testing budget in this scenario is equal to the sum of (1) the number of episodes in the initial population that have been generated through random executions of the agent; (2) the number of mutated episodes that have been executed during the search; and (3) the number of faulty episodes generated by STARLA that have been executed after the search.

Because of randomness in our search approach and its significant execution time (section 2.5.3), for both case studies, we re-executed the search algorithm 20 times and stored the generated episodes and the executed episodes with mutations at each run. We computed the mean number of generated functional-faulty episodes N (in *Cart-Pole* N_c=5313 and in *Mountain Car* N_m = 2809) and the mean number of mutated executed episodes M (in *Cart-Pole* case studyM_c=128 and in *Mountain Car* M_m = 139) over the 20 runs.

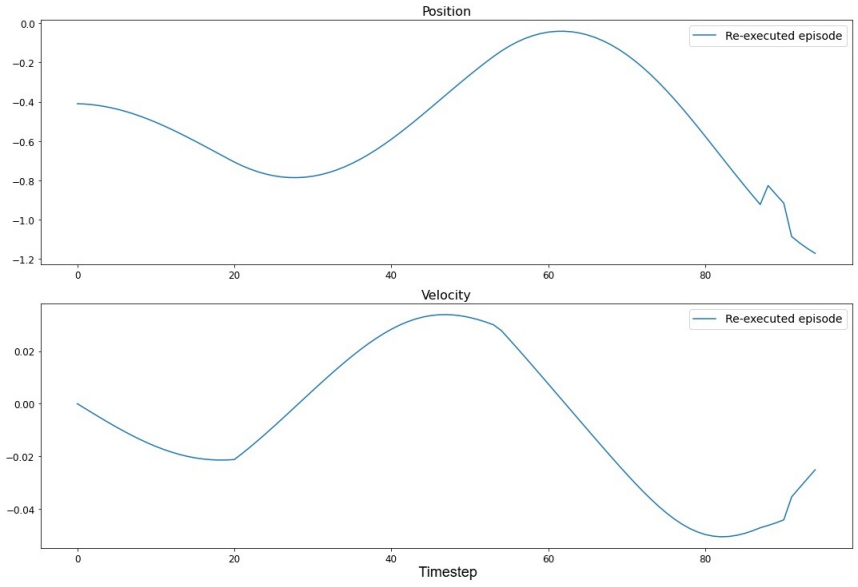

Figure 2.5: An example of re-executed episode

Figure 2.5 illustrates the position and velocity during an episode generated and re-executed by STARLA in the Mountain Car environment. The episode concludes with a

functional fault as the car's position exceeds the limit of -1.2, potentially leading to harm or damage.

We analyzed the distribution of the total number of functional faults identified with STARLA over the 20 runs. Then, we randomly selected (with replacement) 100 samples from the set of episodes that were generated with Random Testing. Each sample contained B episodes to ensure that we had the same testing budget as in STARLA.

In the *Cart-Pole* case study, for the first scenario, B is equal to 5441 (which corresponds to the mean number of generated faulty episodes and executed mutated episodes with STARLA over the 20 runs). On the other hand, for the second scenario, B is equal to 6941 because, as previously explained, this testing budget accounts for the number of episodes in the initial population, which were generated with Random Testing (1500). Also, in the *Mountain Car* case study, the mean number of generated faulty episodes and executed mutated episodes over the 20 runs is equal to 2948 (B = 2948 in scenario 1). For the second scenario, B is equal to 4448.

We analyzed the distribution of the identified faults in the two testing scenarios, compared it with STARLA, and reported the results. The results of the *Cart-Pole* and *Mountain Car* case studies are depicted in Figure 2.6 and Figure 2.7, respectively. We should note that in the first scenario, we only compute faults that are generated with the genetic search. We do not consider faults that are in the initial population because we assume that these episodes are provided to STARLA and are not included in the testing budget. In contrast, in the second scenario, we include in STARLA's final results the faulty episodes in the initial population as they are part of our testing approach and are included in the testing budget.

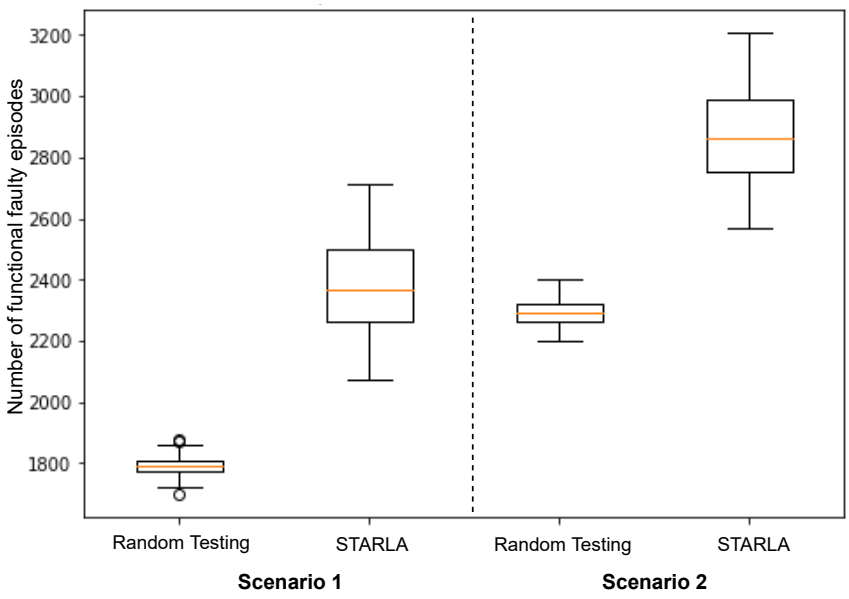

Figure 2.6: Number of functional-faulty episodes generated with STARLA compared to Random Testing in the *Cart-Pole* problem

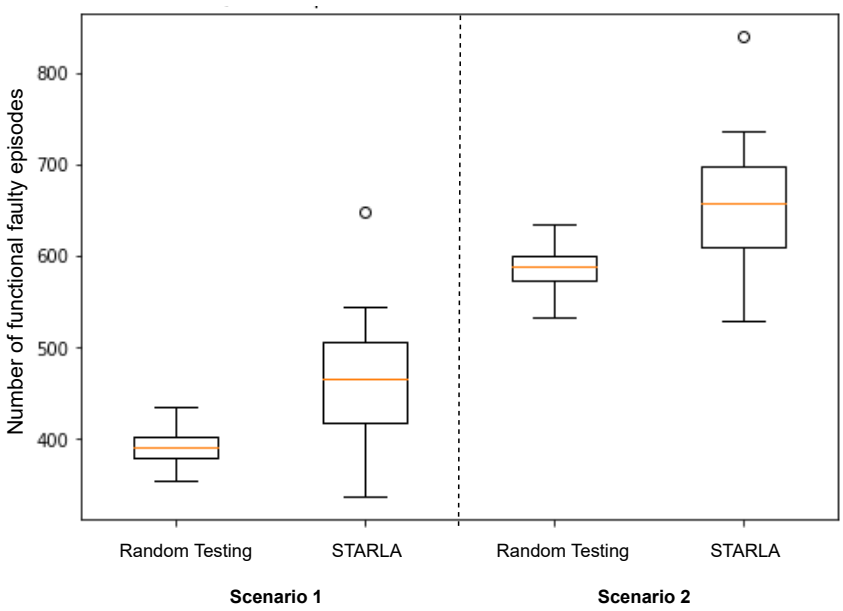

Figure 2.7: Number of functional-faulty episodes generated with STARLA compared to Random Testing in the *Mountain Car* problem

As we can see from the boxplots, our approach outperforms Random Testing in detecting faults in both scenarios. Indeed, in the first scenario, the average number of faulty episodes detected by STARLA is 2367, while an average of 1789 faulty episodes is detected with Random Testing in the *Cart-Pole* case study. In the *Mountain Car* case study, the average number of faulty episodes detected by STARLA is 466 while an average of 390 faulty episodes is detected with Random Testing (+19.5%).

In the second scenario, where we consider a bigger testing budget, STARLA also outperforms Random Testing by identifying, on average, 2861 faulty episodes compared to 2291 identified by Random Testing (+24.9%) in the *Cart-Pole* case study, and 657 faulty episodes compared to 591 identified by Random Testing. To assess the statistical significance of the average difference of the number of detected functional faults between STARLA and Random Testing, we use the non-parametric Mann-Whitney U-test [86] and compute the corresponding p-value in both testing scenarios and case studies.

In both scenarios for the two case studies, the p-values are far below 0.01, and thus show that our approach significantly outperforms Random Testing in detecting functional faults in DRL-based systems.

Answer to RQ1: We find significantly more functional faults with STARLA than with Random Testing using the same testing budget.

2.5.4.2 RQ2. Can we rely on ML models to predict faulty episodes?

We investigate the accuracy of the ML classifier to predict faulty episodes of the RL agent. We use *Random Forest* to predict the probability of functional faults in a given episode. To build our training dataset in the *Cart-Pole* case study, we sampled 2111 episodes, including 1500 episodes generated through random executions of the agent and 611 episodes from the agent's training phase (as described in section 2.4.3). Of these episodes, 733 correspond to functional faults while 1378 are non-functional faulty. Further, the ML training dataset of the *Mountain Car* case study is created by sampling 2260 episodes where (1) 1862 episodes are generated through random executions of the agent, and (2) 398 episodes are from the training phase of the *Mountain Car* agent. Of these episodes, 456 are functional-faulty while 1804 are not.

d	**Abstract states**	**Accuracy**	**Precision**	**Recall**
0.005	195073	63%	39%	63%
0.01	146840	63%	39%	63%
0.05	33574	73%	81%	73%
0.1	15206	92%	92%	92%
0.5	2269	95%	95%	95%
1	1035	97%	97%	97%
5	134	84%	84%	84%
10	48	79%	81%	79%
50	8	77%	78%	77%
100	4	77%	78%	77%

Table 2.1: Prediction of *functional faults* with *Random Forest* in the *Cart-Pole* case study. The first column represents the abstraction level d, the second column shows the number of abstract states for each abstraction level, and we report the accuracy, precision, and recall in the next columns.

In both case studies, we trained *Random Forest* models with standard hyperparameters using the previously described datasets to predict functional faults. Because of the high number of concrete states in our dataset (about 250 000 in *Cart-Pole* and 270 000 in *Mountain Car*), we need to reduce the state space by using state abstraction to facilitate the learning process of the *Random Forest* models. As presented in section 2.4.4.6, we used the Q^*-irrelevance state abstraction technique [1] to reduce the state space. We experimented with several values for the abstraction level d (section 2.2.2) and reported the prediction results in terms of precision, recall, and accuracy in Tables 2.1 and 2.2. We obtained fewer abstract states when we increased d because more concrete states were included in the same abstract states. For each value of d, we considered a new training dataset (with different abstract states). For each dataset, we trained *Random Forest* by randomly sampling 70% of the data for training and used the remaining 30% for testing.

The overall prediction results for functional faults are promising. As shown in Table 2.1, the best results for the prediction of functional faults yield a precision and a recall of 97%

d	Abstract states	Accuracy	Precision	Recall
0.005	256826	79%	84%	79%
0.01	246624	79%	84%	79%
0.05	195864	84%	87%	84%
0.1	160142	88%	90%	88%
0.5	83887	99%	99%	99%
1	59169	99%	99%	99%
5	19019	99%	99%	99%
10	10108	99%	99%	99%
50	2012	99%	99%	99%
100	890	99%	99%	99%
500	93	99%	99%	99%
1000	38	99%	98%	99%
5000	10	98%	98%	98%
10000	9	93%	94%	93%

Table 2.2: Prediction of *functional faults* with *Random Forest* in the *Mountain Car* case study. The first column represents the abstraction level d, the second column shows the number of abstract states for each abstraction level, and we report the accuracy, precision, and recall in the next columns.

in the *Cart-Pole* problem. Similarly, a precision and a recall of 99% was achieved in the *Mountain Car* problem as depicted in Table 2.2.

Also, for both case studies, we observe that when we increase the state abstraction level, the accuracy of the ML classifiers improves until it plateaus and then starts to decrease as information that is essential for classification is lost. This highlights the importance of finding a proper state abstraction level to (1) facilitate the learning process of the ML classifiers, and (2) more accurately predict functional faults. Note that we consider the abstraction level d that maximizes the accuracy of the ML model while significantly decreasing the total number of distinct abstract states in the dataset. In the *Cart-Pole* and *Mountain Car* case studies, d is equal to 1 and 500, respectively. Differences in the abstraction levels are due to the differences in the complexity of the environments and the state representations.

Answer to RQ2: By using an ML classifier (based on *Random Forest*) combined with state abstraction, we can accurately classify whether episodes are functional-faulty or not. Such a classifier can therefore be used as fitness functions in our search. However, finding a suitable level of state abstraction is essential to increase the learnability of the ML classifier and thus to improve the accuracy of the fault prediction results.

2.5.4.3 RQ3. Can we learn accurate rules to characterize the faulty episodes of RL agents?

We investigate the learning of interpretable rules that characterize faulty episodes to understand the conditions under which the RL agent can be expected to fail. Consequently, we rely on interpretable ML models, namely *Decision Trees*, to learn such rules. We assess the accuracy of decision trees and therefore our ability to learn accurate rules based on the faulty episodes that we identify with our testing approach. In practice, engineers will need to use such an approach to assess the safety of using an RL agent and targeting its retraining.

In our analysis, we consider a balanced dataset that contains (1) faulty episodes created with STARLA, and (2) non-faulty episodes obtained through random executions of the RL agent. We consider the same proportions of faulty and non-faulty episodes. Such a dataset would be readily available in practice to train decision trees. For training, we use the same type of features as for the ML model that was used to calculate one of our fitness functions (section 2.4.4.7). Each episode is encoded with a feature vector of binary values denoting the presence (1) or absence (0) of an abstract state in the episode. We rely on such features because the ML model that we have used to predict functional faults showed good performance using such representation. Moreover, we did not rely on the characteristics of concrete states to train the model and extract the rules due to (1) the potential complexity of state characteristics in real-world RL environments, and (2) Q^*-values matching abstract states are more informative since they also capture the next states of the agent and the optimal action (i.e., the agent's perception).

Since we simply want to explain the faults that we detect by extracting accurate rules, we measure the accuracy of the models using K-fold cross-validation. We repeat the same procedure for all executions of STARLA in each case study to obtain a distribution of the accuracy of the decision trees. More specifically, we study the distributions of precision, recall, and F1-scores for the detected faults and report the results in Figures 2.8 and 2.9.

As shown in Figures 2.8 and 2.9, we learned highly accurate decision trees and therefore rules (tree paths) that characterize faults in RL agents. Indeed, rules predicting faults in the *Cart-Pole* case study have a median precision of 98.75%, a recall of 98.90%, and an F1-score of 98.85%.

Furthermore, rules predicting functional faults in the *Mountain Car* case study have a median precision of 96.25%, a recall of 93%, and an F1-score of 94.75%. The rules that we extract consist of conjunctions of features capturing the presence or absence of abstract states in an episode.

We provide in the following an example of a rule that we obtained in the *Cart-Pole* problem:

$$\textbf{R1: } not(S_5^{\phi}) \textit{ and } S_{12}^{\phi} \textit{ and } S_{23}^{\phi} \tag{2.14}$$

where rule *R1* states that an episode is faulty if there are no concrete states in the episode

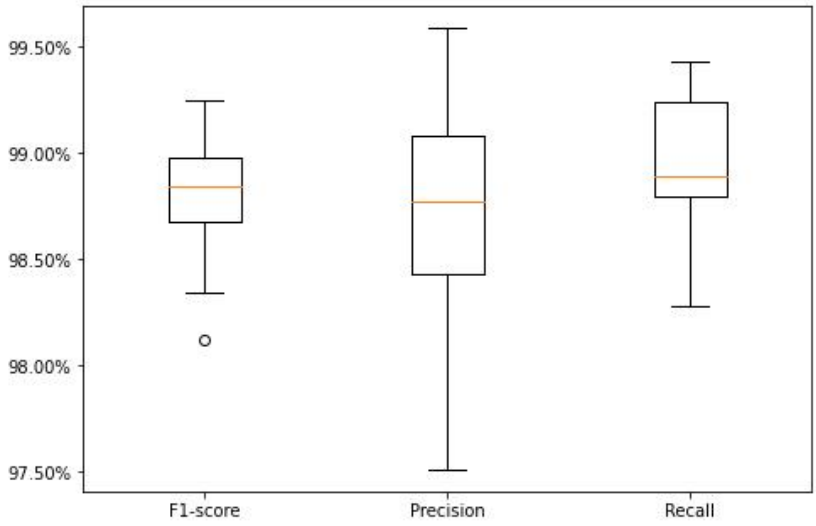

Figure 2.8: Accuracy of rules predicting faults in *Cart-Pole*

that belong to abstract state S_5^ϕ and we have at least two concrete states matching abstract state S_{12}^ϕ and S_{23}^ϕ, respectively.

From a practical standpoint, such highly accurate rules can help developers understand, with high confidence, the conditions under which the agent fails. For example, one can analyze the concrete states that correspond to abstract states that lead to faults to extract real-world conditions of failure. For example, to interpret the rule *R1*, we first extract all faulty episodes following this rule. Then, we extract from these episodes all concrete states belonging to the abstract states that must be present according to *R1* (i.e., S_{12}^ϕ and S_{23}^ϕ, respectively). For abstract states that the rule specifies as absent (abstract state S_5^ϕ in our example), we extract the set of all corresponding concrete states from all episodes in the final dataset. Finally, for each abstract state in the rule, we analyze the distribution of each characteristic of the corresponding concrete states (e.g., the position of the cart in the *Cart-Pole* environment, the velocity, the angle of the pole, and the angular velocity) to interpret the situations under which the agent fails. Due to space limitations, we include the box plots of the distributions of the states' characteristics in our replication package.

Note that we did not rely directly on the abstract states' Q^*-values to understand the failing conditions of the agent since they are not easily interpretable. We rely on the median values of the distribution of the states' characteristics to illustrate each abstract state and hence the failing conditions. We illustrate these conditions in Figure 2.10.

Our analysis shows that the presence of abstract states S_{12}^ϕ and S_{23}^ϕ represent situations where the cart is close to the right border of the track and the pole strongly leans toward the right. To compensate for the large angle of the pole, shown in the figure, the agent has no choice but to push the cart to the right, which results in a fault because the border is crossed. Moreover, abstract state S_5^ϕ represents a situation where (1) the angle of the pole is not large, and (2) the position of the cart is toward the right but not close to the border.

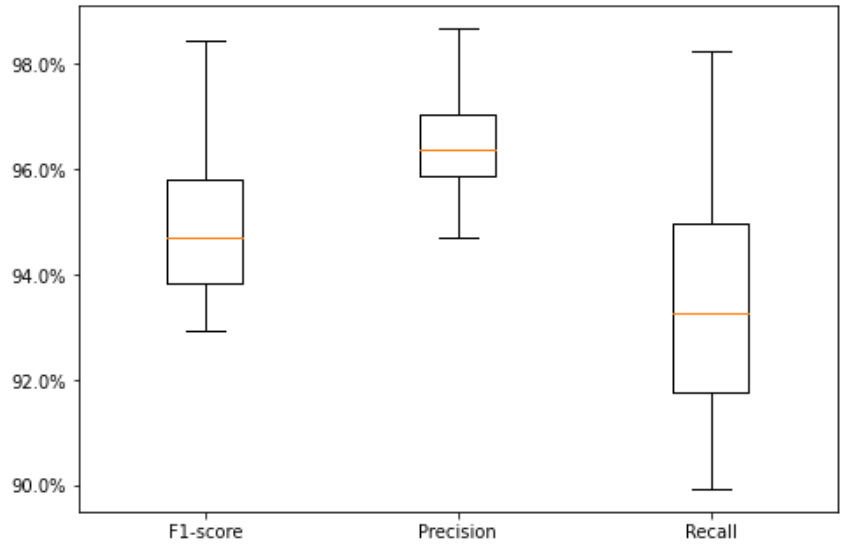

Figure 2.9: Accuracy of rules predicting faults in *Mountain Car*

In this situation, the agent will be able to control the pole in the remaining area and keep the pole upright without crossing the border, which justifies why this abstract state is absent in faulty episodes that satisfy rule *R1*. Note that we only provide an example of a faulty rule from the *Cart-Pole* case study. Different rules that consist of more complex combinations of different abstract states can be extracted and therefore analyzed. Such interpretable rules can thus assist engineers to ensure safety and analyze risks prior to deploying the agent.

We acknowledge that the extracted rules from the detected faulty episodes are not sufficient to evaluate the risk of deploying RL agents. However, characterizing the DRL agent's faulty episodes, as we automatically do, is indeed a necessary piece of information for risk analysis. If they are accurate, such rules can be used to understand the conditions under which the agent is likely to fail.

Answer to RQ3: By using our search-based technique and interpretable ML models, such as *Decision Trees*, we can accurately learn interpretable rules that characterize the faulty episodes of RL agents. Such rules can then serve as the basis for risk analysis before deployment of the agent to avoid safety violations.

2.6 Discussions

In this research we propose STARLA, a search-based approach to detect faulty episodes of an RL agent. To the best of our knowledge, this is the first testing approach focused on testing the agent's policy and detecting what we call functional faults.

Simulation cost. We rely on a small proportion of the training data of an RL agent and do

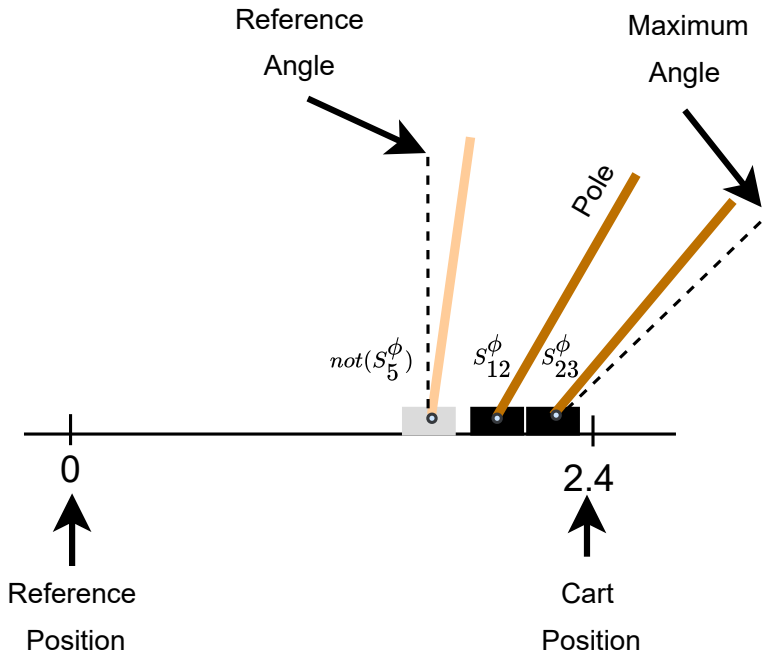

Figure 2.10: Interpretation of Rule *R1*. Each cart represents one abstract state. The gray cart depicts the state of the system in abstract state S_5^ϕ, which should be absent in the episode. The black carts represent the presence of abstract states S_{12}^ϕ and S_{23}^ϕ, respectively. Having both latter states appearing in an episode and not having the state on the left is highly likely to lead to a fault.

not need to access the internals of the RL-based system, hence the data-box nature of our solution. Our testing approach outperforms Random Testing of RL agents because we were able to find significantly more functional faults with the same simulation budget. However, Random Testing might outperform our testing approach in simple environments with fast simulations since it typically generates a much higher number of episodes, including faulty episodes. Nonetheless, RL agents are generally used in complex environments (e.g., cyber-physical systems), where the simulation and therefore test execution costs are very high. Narrowing the search toward the faulty space of the agent is therefore crucial to optimize RL testing in a scalable way.

Feature representation. Relying on state abstraction helped us to reduce the search space and increase the learnability of the ML models that we used to (1) calculate the probabilities of functional faults, and (2) extract and interpret the rules characterizing faulty episodes. However, depending on the type of the RL task, in practice, one needs to select the right state abstraction type and level to effectively guide the search toward finding faults in the RL agent. State abstraction has allowed us to extract accurate rules predicting the presence of faults and thus enables effective risk analysis. Although we investigated different feature representations, such as encoding episodes with sequences of abstract states, the accuracy of the ML model was only slightly improved. Therefore, it was considered not worth the additional complexity to account for such sequential information.

State abstraction. As described in section 2.4.5.1, we rely on state abstraction to find matching states for crossover points. Based on the applied abstraction method, the matched abstract states may not capture the exact physical situations. However, they correspond to concrete states leading to similar expected rewards for the same selected actions. State abstraction is one of the key heuristics to enable an effective search in our context. If we find two similar states where the agent has similar Q^*-values, we consider that those states are similar enough to perform the crossover. Like any heuristic, it needs to be evaluated. Therefore, to ensure the validity of the generated episodes, we validated the final created episodes by re-executing them as explained in section 2.4.6. We also re-executed these episodes to check their consistency with the policy of the agent and whether they actually trigger failures. To further investigate the physical similarity of concrete states from the same abstract states, we analyzed the distribution of concrete state parameters (e.g., position, velocity, pole angle) in abstract states. We checked the abstract states in the rules characterizing the faulty episodes and have included examples of these distributions in our replication package. Our analysis revealed low variance in the distributions of parameters, indicating that concrete states within the same abstract state exhibit similar physical characteristics and Q^*-values. Our validation also shows that the states of the environment and in the episodes where deviations occur are not frequent and have very low cosine distances. Furthermore, the results confirmed that such abstraction still leads to good search guidance since our approach outperforms Random Testing by finding significantly more functional faults.

Functional faults. We should note that the number of functional faults in our case studies is relatively high since the reward function of the RL agents does not help prevent this type of fault (despite the high average reward of the agent). As mentioned in section 2.5.2, we have relied on standard reward functions. For example, in the *Cart-Pole* environment, the agent's reward does not increase when it crosses the borders (i.e, where there is a functional fault and the episode terminates). We are using a standard, widely used, artificial benchmark to validate our testing approach, but we expect the number of functional faults in real-world RL agents to be much lower. Indeed, in more complex environments, a high penalty for functional faults could be part of the reward function to minimize them and prevent safety violations. However, as we explained in section 2.3.1, this is not always enough to prevent safety-critical situations because the agent's reward could still be relatively acceptable in the presence of functional faults. For example, a car may successfully reach its destination sooner by driving at a higher speed while being dangerously close to other cars. In this case, we have a high reward, because the car successfully reached the destination sooner without having any accident (the reward is defined based on the time of arrival and the penalty is applied when a collision actually occurs). However, such an episode has a high probability of functional fault since the ego car remains very close to the other cars. Relying only on the agent's reward thus makes it challenging for the search to identify functional faults. In other words, the reward and probability of functional faults are not always related. They are instead complementary and both used by STARLA to guide the search toward functional-faulty episodes.

Furthermore, we can have multiple types of functional faults related to an RL agent. In that case, we can consider the probability of each type of fault as a separate fitness

function in the genetic algorithm. More specifically, we can rely on the ML model to predict the probability of each type of fault and use them as separate fitness functions. Note that considering more fitness functions to predict multiple types of functional faults is rather straightforward thanks to the use of MOSA, which is specifically designed for many-objective search problems (section 2.4.5.3).

Initial population. We also investigated different sizes of the initial population (i.e., 500, 1050, and 3000) and obtained consistent results: STARLA outperformed Random Testing in terms of the total number of detected functional faults. Furthermore, we observed that the number of detected faults increased with the size of the initial population of the search. For example, in the *Cart-Pole* case study, the accuracy of the faults prediction decreased for training data sets with less than 670 episodes. In some practical cases, costly simulations may lead to limited data for testing RL agents with STARLA. Consequently, this can affect the accuracy of ML models due to small ML training datasets and the results of the genetic search (i.e., due to the small initial population). From a practical standpoint, the size of the initial population is bounded by a predetermined testing budget and can be determined according to the accuracy of the ML model. Depending on the case study, we may choose the size of the initial population that maximizes the accuracy of the ML model while consuming a reasonable portion of the testing budget.

Improving the diversity in initial populations for genetic algorithms may potentially increase the quality of the search results by enabling a faster convergence to optimal solutions. As mentioned in section 2.4.3, we considered episodes starting from different initial states to diversify the initial population. In future work, we aim to investigate the use of state-of-the-art diversity metrics to guide the generation of a diverse initial population to determine whether the additional diversity computations are beneficial given that they use part of the test budget. Examples of such metrics include geometric diversity [71], entropy measure [6], and hamming distance [97].

Rules characterizing faulty episodes. In this research, we investigate the learning of interpretable rules that characterize faulty episodes to understand the conditions under which the RL agent can be expected to fail. If accurate, these rules can help developers to focus their analysis on specific abstract states that lead to faults, and analyze the risks related to the deployment of the RL agent. For example, after analyzing the failing rules of the agent, engineers can use abstract states leading to faults to automatically ensure safety at run-time. The agent state can be monitored to assess the risks and activate corrective measures. To prevent a failure, for example, the agent can be forced to avoid specific actions that lead to states that can violate safety requirements. We have relied on the presence and absence of abstract states as features to learn rules that characterize faulty episodes. However, other types of features that consider temporal information regarding the agent's states and actions in faulty episodes (e.g., considering the sequence of abstract states) could provide additional relevant information to explain the occurrence of faulty episodes. However, learning such temporal patterns typically requires much more data to achieve accurate results and appeared to be unnecessary and therefore practically justified in our two case studies. In future work, we aim to investigate other feature representations that include temporal information to extract rules that characterize faulty episodes. We also intend to investigate in depth the use of STARLA for safety analysis before deploying

RL agents. Finally, we aim to perform a user study to understand how engineers can use such rules to make decisions about the safety of the RL agent under test.

This research takes a first step toward testing RL agents using a data-box genetic search. Our proposed testing approach and associated results have several practical implications. The generated faulty episodes and the corresponding rules that characterize them can be used for safety analysis and retraining. Indeed, analyzing the states and actions in the generated faulty episodes could help developers to (1) understand the root-causes of faults in the RL agent, and (2) analyze the safety risks at run-time based on the prevalence of such root causes in practice and the consequences of identified faults. Moreover, one can retrain the agent using some of the generated faulty episodes, guided by the rules, as a mechanism to improve the policy of RL agents.

2.7 Threats to Validity

In this section, we discuss the different threats to the validity of our study and describe how we mitigated them.

Internal threats concern the causal relationship between the treatment and the outcome. Invalid episodes generated by STARLA might threaten internal validity. To mitigate this threat and to ensure the validity of the generated final episodes, we have relied on state abstraction and the application of realistic state transformations when using the search operators. For instance, for crossover, instead of selecting random crossover points, we have used state abstraction to find a matching pair for the crossover point. Furthermore, to ensure the validity and the exploratory aspect of the mutation operator, we alter the state of the mutation point using realistic state transformation methods to produce realistic and feasible states that could happen in the real-world environment. Finally, the validity of the episodes is checked through their execution. Thus, we only retain valid failing episodes in our final results. Frequent replacement of states during executions may pose a potential threat to the validity of our study. However, we observed that the replaced states are highly similar in terms of cosine similarity, suggesting that the physical characteristics of the replaced states are also similar.

Our search approach relies on the specification of several thresholds that are context-dependent and vary from case to case. The threshold of the reward can change based on the expected minimum reward of the agent. For the reward fitness function in the *Cart-Pole* problem, we used a threshold equal to 70, while in the *Mountain Car* problem the reward threshold was -180. Based on experiments, we also realized that STARLA performs better when the threshold of the probability of functional fault fitness value is 95% and the threshold of the certainty level is 0.04. It is important to fine-tune these parameters for each case study to get optimal detection results. We acknowledge that the fine-tuning of STARLA parameters may require additional efforts from practitioners for more complex RL problems. Therefore, in future work we plan to investigate how to fine-tune STARLA parameters for more complex problems.

The choice of an inappropriate state abstraction method and level might also be a threat. To mitigate it, we have studied several state abstraction methods and have tried different abstraction levels to train our ML model. We have selected the best abstraction level that maximizes the accuracy of the model and significantly decreases the number of abstract states.

The current solution does not consider newly seen abstract states during testing. We acknowledge that this is a limitation of our testing approach, though any ML solution is always based on incomplete features, and what matters is whether the guidance provided to the test process is sufficiently effective. To mitigate the risk of missing abstract states in our feature representation, we have relied on a state abstraction method that considers a large number of concrete states in both the training phase and random executions of the RL agent. We computed the percentage of newly seen abstract states during the execution of the RL agent. We observed, on average, eight new abstract states out of 93 in the *Mountain Car* problem and 209 new abstract states out of 1035 in the *Cart-Pole* problem. Nevertheless, our results show that we trained accurate models based on known abstract states in both case studies. To further enhance the accuracy of the ML model, we can increase the size of the training dataset, thereby expanding the range of abstract states from which the model can learn. Additionally, we can retrain the ML model after newly identified abstract states are found, enabling the model to incorporate these states into the decision-making process.

Conclusion threats are concerned with the relationship between treatment and outcome. The randomness in our search approach leads to the generation of different episodes after each run of STARLA. To mitigate this threat in our experiments, we executed several runs of our search method and studied the distribution of the number of the detected faults for both our method and Random Testing.

Reliability threats concern the replicability of our study results. We rely on publicly available RL environments and provide online all the materials required to replicate our study results. This includes the set of the executed and generated episodes and the different configurations that we used in our experiments.

External threats concern the generalizability of our study. Due to the high computational expense of our experiments and the lack of publicly available, realistic RL agents, we relied on two case studies in this research. However, we mitigated this threat by using widely studied RL tasks which are considered as valid benchmark problems in many RL-related studies [16, 17, 29, 107]. However, our approach is customizable and can be applied to any other RL problem. Furthermore, in future work we aim to apply our testing approach on other RL problems to generalize the obtained results.

2.8 Related Work

Several approaches have been proposed in the literature to study the safety of RL agents during the training and execution phases. However, limited research has been targeted at testing RL-based systems.

In a very recent study, Tapple *et al.* [133] presented a search-based testing method for RL agents. The method relies on a backtracking-based, depth-first search algorithm [134]that is used in the RL agent's execution to identify a reference trace that solves the RL task and contains a set of boundary states that can lead to unsafe states. Test suites are generated by extracting the action traces that lead to these boundary states from the initial state. The main objective of these test suites is therefore to guide the RL agent toward safety-critical states. Furthermore, to evaluate the agent's performance, performance test suites are generated by using a genetic algorithm to create a diverse set of action traces starting from the reference trace. The average reward gained by the policy of the agent is then compared to the average reward from the fuzz trace executions to evaluate the agent's performance. However, this testing approach is only applicable to RL agents with a stochastic policy, while our focus is on testing agents with a deterministic policy interacting with a stochastic environment (which is normally the case in safety-critical domains). Another limitation is that this approach relies on finding a reference trace that (1) solves the RL task, and (2) contains boundary states that lead to all unsafe states. Given these requirements, such a reference trace may be difficult to find and may not lead to all possible safety-critical scenarios. Indeed, while this framework can identify safety-critical situations near boundary states, it may not be able to identify all potential safety issues in more complex environments. Finally, the approach requires the repetitive execution of all possible actions from the different states in the reference trace, making it computationally intensive and highly sensitive to the size of the action space.

Nikanjam *et al.* [101] presented a taxonomy of DRL faults and a tool to locate these faults in DRL programs. To build the taxonomy, they analyzed DRL programs on GitHub, mined Stack Overflow posts, and conducted a survey with 19 developers. They proposed *DRLinter*, a model-based fault detection tool that relies on static analysis of DRL programs and graph transformation rules to locate existing faults in DRL source code. Although we have similar objectives, this work differs greatly from ours, as we detect faults related to the execution of RL agents through the search for and generation of faulty episodes. Nonetheless, this work may complement our approach and could be used as a root cause analysis mechanism of the faults reported by our search approach.

Trujillo *et al.* [137] studied the reliability of neuron coverage [81] in testing DRL systems. They studied the correlation between coverage metrics and rewards obtained by two different models of DQN that were implemented for the *Mountain Car* problem [125]. They show that neuron coverage is not correlated to the agent's reward. For instance, reaching high coverage does not necessarily mean success in an RL task in terms of reward. They also showed that maximum coverage is obtained through excessive exploration of the agent, which leads to exploration of different actions that do not help maximize the agent's reward. Finally, they conclude that neuron coverage is not suitable to guide the testing of DRL systems. We therefore do not use search guided by neuron coverage as a baseline in our work.

Several approaches have been proposed in the literature to study the robustness of RL agents against adversarial attacks [58, 60, 105]. For example, Ilahi *et al.* [60] studied 28 adversarial attacks on RL and provided a taxonomy of existing attacks in the literature. They considered attacks that rely on perturbing (1) the state space, (2) the reward space,

(3) the action space, and (4) the model space, where one can perturb the model's learned parameters. They show that although many defense approaches are proposed to increase the safety of DRL-based systems, the robustness of such systems to all possible adversarial attacks is still an open issue. This is because the proposed defense techniques in the literature can respond to the types of attacks they are built for. Besides, Moosavi-Dezfooli *et al.* [96] argue that regardless of the number of adversarial examples added to the training data, they were able to generate new adversarial examples to alter the normal behavior of the system.

Huang *et al.* [58] studied the robustness of neural network policies in the presence of adversaries. They studied the effectiveness of black-box and white-box adversarial attacks on policy networks such as DQN [94], TRPO [114], and A3C [93], trained on Atari games [20]. They showed that adversarial attacks can significantly degrade the performance of the agent, even with small, imperceptible perturbations.

Pan *et al.* [105] studied the robustness of the reinforcement learning agent in the specific learning task of power system control. They proposed a new adversary in both white-box and black-box (using a surrogate model) scenarios. They studied the effectiveness of their method and compared it with random and weighted adversarial attacks previously proposed for power system controls [85, 103]. Moreover, they studied the robustness improvement of the agent trained with adversarial training.

Other existing approaches in the literature [15, 18, 49, 70, 72, 107, 128, 129, 141] have proposed adversarial training techniques to increase the robustness of RL agents to adversarial attacks. For example, Pattanaik *et al.* [107] proposed a training approach for DRL agents to increase their robustness to gradient-based adversarial attacks. They train the agent using adversarial samples generated from gradient-based attacks. They show that adding noise to the training episodes increases the robustness of the DRL agent to adversarial attacks.

Tan *et al.* [129] also proposed an adversarial training approach for DRL agents used for decision and control tasks. The purpose of their training approach is to increase the robustness of DRL agents against adversarial perturbations to the action space (within specific attack budgets). Consequently, they relied on gradient-based white-box adversarial attacks during the training phase of a DRL agent. They show that the proposed method increases the robustness of the agent against similar attacks.

Generative model-based approaches, such as those proposed by Li *et al.* [76], offer an alternative perspective to search-based testing by employing generative diffusion models for testing decision-making policies in Markov Decision Processes (MDPs). This method focuses on enhancing both test case diversity and effectiveness by leveraging novelty-based guidance mechanisms to encourage the generation of unique and failure-triggering test cases. Compared to search-based techniques like STARLA, which rely on genetic algorithms and machine learning models to target functional faults, generative model-based testing uses data-driven diffusion models to capture distributions of valid test cases and explore novel state spaces. However, such generative models can produce unrealistic states that may not align with the actual environment's characteristics.

These works differ from our testing approach, as we do not focus on the robustness of

RL agents to adversarial attacks. Rather, we test the policies of RL agents, without using any of their internal information, by relying on a genetic search to effectively find faulty episodes.

2.9 Conclusion

In this chapter, we propose STARLA, a data-box search-based approach to test the policy of DRL agents by effectively searching for faulty episodes. We rely on a dedicated genetic algorithm to detect functional faults. We make use of an ML model to predict DRL faults and guide the search toward faulty episodes. To this end, we use state abstraction techniques to group similar states of the agent and significantly reduce the state space. This helped us to increase the learnability of the ML models and build customized search operators. We showed that STARLA outperforms Random Testing as we find significantly more faults when considering the same testing budget. We also investigated how to extract rules that characterize the faulty episodes of RL agents using our search results. The goal was to help developers understand the conditions under which the agent fails and thus assess the risks of deploying the RL agent.

In future work, we aim to detect other types of faults, such as reward faults, and investigate the retraining of the RL agent using the generated faulty episodes. We aim to study the effectiveness of such episodes in improving the agent's policy. We also want to support the safety of RL-based critical systems by providing mechanisms based on ML and state abstraction to identify sub-episodes that may lead to hazards or critical faults.

Chapter 3

Runtime Safety Monitoring of the Reinforcement learning Agents

3.1 Overview

This chapter addresses TO_2, i.e., Runtime Safety Monitoring of the Reinforcement learning Agents. The content of this chapter has been accepted for publication in *IEEE Transactions on Software Engineering (TSE)* [156] under the title of *SMARLA: A Safety Monitoring Approach for Deep Reinforcement Learning Agents*.

As we discussed in chapter 1, regardless of dedicating significant effort to testing DRL agents across various scenarios [133, 156], ensuring runtime safety remains challenging due to the extremely large agent state space and number of possible execution scenarios. As a result, the runtime monitoring of RL agents is crucial to guarantee their safety, as recommended by industry standards such as ISO 26262 [62] and ISO/PAS 21448 [61] in the automotive domain. While such standards do not explicitly address RL agents, given the increasingly frequent integration of DRL in autonomous systems [67,131], it is essential that such agents adhere to these established safety standards to maintain the safety integrity levels required in many safety-critical domains such as automotive.

Runtime safety monitoring, however, entails the efficient, continuous observation and risk assessment of the states and actions performed by DRL agents. It must also provide a means to predict early and thus prevent unsafe behavior before it leads to catastrophic consequences. If the predictions of safety violations are sufficiently early, corrective measures or safety mechanisms can indeed be applied in most contexts (e.g., giving the control to a human driver or activating automatic emergency brakes in autonomous driving systems). This is highly important in safety-critical applications, such as autonomous vehicles, where even a single violation can have severe consequences. Standards such as ISO 26262 (automotive), DO-178C (aviation), and IEC 60601 (medical devices) emphasize the need to predict known unsafe situations to achieve acceptable safety levels. Moreover, runtime safety monitoring is essential when DRL agents are deployed in uncertain and dynamic environments. DRL agents often are trained on simulators and deployed in environments

where the dynamics can change over time, and the agent's learned policy may no longer be optimal or safe. By monitoring the agents, it is not only possible to ensure safety but also to understand whether further training is required or not. Safety monitoring can also provide insights into the learning process of DRL agents. By analyzing the agent's behavior and the safety violations detected, it is possible to gain a better understanding of the agent's decision-making process and identify areas for improvement.

However, the runtime safety monitoring of DRL agents presents challenges. DRL agents often operate in environments with extremely large state spaces, which can make it challenging to monitor their behavior in real-time and predict potential safety violations [35]. Further, DRL agents learn and adapt their behavior based on interactions with the environment, which introduces uncertainty regarding their policies and resulting actions. Such uncertainty can make it difficult to predict and monitor the agents' behavior accurately. Safety monitoring needs to account for it and consider methods that can scale to large state spaces and can handle the dynamic and evolving nature of DRL agents' behavior.

Existing safety monitoring techniques for regular software systems often rely on formal verification to ensure compliance with safety constraints [50]. However, when it comes to DRL policies, formally verifying their behavior to satisfy safety properties becomes a computationally expensive and an NP-complete problem [66, 83]. Further, the black-box nature of DRL policies makes it challenging to analyze and verify them [35].

There are three distinct types of safety monitoring for DRL agents, each based on the type of information they require: white-box, grey-box, and black-box. White-box safety monitoring requires comprehensive access to internal information of the DRL agent, including its architecture, weights, reward function, and training dataset. Gray-box safety monitoring requires access to the agent's inputs, outputs, and training dataset. Black-box safety monitoring, however, limits its analysis to the observed input-output behavior, including the agent's Q-values. Q-values assign numerical values to actions in states, reflecting their potential to lead to a reward or a penalty [10]. Monitoring DRL agents using a black-box approach is often practical, as testers and safety engineers usually lack access to the agent's internal structure or training dataset [2, 3, 156].

Moreover, most of the existing works on RL safety in the literature have primarily focused on safe exploration strategies to enhance the safety of RL agents during the learning process [102, 108] and RL shielding strategies [7, 37, 69] to block unsafe actions at runtime and force the agent to deviate from its policy. In contrast to these works, we propose a black-box safety monitoring approach, specifically tailored to DRL agents, where the goal is to predict safety violations at runtime, as early as possible, by monitoring the behavior of the agent.

In this chapter, we present *SMARLA*, a Safety Monitoring Approach for Reinforcement Learning Agents. *SMARLA* is a monitoring approach that uses machine learning to predict safety violations, accurately and early, during the execution of DRL agents. We train *SMARLA* based on the testing results of DRL agents. This is important in practice, as testing data contain useful information (i.e., both known safe and unsafe situations) that can be exploited for monitoring. We leverage state abstraction methods [1,64,75] to reduce the large state space usually associated with DRL agents and thus increase the learnability

of machine learning models to predict violations. *SMARLA* currently focuses on Q-learning algorithms. Q-learning is a widely used and common type of RL algorithms [63], where the objective is to find the optimal action-selection policy through Q-values. However, it is important to note that *SMARLA* operates as a black-box monitoring method, as it does not require access to the internals of the Q-learning-based agent or the training dataset. *SMARLA* only requires access to the Q-values of the agent which represent the expected cumulative reward for taking an action in a given state, providing insight into the agent's decision-making process. As a result, *SMARLA* is also agnostic to the type of DRL agent's inputs, which is a practical advantage.

To evaluate the effectiveness of *SMARLA*, we implemented our safety monitor for three well-known RL problems that serve as benchmarks in the RL community [16,17,21,107,110, 146, 151]. Our experimental results suggest that *SMARLA* can accurately predict safety violations, while maintaining a low false positive rate, long before violation occurrences. We thus provide evidence of the potential benefits of *SMARLA* as it is a promising solution to prevent damages and mitigate risks associated with RL agents. The main contributions of our work are as follows:

- We introduce *SMARLA*, a novel black-box safety monitoring approach specifically designed for DRL agents. *SMARLA* is black-box as it only requires access to agents' Q-values as a proxy of its decision-making process. Based on these Q-values, we apply state abstraction to reduce the state space and increase the learnability of a model used to predict safety violations. *SMARLA* therefore aims to predict DRL safety violations at an early stage, enabling proactive and preventive mechanisms for ensuring the safety of DRL-based systems.

- As part of *SMARLA*, we propose a highly accurate machine learning solution to estimate the probability of safety violations at each time step of the DRL agent's execution, along with confidence intervals.

- We investigate alternative decision procedures, based on the confidence intervals of violation probability estimates, in terms of both accuracy and decision time.

- We implement *SMARLA* on three widely used RL benchmark problems and investigate trade-offs between early and accurate predictions of safety violations. Results show the effectiveness of *SMARLA* in accurately predicting safety violations and demonstrate its potential for real-world applications.

- We conducted a qualitative analysis to provide insights into SMARLA's practical application and explain its prediction results, offering an understanding of the conditions under which the risk increases.

- We provide a prototype tool for our safety monitoring solution. We also include all the necessary data and configurations to replicate our experiments and results. However, this comprehensive replication package will only be made publicly available upon acceptance.

The remainder of this chapter is structured as follows. Section 3.2 defines our research problem and outlines the underlying assumptions. Section 3.3 describes our approach. Section 3.4 presents our empirical evaluation and the corresponding results. Section 3.5 discusses our empirical results. Section 3.6 investigates threats to the validity of our results. Section 3.7 describes related works. Finally, Section 3.8 concludes this chapter.

3.2 Problem Definition

We present in this section a comprehensive overview of our research problem and the assumptions that underlie our work.

3.2.1 Problem

RL has become an essential part of intelligent systems that can learn and adapt to complex and dynamic environments. It has been increasingly applied in various safety-critical areas such as autonomous driving, healthcare, and industrial control systems. Therefore, the development of a reliable runtime safety monitoring approach for RL agents is critical for ensuring the safety of the agent and other entities in the environment. A safety monitor aims to predict potential safety violations and trigger safety mechanisms to take corrective or preventive actions in real time. However, designing an algorithm that can learn to predict safety violations accurately and as early as possible presents a significant challenge. In this work, we propose to build a safety violation prediction approach based on *Random Forest* algorithm and features based on abstract states to effectively predict unsafe episodes. Our safety monitoring approach will then use such model to monitor states and actions over time and predict the probability of safety violations before they occur.

3.2.2 Assumptions

In this work, we focus on RL agents with discrete actions and a deterministic policy. Indeed, our state abstraction method is designed for discrete action spaces. We acknowledge that this is a limitation of our approach. However, several studies have explored discretization approaches to effectively transform continuous action spaces into discrete ones [132, 135]. Addressing continuous action spaces is nonetheless beyond the scope of this research. Moreover, we assume that a deterministic policy is realistic since, in many application domains (specifically in safety-critical domains), uncertainty is not acceptable and random actions should be avoided [156]. Further, we build our work on model-free Q-learning RL algorithms since our abstraction method relies on Q-values. Note that such algorithms are commonly used and have been extensively researched [40, 55, 63, 104, 127, 140, 144], hence our choice to target them. Beyond that, Q-learning algorithms are versatile and applicable to a wide range of problems with discrete action spaces. Also, they are relatively sample efficient, and due to the off-policy setting, Q-learning algorithms can learn from experiences generated by any policy, making them valuable for safety-critical problems where

exploration is costly or dangerous [19,42,118]. Note that Q-values leveraged by Q-learning algorithms directly represent the expected cumulative reward for taking an action in a given state, providing insight into the agent's decision-making process. This interpretability can be valuable in domains where understanding the agent's behavior is crucial, such as safety-critical domains.

3.3 Approach

We present in this section our safety monitoring approach for DRL agents, along with its various components.

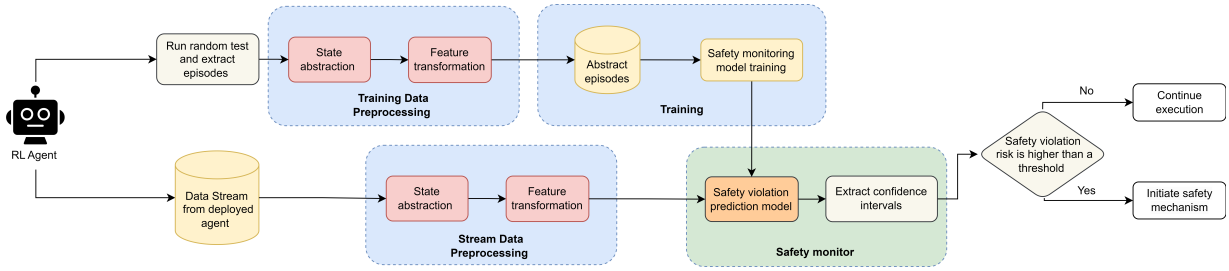

Figure 3.1: Overview of SMARLA

3.3.1 Overview of the Approach

As depicted in Figure 3.1, we propose a safety monitoring system for DRL agents to predict safety violations of DRL agents as early as possible. The early detection of unsafe episodes is indeed crucial for any safety-critical system to enable prompt corrective actions to be taken and thus prevent unsafe behavior of the agent. Our approach predicts safety violations by observing the behavior of the RL agent rather than directly monitoring the state of the environment. Observing and processing such state information can be computationally expensive and inefficient, especially in environments with large or continuous state spaces. In contrast, our method is agnostic to the agent's input and leverages the agent's Q-values as features throughout the episodes. Q-values summarize the expected future rewards of all possible actions in given states, providing a more abstract and computationally efficient representation of the agent's behavior. By focusing on Q-values instead of raw state data, *SMARLA* can be seamlessly applied to agents with various types of input. Moreover, relying on Q-values significantly reduces the computational complexity involved in monitoring. This allows us to employ a lightweight machine learning model, such as *Random Forest*, which is shown to be effective for this task (as reported in Section 3.4.4.1). While it might be possible to predict unsafe episodes using state information, the drawback is that it would require handling a higher-dimension and more complex dataset, potentially necessitating the development of resource-intensive models.

To train the ML model we randomly execute the RL agent and extract the generated episodes. These episodes are labeled as either safe or unsafe. Since the state space is very

large, we rely on state abstraction to reduce the state space and enhance the learnability of our ML model. Once we have trained the model on these labeled episodes, we use it to monitor the behavior of the agent and estimate the probability of encountering an unsafe state while an episode is being executed. We rely on the confidence intervals of such probability to accurately determine the optimal time step to trigger safety mechanisms.

3.3.2 Training of the safety violation prediction model

We need a lightweight ML model that can accurately classify RL episodes as safe or unsafe, and be effectively deployed on resource-constrained edge devices [23]. This is especially relevant in real-world scenarios where safety monitoring is essential, such as for RL agents operating on drones or small autonomous robotic systems [13, 38]. Furthermore, even in resource-rich environments (e.g., some autonomous driving systems), having computational resources for core functionalities does not guarantee the availability of substantial unused capacity for implementing additional safety monitoring. Safety monitoring models should therefore be efficient and non-intrusive, ensuring they do not overburden the system's existing resources. Thus, we exclude DNN models and choose *Random Forest* as a machine learning model because of (1) its ability to handle a large number of features, (2) its proven robustness to overfitting, and (3) its efficiency in providing classification results at inference time [22, 41, 156].

Moreover, the empirical study presented by Caruana *et al.* [27] supports the use of *Random Forest* due to its consistently high performance across multiple evaluation metrics, robustness to overfitting, and effective probability calibration. The authors report that *Random Forest* maintains balanced performance in both traditional metrics, such as accuracy and F-score, and advanced metrics such as ROC Area and cross-entropy. Additionally, *Random Forest* requires less parameter tuning compared to DNN models, making them practical for real-world applications.

We also conducted comparison studies with other ML models, such as *K-Nearest Neighbor* and *Decision Trees*. However, *Random Forest* provided the most accurate classification results. Since this aspect is not the primary focus of our work, we omitted the results of these experiments in this chapter and reported the results in our replication package.

3.3.2.1 Training Data Collection

To build our safety violation prediction model, we collect the training data by randomly executing the RL agent starting from different initial states. Episodes are labeled unsafe if safety violation is observed within episodes; otherwise, they are labeled as safe. The goal of this process is to generate diverse episodes and form a training dataset that encompasses both safe and unsafe episodes. For each execution of the agent, we extract the corresponding episode (i.e., pairs of states and actions) along with the different Q^*-values in each state. Such data will be used to build the abstract states as we describe in the following.

3.3.2.2 State Abstraction and Feature Transformation

Once we collect the training episodes, we need to build the set of abstract states and map each concrete state to its corresponding abstract state. This is meant to reduce the state space and thus enhance the learnability of the ML model. To build abstract states, we use the Q^*-irrelevance abstraction method, which was explained in Section 2.2.2. The abstraction algorithm takes the concrete states as input and an abstraction level d as a control parameter. The algorithm attempts to find the abstract state $s^{\phi_d} \in S_\phi$ corresponding to each concrete state by evaluating the Q^*-values of all available actions, as explained in Section 2.2.2. For each concrete state, if the algorithm detects a match with an abstract state from a previously processed concrete state, it assigns the same abstract state to the current concrete state. If no matching abstract state is found, a new abstract state is generated.

After building the set of abstract states, we replace each concrete state in the training episodes with its corresponding abstract state to enable effective learning of the ML model. To train the model, we translate each episode into a feature vector that determines whether abstract states are present or not in episodes. Specifically, we represent each episode using a binary feature vector that encodes the presence (1) or absence (0) of each abstract state S_j^ϕ within the episode, where $1 \leq j \leq n$, and n represents the total number of abstract states. An example of a feature vector is provided as follows:

	S_1^ϕ	S_2^ϕ	$\cdots$	S_j^ϕ	$\cdots$	S_{n-1}^ϕ	S_n^ϕ
$episode_i$	1	0	$\cdots$	1	$\cdots$	1	0

The main advantage of this representation is that it significantly reduces the feature space by relying on the presence or absence of the abstract states. Specifically, considering n different abstract states, the feature space of this representation is only 2^n. Alternatively, if needed, we can rely on features based on the frequency of abstract states, which enhance the representation by accounting for how often each abstract state is observed during the execution of an episode, at the cost of course of a larger feature space. It is worth noting that we only consider abstract classes that have been observed in the training dataset of the ML model, which we anticipate will be relatively comprehensive. However, the feature representation does not account for the order of abstract states within episodes, which may be a weakness if we are unable to accurately predict safety violations as a result. Empirical results will tell whether these two potential issues are significant in practice.

3.3.3 Safety Violation Prediction Model in Operation

Runtime monitors are designed to detect and prevent safety violations while the system is in operation. Once the training is complete, we deploy the safety violation prediction model alongside the RL agent. The safety violation prediction model monitors episodes

and estimates the probability of a safety violation at each state. We should note that the streamed episodes have different lengths. We then apply the stream data pre-processing step that generates binary vectors of length equal to the size of abstract states (i.e., equal to n). This process is necessary to be able to run the safety monitor at runtime. Stream data pre-processing is accomplished by using state abstraction and feature representation techniques, which are explained in detail in Section 3.3.2.2.

3.3.3.1 Estimating The Probability of Safety Violation

At each time step, the data stream is received and episodes are transformed into binary vectors based on state abstraction and feature representation. Each binary vector indicates the presence (1) or absence (0) of abstract states observed so far in the corresponding episode. The safety violation prediction model uses these binary inputs to estimate the probability of safety violation based on the abstract states visited so far.

Let us suppose that the agent is at time step t, and the running episode is as follows:

$$episode_i(t) = [(s_1, a_1), ..., (s_{t-1}, a_{t-1}), (s_t, a_t)] \tag{3.1}$$

The stream data pre-processing process converts these episodes into binary feature vectors showing the presence and absence of the abstract states visited so far (until time step t). The transformed episode is as follows:

	S_1^ϕ	S_2^ϕ	$\cdots$	S_j^ϕ	$\cdots$	S_{n-1}^ϕ	S_n^ϕ
$episode_i(t)$	0	0	$\cdots$	1	$\cdots$	1	0

At each time step t, the episode is fed into the safety violation prediction model. Then it estimates the probability of safety violation $P_{e_i}(t)$ at time step t. In other words, the safety violation prediction model captures the relationship between the presence of abstract states in an episode and the occurrence of safety violations.

We should note that state abstraction allows us not to depend on the specific occurrence of concrete states in RL episodes when predicting safety violations in operation. State abstraction thus generalizes patterns to concrete states that have possibly not been encountered in operation by analyzing change patterns in the agent's Q-values, which are indicative of possible unsafe states. In other words, in our predictive model, we operate under the hypothesis that patterns of Q-value changes observed in training episodes that contain safety violations are likely to reappear in operation. This assumption allows *SMARLA* to extend its predictive capability beyond the concrete episodes seen during training, by recognizing and responding to analogous change patterns, even in the presence of unseen concrete states. Nonetheless, during the prediction process, we may encounter concrete states belonging to unseen abstract states—those not present in the training data. Such situations require careful handling to ensure safety. Two strategies can be applied:

- Stop the Execution: In the presence of an unforeseen situation, stopping the execution can prevent potential risks associated with unseen states. This conservative approach prioritizes safety by halting operations when the system encounters unfamiliar states that could lead to unknown hazards.

- Ignore the Unseen Abstract State: Alternatively, the system can ignore the unseen state and continue predicting safety violations based on other observed states in the episode. This approach assumes that the unseen state does not significantly impact the safety and leverages the information from other known states to make predictions.

In our experiments, we observed that episodes had only a few unseen abstract states and could still be correctly predicted using only the seen abstract states. Therefore, we chose the latter approach.

We should note that the *Random Forest* model used in this study not only generates class predictions but also calculates probabilities for these classifications, using an ensemble of decision trees (also called estimators). Each tree in the forest is constructed from a bootstrap sample of the dataset, and at each decision node, a randomly selected subset of features is considered. The trees individually predict whether a given episode is *"safe"* or *"unsafe"*. The overall probability for each class is derived by averaging the predictions across all trees in the forest. By convention, a probability threshold of 50% is applied, whereby probabilities of 50% or higher result in an *"unsafe"* classification, while probabilities below 50% lead to a *"safe"* classification. These output probabilities from a Random Forest are crucial as they reflect the confidence level of the model's predictions, providing valuable guidance for decision-making, especially in scenarios where precise and timely detection of safety violations is paramount.

3.3.3.2 Safety Violation Prediction

Based on the estimated probabilities of safety violations we compute the predictions' confidence intervals [36] to determine the time step at which a safety violation is likely to occur with high confidence. At this time step, corrective or preventive measures can be initiated to avoid damage and harm.

To determine the confidence intervals of probabilities of safety violations, we rely on the predictions of the individual estimators in the *Random Forest* model. At each time step for a given episode, each estimator predicts the probability of safety violation. Based on the mean prediction and standard deviation, we calculate the confidence intervals at each time step t as follows:

$$CI(t) = \bar{X(t)} \pm Z \times \frac{\sigma}{\sqrt{m}} \tag{3.2}$$

where $\bar{X(t)}$ represents the mean predicted probability at time step t, Z is the critical value of the normal distribution with $Z = 1.96$ for a 95% confidence interval, σ is the

standard deviation, and m is the number of decision tree estimators [36]. The lower and upper bounds of the confidence intervals are as follows:

$$[Low(t), Up(t)] = [\bar{X(t)} - \frac{\sigma}{\sqrt{m}}, \bar{X(t)} + \frac{\sigma}{\sqrt{m}}] \tag{3.3}$$

More specifically, a 95% confidence interval indicates that, if such intervals were constructed from numerous samples (specifically, the output probabilities of each estimator in a *Random Forest*), there is a 95% likelihood that the probability of a safety violation predicted by the *Random Forest* model will fall between the lower and upper bounds of this interval. The interval is centered on the predicted probability of a safety violation and extends symmetrically in both directions. The remaining 5% probability, not covered by the interval, is equally divided between the distribution's two tails, allocating 2.5% to the lower tail and 2.5% to the upper tail. Consequently, at any time step t, there is a 2.5% chance that the probability of a safety violation will be either lower than $Low(t)$ or higher than $Up(t)$.

SMARLA supports three different decision criteria, based on the confidence intervals of the estimated violation probability, to determine the appropriate time step to classify the episode as unsafe and trigger safety mechanisms. We should note that all decision criteria rely on a threshold that may be adjusted according to the agent and the specific context.

- **Upper Bound of the Confidence Interval:** This first decision criterion involves using the upper bound of the confidence interval to classify an episode as unsafe. If the upper bound of the confidence interval is equal to or greater than a certain prediction threshold, we classify the episode as unsafe. We consider this approach conservative because it allows for earlier predictions of safety violations compared to using the output probability or the lower bound of the confidence interval. Although this method provides an earlier prediction of safety violations, it might lead to a higher number of false positives. This is suitable for scenarios where early detection is critical, even at the cost of some false positives.

- **Output Probability:** The second decision criterion is based on the actual output probability of safety violation being equal to or greater than a certain prediction threshold. This criterion provides a trade-off between timeliness and accuracy for predicting safety violations. It typically results in fewer false positives than using the upper bound but may delay the detection of unsafe episodes in comparison. This method is appropriate for environments where accuracy is more critical than early detection, minimizing unnecessary interventions.

- **Lower Bound of the Confidence Interval:** The third decision criterion uses the lower bound of the confidence interval. If the lower bound is greater than a certain prediction threshold, the episode is predicted as unsafe. This decision criterion minimizes false positives but at the potential cost of detecting unsafe episodes later than the other two criteria. This approach is preferred in situations where it is critical to avoid false positives and where the consequences of late detection are not severe.

In our case studies, we rely on the upper bound because, to ensure safety, we take a conservative approach and rather err on the side of caution. In Section 3.4.4.2, we detail the decision criteria and report empirical results that confirm this is the best choice in our case studies.

3.4 Empirical Evaluation

This section describes the empirical evaluation of our monitoring approach, including our research questions, the used case studies, our experiments, and the obtained results.

3.4.1 Research Questions

Our empirical evaluation is designed to answer the following research questions.

RQ1. How accurately and early can we predict safety violations during the execution of episodes? This research question aims to investigate how accurately and early our approach can predict safety violations of the RL agent during its execution. Preferably, high accuracy should be reached as early as possible before the occurrence of safety violations to enable the effective activation of safety mechanisms.

RQ2. How can the safety monitor determine when to trust the prediction of safety violations? This research question investigates the application of confidence intervals, based on *Random Forest* models, to enable the safety monitoring model identify the earliest time step at which the prediction of violations can be trusted.

RQ3. What is the effect of the prediction threshold on the safety monitoring system? This research question explores the impact of varying the prediction threshold values on our safety monitoring approach. We aim to understand how different prediction thresholds affect both the prediction time and the accuracy of our safety monitoring system during operation. Our goal is to understand the trade-offs between making early predictions of safety violations and maintaining high prediction accuracy. In other words, we want to assess how setting lower or higher thresholds influences SMARLA's ability to promptly detect safety violations while minimizing false positives and false negatives.

RQ4. What is the effect of the abstraction level on the safety monitoring system? We aim in this research question to investigate if and how different levels of state abstraction affect the safety violation prediction capabilities of the model. Specifically, we want to study the impact of state abstraction levels on (1) the accuracy of the safety violation prediction model after training, and (2) the accuracy of the ML model in operation. Our goal is to gain insights into the possible trade-offs between the size of the feature space and the granularity of information captured by features, both determined by the abstraction level. Such an analysis aims to provide guidance in selecting proper abstraction levels in practice.

3.4.2 Case Studies

To validate our safety monitoring approach, we considered three well-known case studies widely used as benchmarks in RL-related research [16,17,29,43,107]: the *Cart-Pole* balancing problem [32], the *Mountain-Car* problem [95], and the *Highway Driving* Problem [74] which we describe in detail in the following sections.

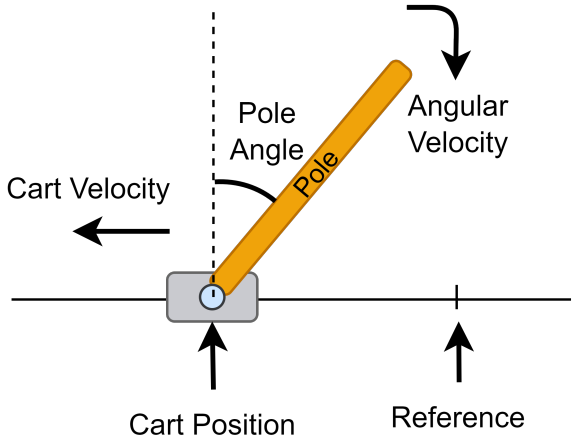

Figure 3.2: *Cart-Pole* case study

3.4.2.1 Cart-Pole Balancing Problem

The *Cart-Pole* balancing problem involves a pole that is attached to a cart moving on a track. The cart can move horizontally in both directions within a specific range. The objective is to keep the pole upright by moving the cart and adjusting its velocity. As shown in Figure 3.2, we characterize the state of the agent by four elements: (1) the position of the cart, (2) the velocity of the cart, (3) the angle of the pole, and (4) the angular velocity of the pole. There are two discrete actions that can be used to control the cart: move to the left, and move to the right.

A reward of +1 is granted for each time step when the pole is still upright. An episode terminates if (1) the cart is away from the center with a distance of more than 2.4 units, or (2) the angle of the pole is larger than 12 degrees, or (3) the pole remains upright for 200 time steps. An episode is considered unsafe if the cart moves away from the center with a distance above 2.4 units, regardless of the accumulated reward. In such a situation, the cart can pass the border and cause damage, which is therefore considered a safety violation.

3.4.2.2 Mountain-Car Problem

In the *Mountain-Car* problem, an under-powered car is placed in a valley between two hills and tries to reach a goal state on the top of the right hill. Since the gravity is stronger than the engine of the car, the car cannot climb up the steep slope even with full throttle. The objective is to control the car in such a way that it can accumulate enough momentum to eventually reach the goal state on top of the right hill as soon as possible. We characterize the state of the agent with two elements: (1) the location of the car along the x-axis, and (2) the velocity of the car as illustrated in Figure 3.3. The agent controls the car with three actions: (1) accelerate to the left, (2) accelerate to the right, and (3) do not accelerate.

A penalty of -1 is applied for each time step until reaching the goal. An episode terminates if the car (1) reaches the goal state, (2) crosses the left border (in this case the reward is -200), or (3) exceeds the limit of 200 time steps. A safety violation is simulated by considering the crossing of the left border of the environment as an irrecoverable unsafe state that poses potential damage to the car. Consequently, when the car crosses the left border of the environment, it triggers a safety violation, leading to the termination of the episode. This modification allows us to assess the effectiveness of *SMARLA* in predicting safety violations.

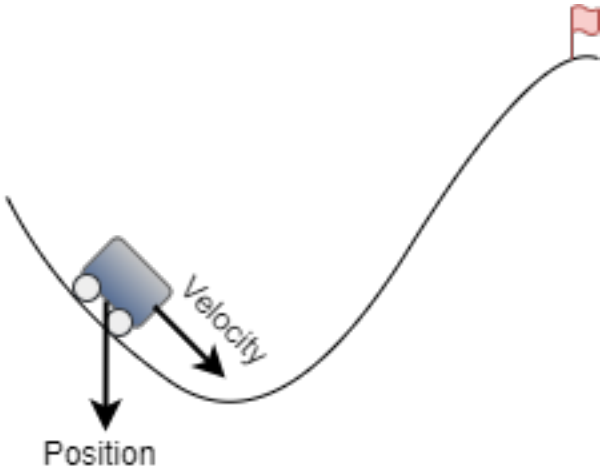

Figure 3.3: *Mountain-Car* case study

3.4.2.3 The Highway Driving Problem

In the *Highway Driving* problem, an RL agent is responsible for driving a car on a highway with three lanes, as shown in Figure 3.4. In this case study, we have a partially observable environment where the state information is only observable within a certain range. The objective of the agent is to drive along the highway at a high speed without colliding with other vehicles, for as long as possible. The environment is characterized by a continuous state space that includes kinematic details of the ego vehicle and other

vehicles on the road, amounting to a total of 15 parameters. The discrete action space available to the vehicle consists of five high-level actions: (1) moving right, (2) moving left, (3) accelerating, (4) remaining idle, and (5) braking. The agent receives a penalty of -10 for collisions, a 0.1 reward for maintaining its position in the rightmost lane, and a 0.4 reward for driving at high speeds (i.e., 20 to 30), which diminishes linearly for speeds outside the 20-30 range. Finally, the reward is normalized to the range of [0,1]. An episode ends under two conditions: (1) if the car collides with another vehicle, or (2) if it exceeds the maximum duration of episodes (30 time steps in our case) without any collision. An episode is deemed unsafe if the ego vehicle collides with another vehicle on the highway, constituting a failure to maintain a safe driving session.

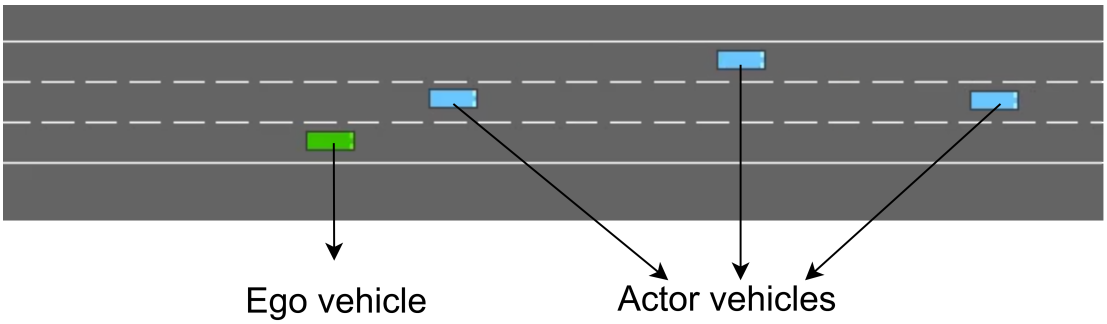

Figure 3.4: *Highway Driving* case study

3.4.3 Implementation

We implemented the RL agents for publicly available RL case studies using stable baselines [56]. To develop these agents, we used a DQN policy network [94] with the standard settings provided by stable baselines. Specifically, we employed Double Q-learning [139] and dueling DQN [144].

We trained the *Cart-Pole* RL agent for 70,000 time steps. The trained agent achieved an average reward of 193. On average, the pole remains upright for 193 time steps out of a maximum of 200. The *Mountain Car* agent was trained for 90,000 time steps. Considering the penalty of -1 applied at each time step in this specific case study, the agent achieves an average reward of -126, with an average episode duration of 112. We trained the *Highway Driving* RL agent for 70,000 time steps. On average, the car drives safely without any collision for 29 time steps out of a maximum of 30.

To train our safety monitor, we sampled episodes through random execution of the RL agent for each case study. Safety monitor's training data for *Cart-Pole* includes 2200 episodes from which 215 are unsafe, for *Mountain-Car* 2200 episodes with 279 unsafe episodes, and for *Highway* case study 4000 episodes with 222 unsafe episodes. Our safety monitor is based on a Random Forest classifier with standard hyperparameter values. We should note that since our safety monitoring approach is black-box, it is designed to predict safety violations without depending on the training level of the DRL agents. This approach

allows us to focus exclusively on the detection and prediction of safety violations, which is crucial for applications where the primary concern is maintaining operational safety rather than achieving training optimality.

Also, we empirically determined that suitable abstraction levels were 0.11 for *Cart-Pole*, five for *Mountain-Car* and 0.2 for the *Highway Driving* problem. Based on these abstraction levels, the number of abstract states is 1659 for *Cart-Pole* with abstraction level 0.11 (compared to 424,962 concrete states), and the number of abstract states is 13,171 for *Mountain-Car* with abstraction level 5 (compared to 247,035 concrete states). Finally, in the *Highway Driving* problem the number of abstract states with abstraction level 0.2 is 136 (compared to 115,730 concrete states). Also, with these levels, *SMARLA* achieved the highest F1-score and the earliest prediction of safety violations (explained in detail in Section 3.4.4.4).

It is important to note that some safety monitoring methods were not considered for comparison due to fundamental differences in their assumptions and objectives. For example, many approaches are white-box [54, 89, 90], assuming access to the internals of the agent, which contradicts our focus on black-box monitoring scenarios. Additionally, methods that depend on DNN models or image inputs rely on image transformations [148], whereas *SMARLA* is designed to be agnostic to the agent's input type, thus widening its applicability. Several existing methods, such as forward filtering-based and model-checking-based monitoring approaches [65], rely on detailed models of the environment, which are often not available or are too complex to model. *SMARLA* is designed for model-free RL algorithms, making it more applicable to real-world scenarios where environment models are difficult to develop. These distinctions justify the exclusion of such methods from our evaluation, as they do not align with the goals and assumptions of our work.

3.4.4 Evaluation and Results

3.4.4.1 RQ1. How accurately and early can we predict safety violations during the execution of episodes?

To answer this research question, we performed a series of n random executions ($n = 1000$) of the RL agent and extracted the corresponding episodes $e_{1 \leq i \leq n}$. To build our ground truth, these episodes were labeled as either safe or unsafe, taking into account the presence or absence of safety violations observed within each episode. We should note that in all case studies, there was a maximum of one safety violation per episode, occurring at the end of an unsafe episode as one of the termination criteria. We monitored the execution of each episode at each time step with *SMARLA*. As described in Section 3.3.3.2, when the upper bound of the confidence interval $Up(t)$ is greater than 50% during the execution of the episode, *SMARLA* classifies the episode as unsafe. For each case study, we computed the number of successfully predicted safety violations, measured the prediction precision, recall, and F1-scores at each time step, over the set of episodes, and depict the results in Figures 3.5, 3.6 and 3.7. Note that in this research, we report the weighted precision, recall, and F1-scores to show the performance of *SMARLA* in real-world situations, where

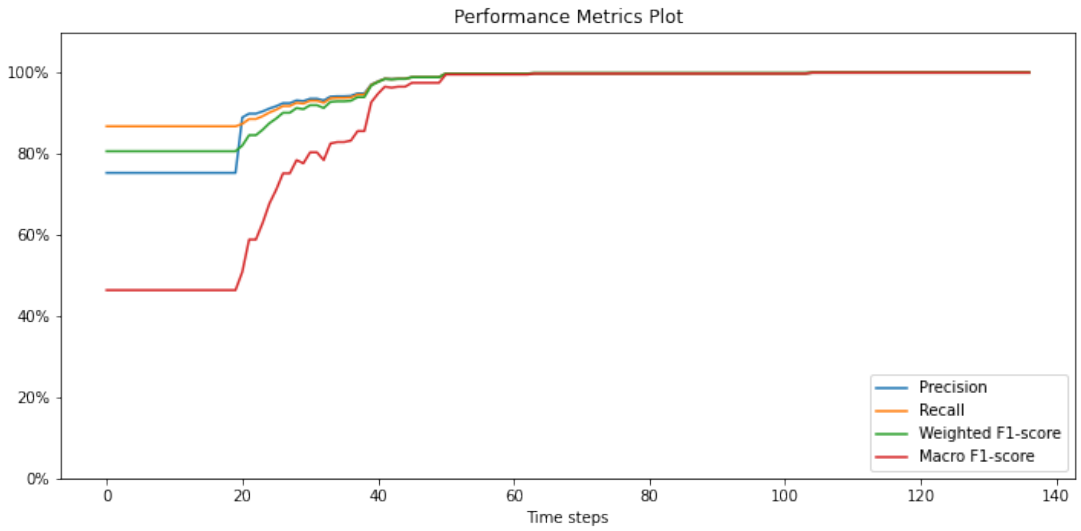

Figure 3.5: Performance of the safety violation prediction models in the *Mountain-Car* case study

operational data for predicting safety violations is typically imbalanced. Additionally, we report the macro F1-score, which calculates the F1-score for each class individually and then averages them. Unlike the weighted F1-score, which can be influenced by the majority class, the macro F1-score ensures that each class is treated equally. This provides a more balanced and accurate evaluation of our safety monitoring model's performance across all classes (i.e., safe and unsafe episodes), regardless of their prevalence. Note that in the following when we refer to the F1-score in our work, we are specifically referring to the F1-macro score, to cope with the class imbalance problem.

These figures show a consistent pattern where the precision, recall, and F1-score exhibit a general increase over time before eventually reaching a plateau.

SMARLA's accuracy is thus improving over time as episodes execute. Ultimately, our safety monitor correctly predicted 99 (out of 99), 132 (out of 132), and 40 (out of 59) safety violations in the *Cart-Pole*, *Mountain-Car* and *Highway Driving* case studies, respectively. We obtained highly accurate safety violation prediction results for *Cart-Pole* and *Mountain-Car* case studies after roughly 50 times steps, while in the *Highway Driving* case study we obtained highly accurate predictions after 10 time steps. In the *Cart-Pole* case study, we achieved precision, recall, and a weighted F1-score of 98.4%, as well as an F1-score of 95%, after time step 59. Furthermore, in the *Mountain-Car* case study, we obtained a precision of 99.8%, a recall of 99.8%, a weighted F1-score of 99.8% and an F1-score of 99.5% after 50 time steps. Finally, in the *Highway Driving* case study we achieved a precision of 96.3%, a recall of 96.6%, a weighted F1-score of 96.4% and an F1-score of 82.1%, after time step 10. In all case studies, results highlight *SMARLA*'s capability to provide accurate predictions of safety violations, relatively early, roughly halfway through the episode's execution as described next. Reasons for the lower F1-score for *Highway Driving* are discussed below and are not related to *SMARLA*.

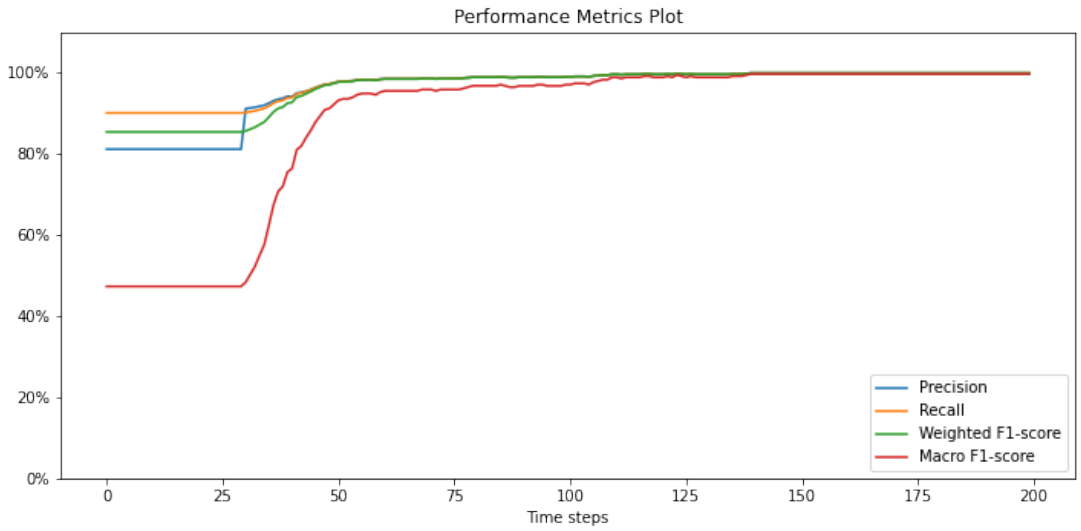

Figure 3.6: Performance of the safety violation prediction models in the *Cart-Pole* case study

In detail, the precision, recall, and F1-score plateau at 98% from time step 106 in *Cart-Pole* where the average length of the episodes is 193 and the minimum length is 118. At time step 106, on average, 45% of the steps within the episodes remain to execute until reaching an unsafe state. This indicates that there is significant time to apply safety mechanisms.

Similarly, in the *Mountain-Car* case study, we observed early accurate predictions of safety violations with *SMARLA*. This is indicated by the precision, recall, and F1-score plateauing from time step 50. The average and minimum episode lengths in this case study are 112 and 95. At this time step, an average of 55% of the steps within the episodes remain to execute. These results further validate the safety violation prediction model's ability to anticipate safety violations long before they occur.

Finally, in our *Highway Driving* case study, by time step 15, the F1-score of *SMARLA* reached 85% while the weighted F1-score, precision and recall plateaus at 97%. Also, at this time step, on average 48% of the episodes remain to execute until reaching an unsafe state. We observe that we achieved a relatively lower F1-score in *Highway Driving* case study compared to the *Cart-Pole* and *Mountain-Car* case studies. Since this might be due to the lack of information to predict safety violations, we explored a more informative feature representation, that is features based on frequency of abstract states. Features based on the frequency of abstract states capture the number of times each abstract state is visited within an episode, providing a richer representation of the agent's behavior. An example of a transformed episode is as follows, where the episode is characterized by the number of occurrences of each abstract state:

$$\begin{array}{rccccccc} & S_1^\phi & S_2^\phi & \cdots & S_j^\phi & \cdots & S_{n-1}^\phi & S_n^\phi \\ episode_i(t) & 0 & 1 & \cdots & 3 & \cdots & 2 & 0 \end{array}$$

Using this frequency-based feature representation, we evaluated *SMARLA*'s performance

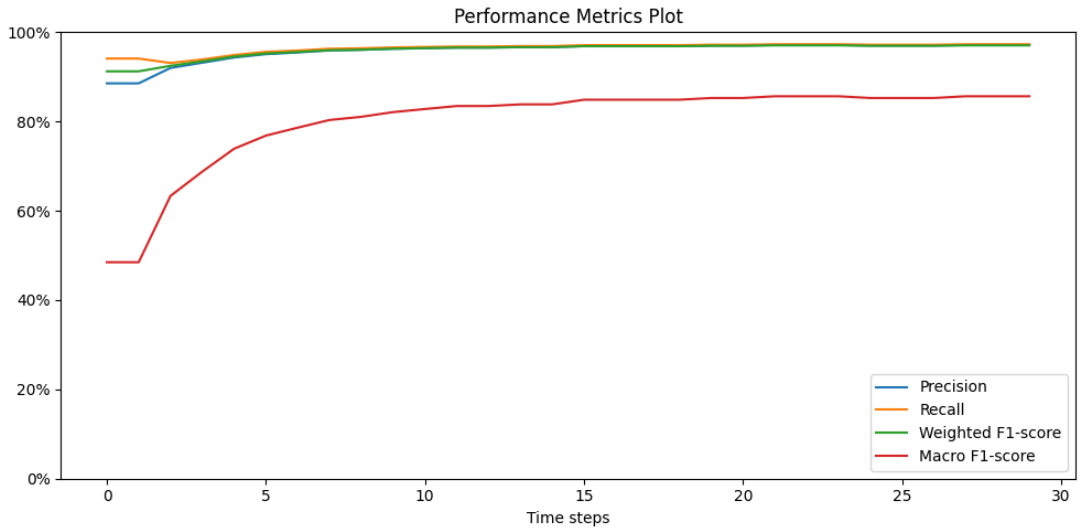

Figure 3.7: Performance of the safety violation prediction models in the *Highway* case study

on the same test dataset of 1000 episodes for the *Highway Driving* case study. The results are depicted in Figure 3.8

Relying on frequency-based features in the *Highway Driving* case study we achieved a precision of 96.2%, a recall of 96.3%, a weighted F1-score of 96.3% and an F1-score of 82.9%, after time step 10. By time step 20, the F1-score of *SMARLA* reach 91.9% while the weighted F1-score, precision and recall all reach 98.2%. Also, at this time step, on average 30% of the episodes remain to execute until reaching an unsafe state.

While binary features excel at early-stage predictions, the frequency of abstract state visits allows for improved prediction accuracy as more data accumulates (i.e., in later time steps when states are repeated). This is particularly useful in longer or more complex episodes, where additional data helps refine the model's predictions.

Answer to RQ1: *SMARLA* demonstrated high accuracy in predicting safety violations from RL agents. Moreover, such accurate predictions can be obtained early during the execution of episodes, thus enabling the system to prevent or mitigate damages.

3.4.4.2 RQ2. How can the safety monitor determine when to trust the prediction of safety violations?

In this research question, we investigate the use of confidence intervals as a means for the safety monitor to determine the appropriate time step to trigger safety mechanisms. This investigation is based on the same set of episodes randomly generated for RQ1 (Section 3.4.4.1). At each time step t, we collect the predicted probability of safety violation $P_{e_i}(t)$ and the corresponding confidence interval $[Low(t), Up(t)]$. The lower bound

Decision criteria	Case study	Decision time step			Remaining time steps			Remaining % of time steps			# FP
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	
Upper bound	Mountain-Car	15	20.96	38	57	74.21	80	59%	77.16%	83%	2
	Cart-Pole	22	41.36	105	67	104.5	164	38.70%	71.35%	82%	11
	Highway Driving	2	4.02	15	1	3.07	11	9.09%	44.60%	66.67%	46
	Highway Driving *	2	6.15	20	1	3.96	11	5.56%	43.79%	66.67%	67
Output probability	Mountain-Car	20	32.53	63	32	62.65	75	33.33%	65.15%	78.12%	1
	Cart-Pole	30	48.8	126	46	96.48	129	27%	66.58%	78.60%	2
	Highway Driving	2	4.16	15	1	2.92	11	9.09%	42.81%	66.67%	40
	Highway Driving *	2	6.71	22	1	3.55	11	5.56%	37.69%	64.28%	51
Lower bound	Mountain-Car	25	44.39	76	19	50.79	70	19.79%	52.81%	72.92%	1
	Cart-Pole	33	56.94	171	8	83.36	123	0.49%	61.34%	73.33%	0
	Highway Driving	2	3.6	10	1	2.96	11	9.09%	45.13%	66.67%	38
	Highway Driving *	3	6.40	20	1	2.93	10	11.11%	31.81%	57.14%	39

Table 3.1: Overview of *SMARLA*'s decision times and the remaining percentage of time steps before safety violations occur across different decision criteria and case studies (# FP stands for the total number of false positives and *Highway Driving** refers to *Highway Driving* case study using frequency-based features)

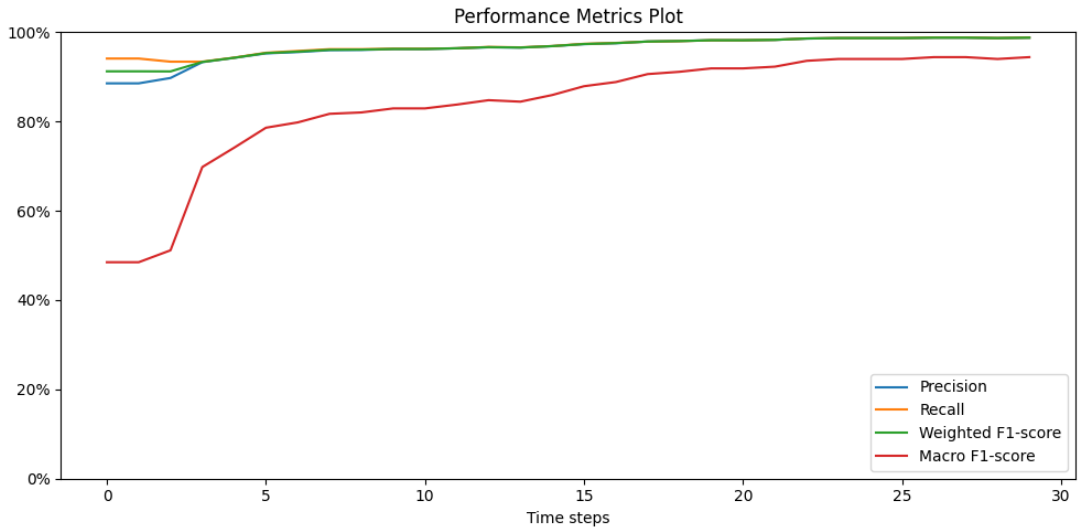

Figure 3.8: Performance of the safety violation prediction models in the *Highway* case study with frequency-based features

($Low(t)$) and upper bound ($Up(t)$) of the confidence interval are computed using the methodology detailed in Section 3.3.3.2.

To determine the best decision criterion for triggering safety mechanisms, we considered and compared the following alternative criteria:

- If the probability of safety violation, $P_{e_i}(t)$, is equal to or greater than 50%, then the safety mechanism is activated.
- If the upper bound of the confidence interval at time step t (based on the confidence level of 95%) is above 50% (i.e., $Up(t) \geq 50\%$), then the safety mechanism is activated. This is a conservative approach as the actual probability has a 97.5% chance to be below that value. This may result in many false positives but it leads to early predictions of unsafe episodes and is unlikely to miss any unsafe episodes.
- If the lower bound of the confidence intervals at time step t is above 50% (i.e., $Low(t) \geq 50\%$), then the safety mechanism is activated. In this criterion, the actual probability has only a 2.5% chance to be below that bound and we thus minimize the occurrence of false positives, at the cost of relatively late detection of unsafe episodes and more false negatives.

Decision criteria identify the time step when the execution should be stopped and safety mechanisms should be activated. However, note that during our test, we continue the execution of the episodes until termination in order to extract the number of time steps until termination and the true label of episodes for our analysis.

In Figure 3.9, we provide representative examples of the estimated probability of safety violation over time for a safe episode, an unsafe episode, and a mispredicted episode (false

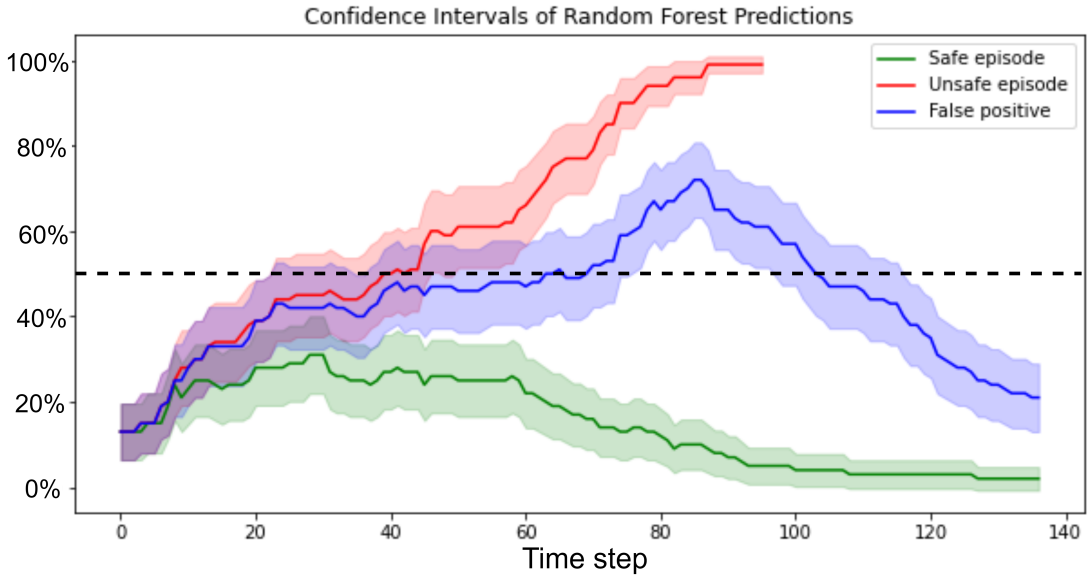

Figure 3.9: Confidence intervals of a safe episode (in green), an unsafe episode (in red) and a false positive episode (in blue) in the *Mountain-Car* case study

positive) in the *Mountain-Car* case study. The curves in the graph represent the output probabilities generated by the safety violation prediction model, while the shaded areas depict the corresponding 95% confidence intervals. We observe that, as more information is acquired, probabilities either tend to increase or decrease depending on whether the episode is safe or unsafe. At some point, the confidence intervals tend to narrow down as enough information from episodes is collected to make an accurate prediction. As expected, in the figure, relying on the upper bound of the confidence interval leads to a much earlier safety violation prediction in the unsafe episode (depicted in red in the figure), compared to when considering the predicted output probability or the lower bound of the confidence interval.

Now the question is how do the predictions based on the three above criteria compare in terms of accuracy. Figures 3.10, 3.11 and 3.12 present a comparison of the F1-scores of the three predictions at each time step for the *Mountain-Car*, *Cart-Pole* and *Highway Driving* case studies, respectively. Additionally, Figure 3.13 presents the comparison of decision criteria for the *Highway Driving* case study, using frequency-based features. Though there are differences in the magnitude of the trend, results from our case studies consistently show that using the upper bound is the best choice as it leads to early accurate predictions.

More in detail, we observe that in the *Cart-Pole* case study, the F1-score based on the upper bound is above 95% from time step 50 and plateaus around 99% at time step 102. On the other hand, when using the output probability of the safety violation prediction model, the F1-score is above 95% starting from time step 59 and reaches its plateau of 99% at time step 135. With the lower bound, however, F1-score is above 95% from time step 99 and reaches a plateau of 98% at time step 150. Also, for the *Mountain-Car* case study, with the upper bound we reach an F1-score above 95% from time step 23 and plateau

around 99% at time step 38. In contrast, the F1-score based on the output probability of the safety violation prediction model is above 95% from time step 41 and plateaus around 99% at time step 50. Finally, with the lower bound, the F1-score is above 95% from time step 46, and plateaus around 99% at time step 69.

In the Highway Driving case study, using the upper bound as a decision criterion allowed us to achieve an F1-score that plateaus at 85% by time step 12. This early stabilization underscores the robustness of the upper bound criterion in rapidly identifying safety violations. Similarly, when using the output probability of the safety violation prediction model, we observed that the F1-score exceeds 82% by time step 10 and reaches a plateau around 85% by time step 15, demonstrating close and effective results. However, when applying the lower bound criterion, the F1-score stabilizes later, plateauing at 83% by time step 21. This delay illustrates a more conservative approach, which, while reducing false positives, may also delay the prediction of safety violations.

In the Highway Driving case study using frequency-based features, applying the upper bound as a decision criterion resulted in an F1-score of 82% by time step 10. Achieving high accuracy predictions early on time highlights our ability to quickly and effectively detect safety violations. However, this accuracy continues to improve over time, reaching an F1-score of 90% and 94% at time step 18 and time step 25, respectively. Using the output probability of the safety violation prediction model yields similar performance to the upper bound, with the F1-score at 82% by time step 10, increasing to 91% by time step 18, demonstrating comparable effectiveness. In contrast, the lower bound criterion led to a slower stabilization, with the F1-score reaching 85% by time step 15 and achieving an F1-score of 90% by time step 23.

We also observe that, for all three decision criteria in each case study, the F1-score is low during early steps due to lack of information. It then increases sharply (around time steps 30-40 for *Cart-Pole*, 15-45 for *Mountain-Car* and 2-5 for *Highway Driving*) to reach a plateau shortly after 50 time steps for both *Cart-Pole* and *Mountain-Car*, while it plateaus after 15 time steps for the *Highway Driving* case study. During that time frame, the difference among decision criteria for *Mountain-Car* is more significant than for *Cart-Pole* and *Highway Driving*. This discrepancy is due to the wider confidence intervals present in the *Mountain-Car* case study, which can be explained by wider variance in decision tree predictions. Furthermore, the high F1-scores for all three decision criteria indicate that the confidence intervals significantly narrow once the plateau is reached. At that point, there is no overlap between the confidence intervals of unsafe and safe episodes.

Using these observations, we computed the average improvements, in terms of time steps required to achieve peak performance, when predicting safety violations by considering the upper bound of the confidence interval, in contrast to (1) the output probability and (2) the lower bound of the confidence interval. Our findings indicate that using the upper bound of the confidence intervals results in an average decrease of 24% for both *Cart-Pole* and *Mountain-Car*, and 20% for *Highway Driving* in terms of time steps required to achieve peak performance, as compared to using the predicted probability. When compared to using the lower bound, the decrease is 32% for *Cart-Pole*, 45% for *Mountain-Car* and 43% for *Highway Driving*.

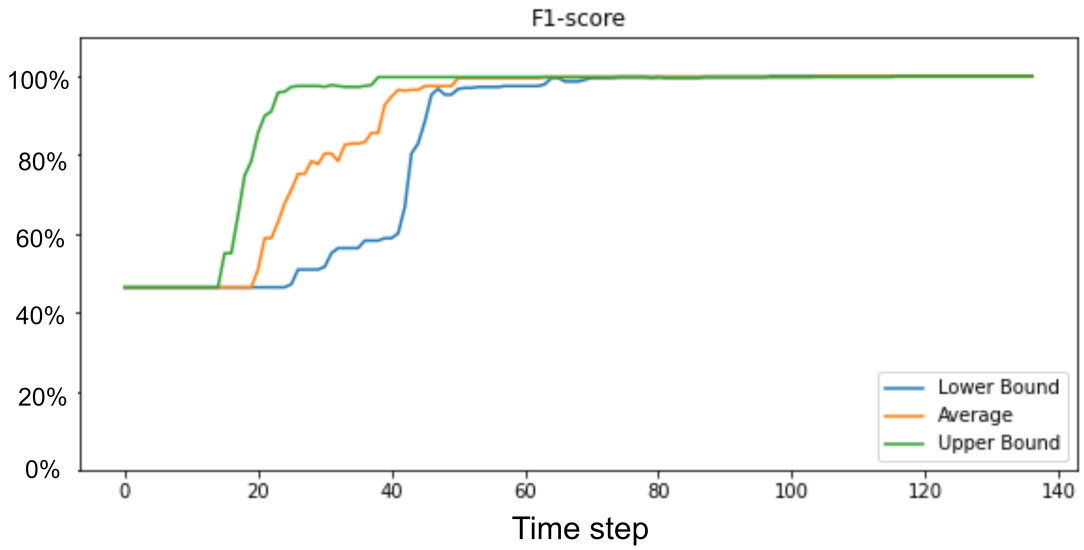

Figure 3.10: F1-score of the safety violation prediction model for different decision criteria in the *Mountain-Car* case study

Regarding *Highway Driving* with frequency-based features, we observed similar performance in terms of prediction time steps when considering the upper bound and prediction probability. However, there was a 32% improvement compared to the lower bound.

We also investigated in all case studies three important metrics: (1) the decision time step, (2) the remaining time steps until the occurrence of a safety violation, and (3) the remaining percentage of time steps to execute until violation. For each metric, we present in Table 3.1 the minimum, maximum, and average values. The result in Table 3.1 suggests that, relying on the upper bound, the average decision time step in the *Mountain-Car* case study is 21. However, in the best-case scenario, safety violations are predicted as early as time step 15, while in the worst-case scenario, such violations are predicted at time step 38. Notably, the results demonstrate that when safety mechanisms are triggered, on average 74 times steps (77%) of episodes remain to be executed. This observation suggests there is ample time to initiate safety mechanisms and hopefully prevent safety violations.

Also, in the *Cart-Pole* case study, employing the upper bound, the average prediction time step is determined to be 41. The earliest prediction of a safety violation occurs at time step 22, while the latest prediction is observed at time step 105. Remarkably, on average, there are still 104 time steps remaining to be executed when safety mechanisms are triggered. This accounts for approximately 71% of the average length of episodes, once again suggesting there is significant time available for carrying out safety measures. Similarly, in the *Highway Driving* case study, when using the upper bound of confidence intervals, the average prediction time step is 4, the earliest prediction of a safety violation occurs at time step 2 while the latest prediction is made at time step 15. Also, at the time of prediction, 44.6% of the average length of episodes still remains to be executed. Regarding the Highway Driving case study using frequency of abstract states as features, with the upper bound of confidence intervals, the average prediction time step is 6, the earliest

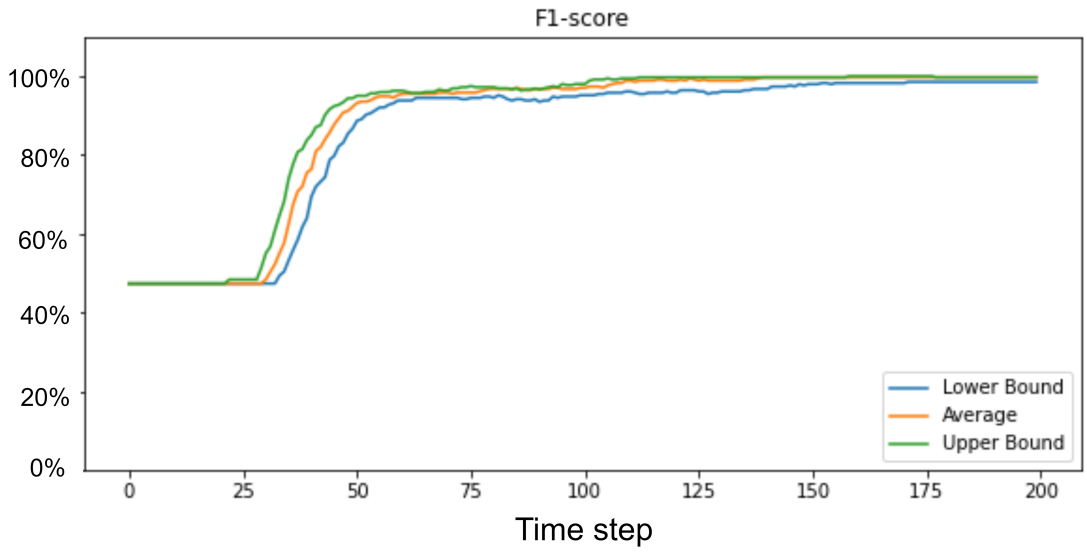

Figure 3.11: F1-score of the safety violation prediction model for different decision criteria in the *Cart-Pole* case study

prediction of a safety violation occurs at time step 2 while the latest prediction is made at time step 20. Also, at the time of prediction, 43.8% of the average length of episodes still remains to be executed. Overall, the above results demonstrate that *SMARLA* can predict safety violations early and accurately, suggesting it is a good solution for ensuring system safety when relying on RL agents.

Furthermore, in the *Cart-Pole* case study, our observations revealed the occurrence of 11 false positives leading to a false positive rate (FPR) of 1%, when relying on the upper bound. In contrast, only two false positives were identified with the output probability ($FPR = 0.2\%$), and none when employing the lower bound. Regarding the *Mountain-Car* case study, two false positives were observed ($FPR = 0.2\%$) when employing the upper bound. In contrast, using the output probability or the lower bound resulted in only one false positive ($FPR = 0.1\%$).

Finally, in the *Highway Driving* case study, we obtained 46 false positives leading to an FPR of 4.8% when using the upper bound and 40 false positives with an FPR of 4.2% considering the output probability. Furthermore, we identified 38 false positives ($FPR = 4\%$), when using the lower bound. Similarly, when using frequency-based features, we observed 67 false positives, resulting in an FPR of 7.1% with the upper bound, and 51 false positives with an FPR of 5.4% when considering the output probability. Additionally, we identified 39 false positives ($FPR = 4.1\%$) when applying the lower bound.

To summarize, while relying on the upper bound of confidence intervals leads to an earlier prediction of safety violations, it also introduces a higher rate of false positives compared to using the predicted probability and the lower bound. Therefore, considering the trade-off between earlier detection of safety violations and the number of false positives, the selection of an appropriate decision criterion relies heavily on the level of criticality of the RL agent. For instance, in certain scenarios, prioritizing early detection of safety

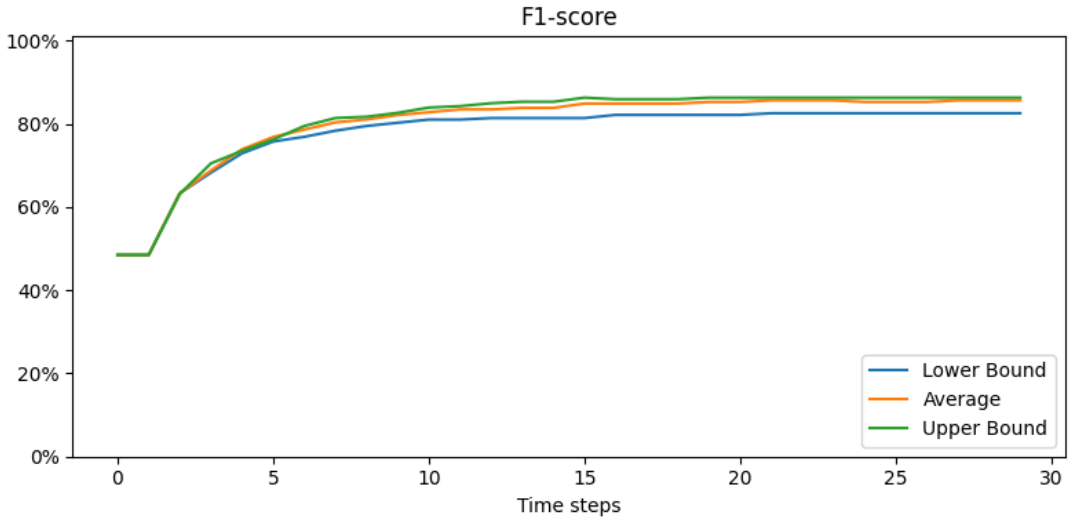

Figure 3.12: F1-score of the safety violation prediction model for different decision criteria in the *Highway* case study

violations and allowing for a longer time frame to apply safety mechanisms may be of critical importance, even at the expense of a slightly higher false positive rate. Conversely, in other cases, there might be a preference to sacrifice time in order to optimize accuracy and minimize the occurrence of false positives. The selection of the appropriate decision criterion depends on the specific context and the relative importance of early detection and prediction accuracy. In our case studies, the increase in false positives appears to be limited, and therefore using the upper bound is the best option.

Answer to RQ2: Considering the upper bound of the confidence intervals leads to a significantly earlier and still highly accurate detection of safety violations. This provides a longer time frame for the system to apply preventive safety measures and mitigate potential damages. This, however, comes at the expense of a slightly higher false positive rate.

3.4.4.3 RQ3. What is the effect of the prediction threshold on the safety monitoring system?

We aim in this research question to study the impact of varying the prediction threshold values on our safety monitoring approach. We evaluate the prediction times and performance of *SMARLA* using different prediction thresholds based on the same set of episodes randomly generated in RQ1 (Section 3.4.4.1). Specifically, for each case study, we consider three prediction thresholds $\theta \in \{25\%, 50\%, 75\%\}$ and assess the performance of *SMARLA* according to the three decision criteria explained in Section 3.4.4.2 ($P(t) \geq \theta$, $Up(t) \geq \theta$, and $Low(t) \geq \theta$). For each case study and evaluation criteria, we report (1) the average decision time step, (2) the average remaining percentage of time steps to execute until

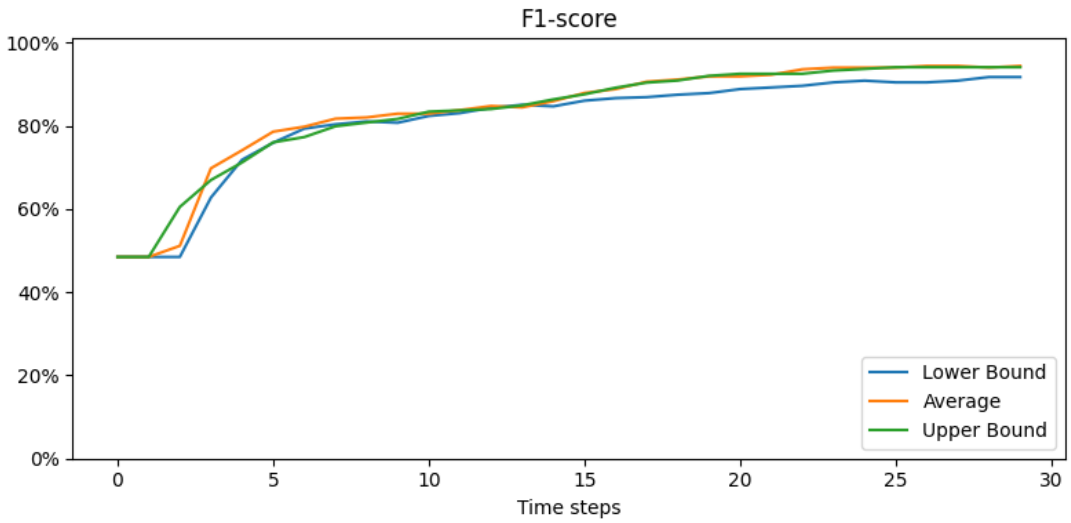

Figure 3.13: F1-score of the safety violation prediction model for different decision criteria in the *Highway* case study using frequency-based feature representation

violation, (3) the number of false positives of the safety monitor, and (4) the number of false negatives. Results are reported in Figures 3.14 to 3.25.

As visible from the figure, we obtain consistent results over the three decision criteria and case studies. As expected, varying the prediction threshold does impact the average prediction time step, the percentage of the remaining portion of the episode to be executed, as well as the number of false positives and false negatives. More precisely, having a lower prediction threshold, such as $\theta = 25\%$, leads to an earlier decision time to predict safety violations, which may be critical for taking preemptive safety measures, but increases the occurrence of false positives. On the other hand, higher thresholds, such as $\theta = 75\%$, reduce false positives at the expense of delayed safety violation detection. Thus, choosing a high threshold can potentially compromise the system's ability to promptly initiate safety mechanisms and effectively prevent safety violations. Finally, we found that the intermediate threshold $\theta = 50\%$ offers a balanced solution for all case studies, minimizing false positives and false negatives while still providing timely predictions. We therefore rely on such a threshold value in our experiments as it offers the best trade-off between accurate and early predictions of safety violations. We should note, however, that determining a suitable prediction threshold is context-dependent and relies heavily on the level of criticality of the RL agent. In practice, we recommend building a testing dataset that encompasses both safe and unsafe episodes based on random executions of the agent. Then, we recommend trying different prediction thresholds and conducting a sensitivity analysis specific to the deployment environment of the DRL agent, to determine an optimal threshold that offers the best trade-off between timely and accurate prediction of safety violations.

Answer to RQ3: The performance of safety monitoring is sensitive to the prediction threshold θ. Higher prediction thresholds reduce false positives but lead to a delayed prediction of safety violations and higher false negatives, whereas lower thresholds lead to earlier predictions and lower false negatives but at the cost of increased false positives. Therefore, selecting an optimal threshold is crucial to balance timely and accurate predictions, enhancing the system's effectiveness in predicting safety violations. Such selection is specific to the deployment environment of the DRL agent but a threshold value around $\theta = 50\%$ is likely to be balanced.

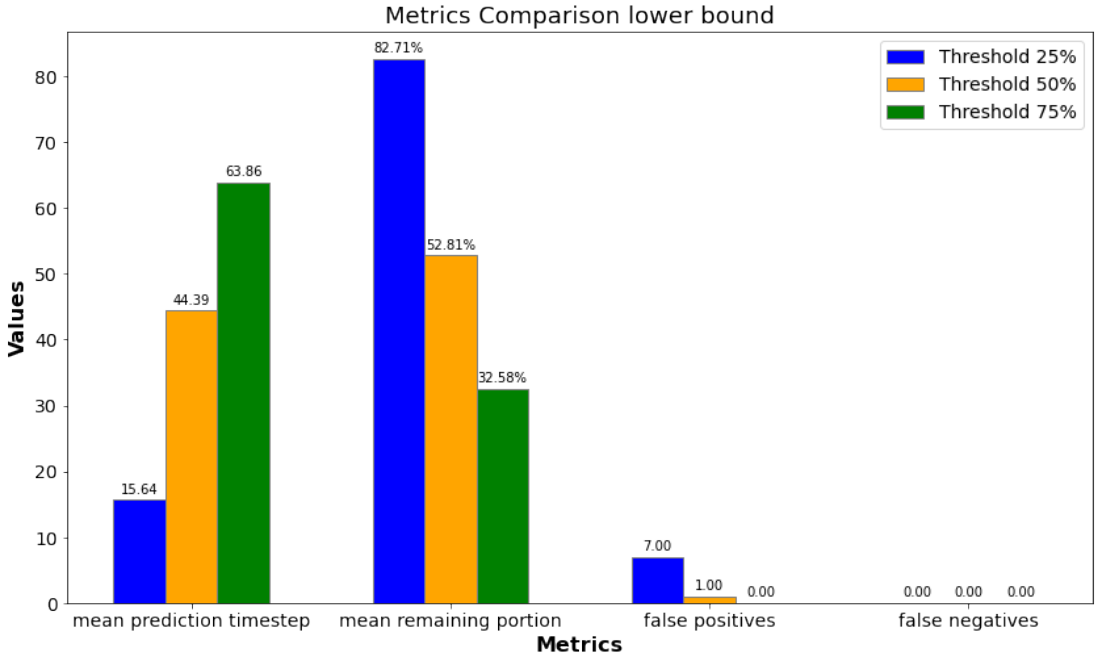

Figure 3.14: Comparison of 25%, 50% and 75% threshold values for *Mountain-Car* case study based on the lower bound

3.4.4.4 RQ4. What is the effect of the abstraction level on the safety monitoring system?

To answer this research question, we studied how different levels of state abstraction affect the performance of the safety violation prediction model in the training phase and in operation.

The accuracy of the *Random Forest* model after training with different abstraction levels. This aspect involves evaluating the performance of the *Random Forest* model once it has been trained on the available training data. We randomly sampled 70% of the dataset to train and 30% to compute the F1-scores of the models using different levels of abstraction (d). A lower abstraction level implies finer-grained states, while higher abstraction levels represent coarser ones that lead to a smaller feature space.

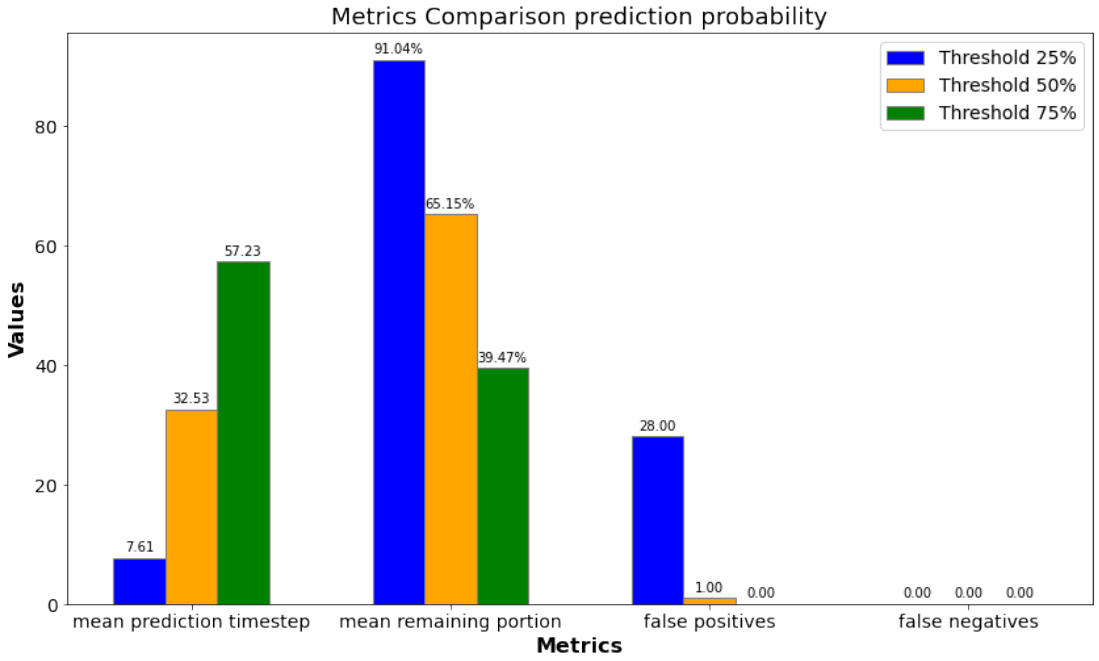

Figure 3.15: Comparison of 25%, 50% and 75% threshold values for *Mountain-Car* case study based on the prediction probability

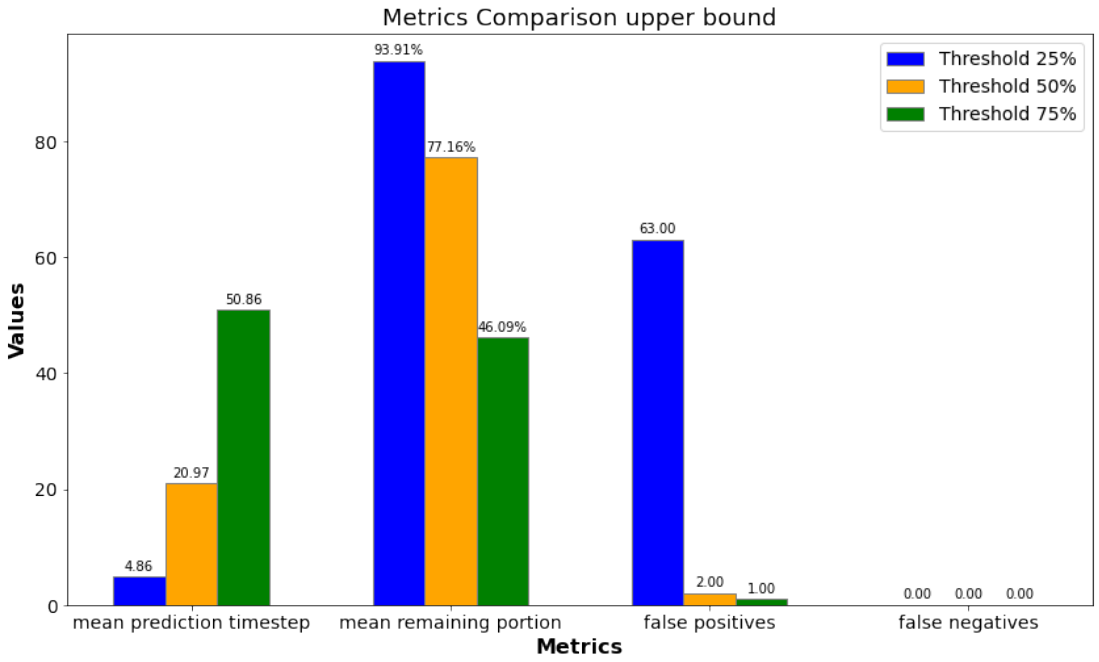

Figure 3.16: Comparison of 25%, 50% and 75% threshold values for *Mountain-Car* case study based on the upper bound

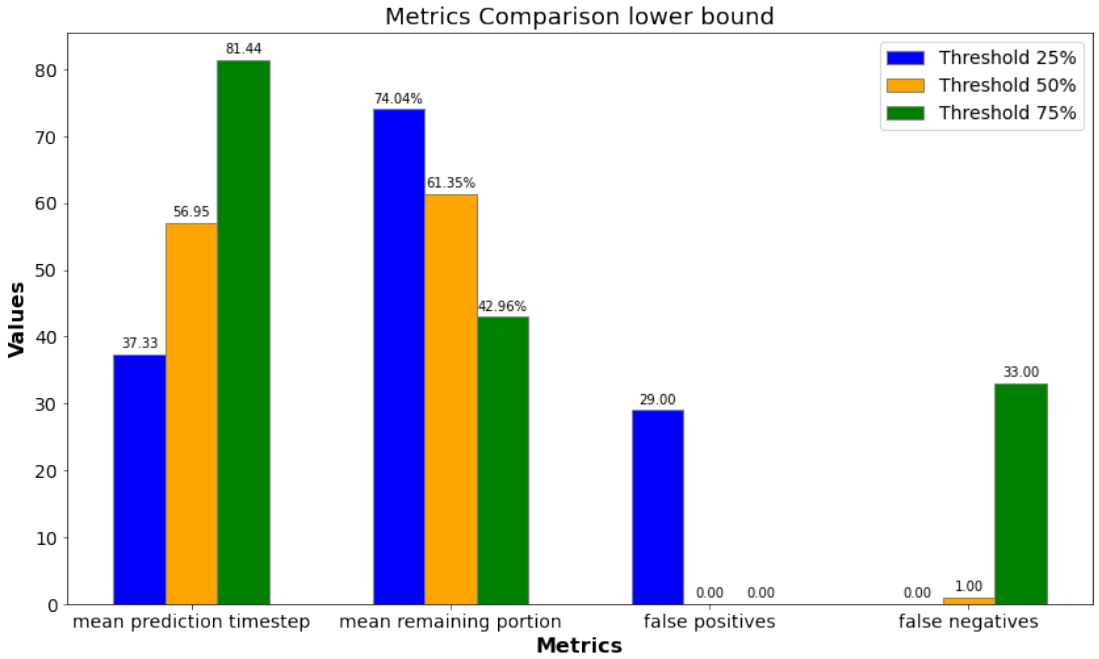

Figure 3.17: Comparison of 25%, 50% and 75% threshold values for *Cart-Pole* case study based on the lower bound

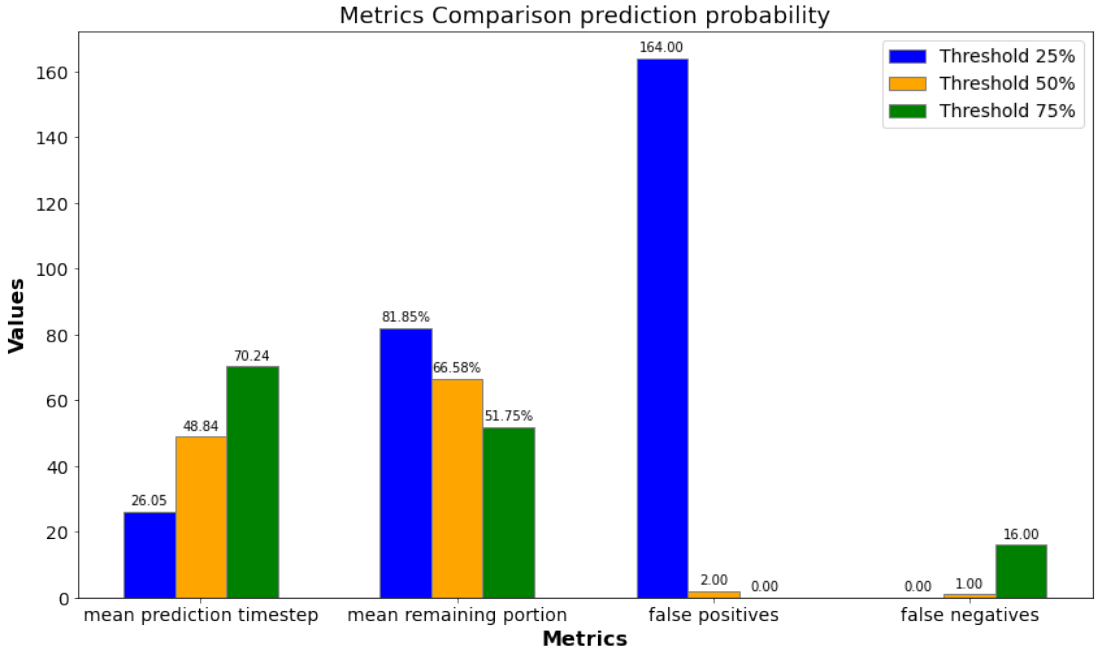

Figure 3.18: Comparison of 25%, 50% and 75% threshold values for *Cart-Pole* case study based on the prediction probability

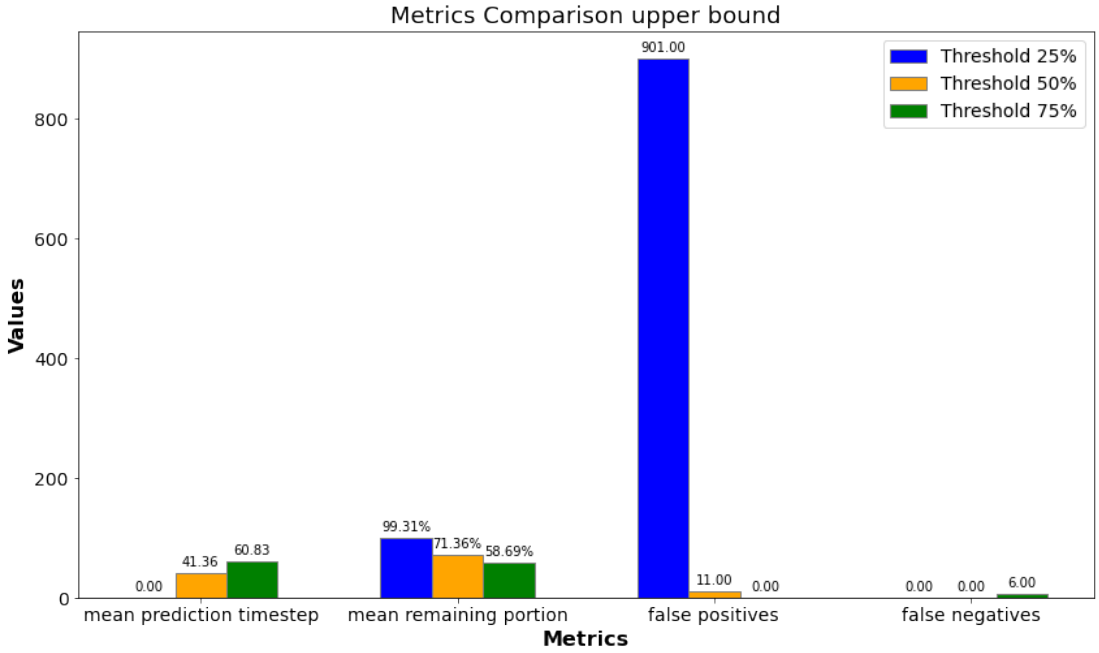

Figure 3.19: Comparison of 25%, 50% and 75% threshold values for *Cart-Pole* case study based on the upper bound

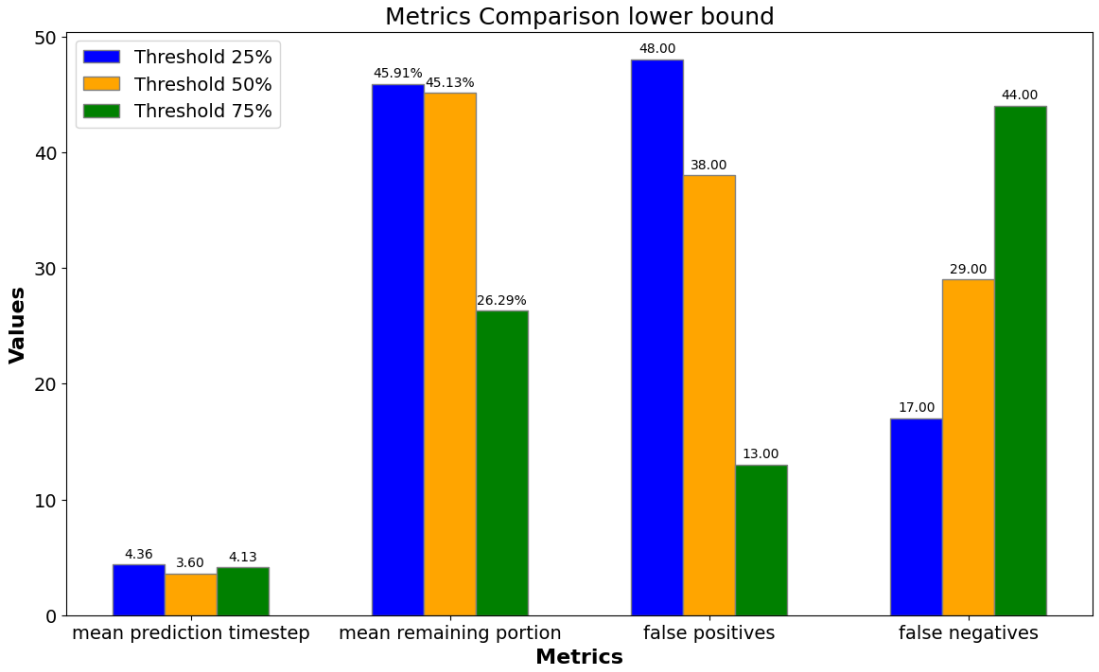

Figure 3.20: Comparison of 25%, 50% and 75% threshold values for *Highway Driving* case study based on the lower bound with binary features

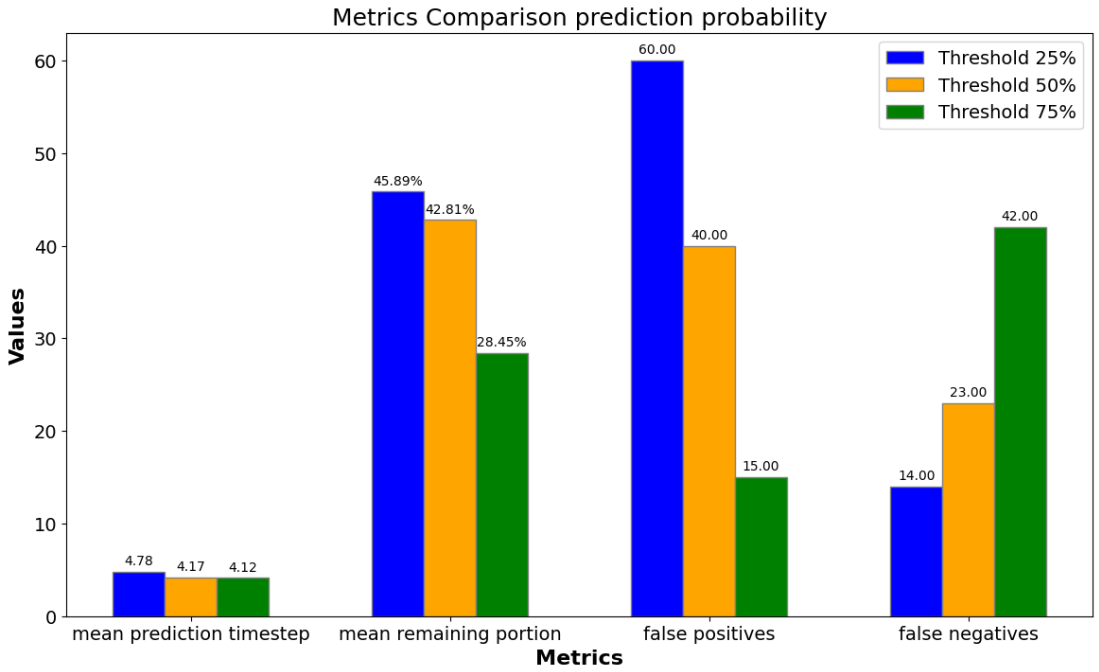

Figure 3.21: Comparison of 25%, 50% and 75% threshold values for *Highway Driving* case study based on the prediction probability with binary features

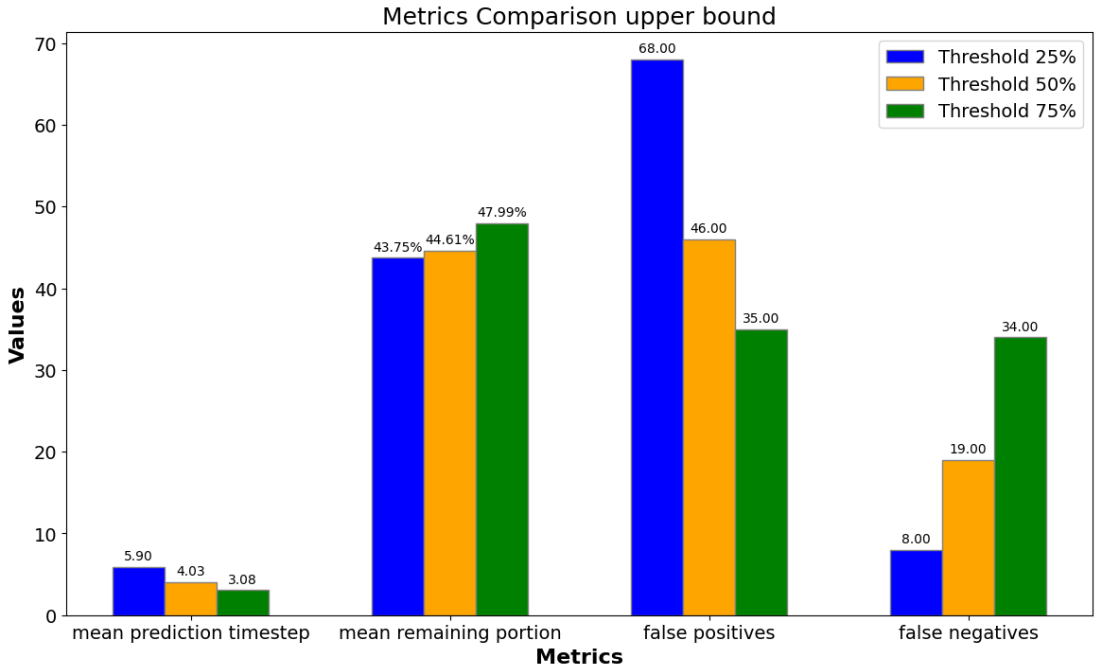

Figure 3.22: Comparison of 25%, 50% and 75% threshold values for *Highway Driving* case study based on the upper bound with binary features

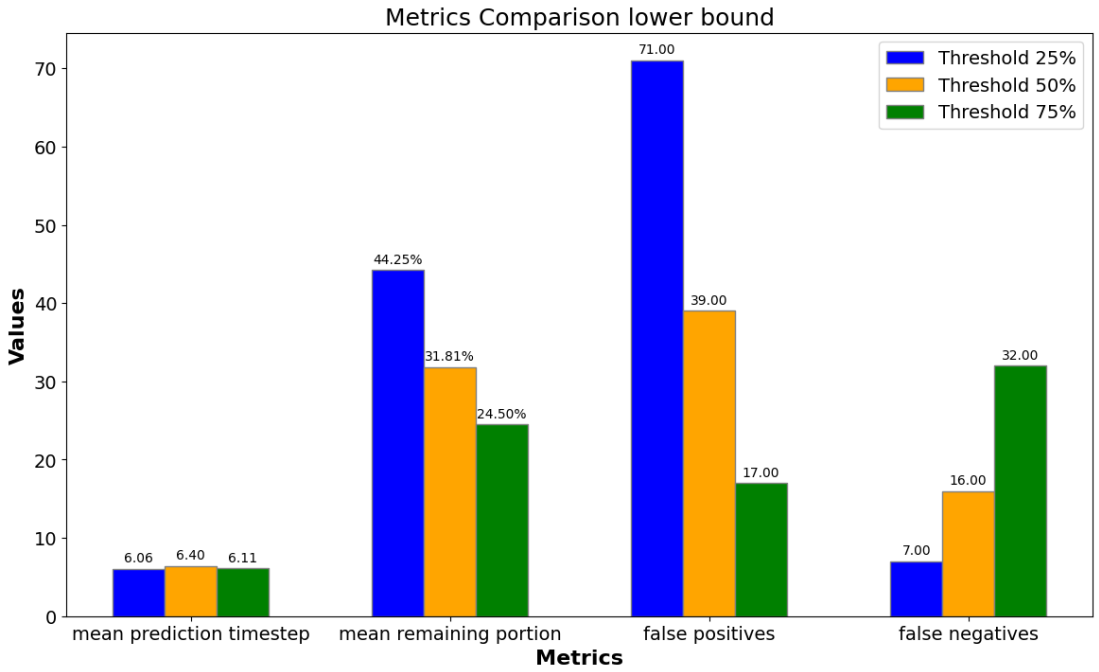

Figure 3.23: Comparison of 25%, 50% and 75% threshold values for *Highway Driving** case study based on the lower bound with frequency-based features

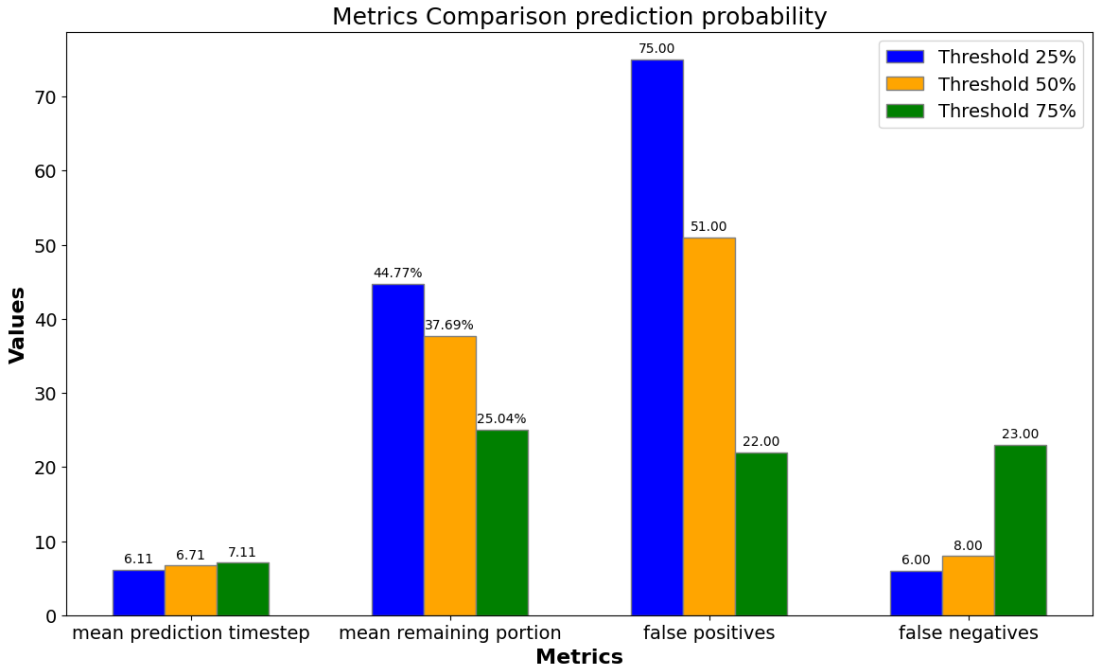

Figure 3.24: Comparison of 25%, 50% and 75% threshold values for *Highway Driving** case study based on the prediction probability with frequency-based features

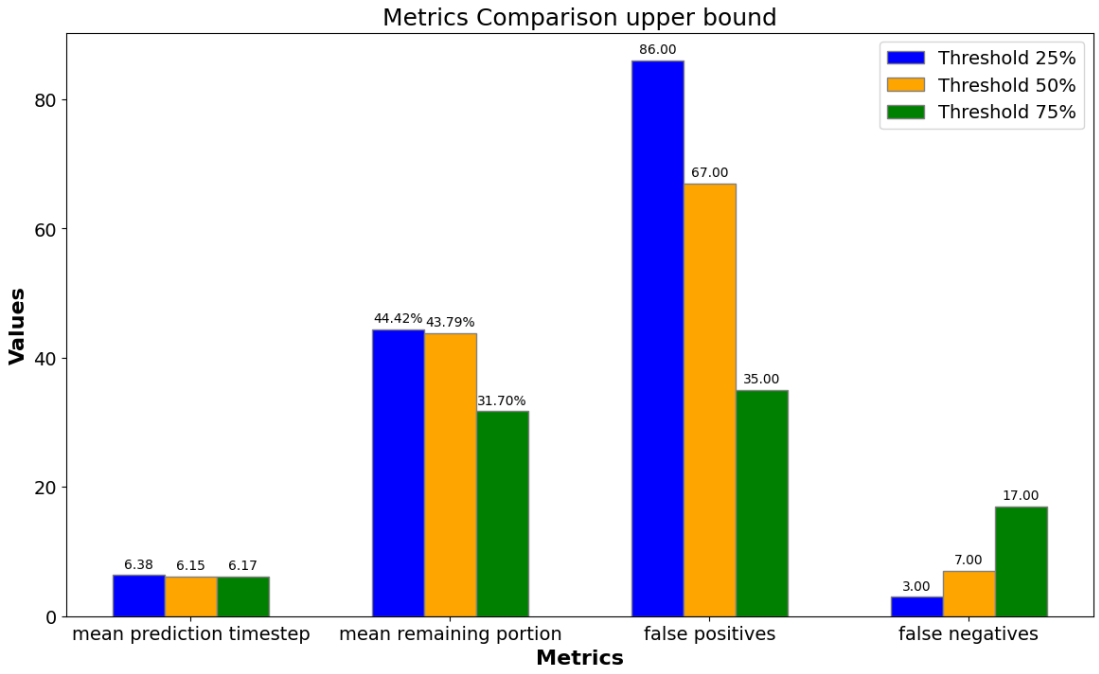

Figure 3.25: Comparison of 25%, 50% and 75% threshold values for *Highway Driving** case study based on the upper bound with frequency-based features

Based on the analysis depicted in Figures 3.26, 3.27, and 3.28 we observed that higher abstraction levels lead to lower accuracy in predicting safety violations, above a threshold of 0.3 for *Cart-Pole*, 1000 for *Mountain-Car* and 0.6 for the *Highway Driving* case study. This is attributed to the smaller feature space associated with higher levels of abstraction. Note, however, that this does not apply to features based on the frequency of abstract states. As the results suggest (Figure 3.29), these features yield higher accuracy even at higher abstraction levels. This is because these features are more informative, even when the number of abstract states is small. As the abstraction level decreases, the feature space grows larger, allowing for more precise information to be captured by features. Consequently, the accuracy of the safety violation prediction model tends to increase until it eventually plateaus and then starts to decrease. This decrease occurs due to the very large number of abstract states, making it more challenging to learn in a larger feature space. This suggests that there is an optimal range of abstraction that yields the highest accuracy in predicting safety violations. Going beyond this optimal range can reduce the performance of safety monitoring.

In the *Highway Driving* case study using the frequency of abstract states, we observed that very high levels of abstraction ($d \geq 0.9$) achieve high accuracy but are not practical. This is because the probability of safety violations tends to consistently decrease, especially for safe episode. Specifically, at the start of the episode, the model predicts a high probability of safety violations (above the prediction threshold). As the safety monitor gathers more information, this probability drops in safe episodes, allowing the model to distinguish between safe and unsafe situations. However, this decreasing trend makes it difficult to predict safety violations and stop execution at early time steps during the execution,

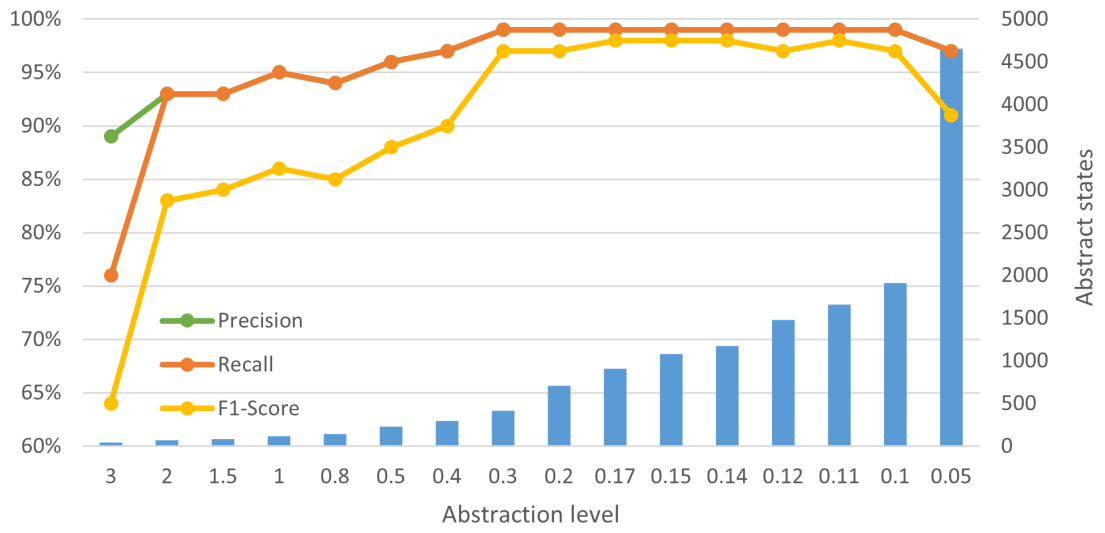

Figure 3.26: Precision, recall, and F1-score achieved after the training of the safety violation prediction model and the number of abstract states for the *Cart-Pole* case study across different abstraction levels

since the probability is above the prediction threshold. To address this, we introduced an additional criterion for choosing abstraction levels with frequency-based features, ensuring that the initial probability of safety violation remains below the prediction threshold.

This optimal range of abstraction level depends on the environment, the RL agent, and the reward. This arises from the fact that the calculation of Q-values, which are used in the abstraction process, relies heavily on the reward signal. Consequently, the choice of reward function significantly influences the optimal range of abstraction levels. Indeed, our empirical analysis revealed that abstraction levels ranging from 0.1 to 0.3 result in the highest accuracy for *Cart-Pole*. Similarly, the highest accuracy for *Mountain-Car*, *Highway Driving* with binary features, and *Highway Driving* with frequency features is obtained with abstraction levels from 0.5 to 1000, 0.06 to 0.6, and 0.2 to 0.8, respectively. Therefore, for the next experiments, we consider the abstraction levels within the optimal ranges of each case study.

The performance of the model in operation with different abstraction levels. This part focuses on evaluating the performance of the safety violation prediction model during the execution of episodes. We analyze how well the trained *Random Forest* models perform in operation across different time steps while considering different abstraction levels. The main focus for such evaluation is the model's ability to accurately predict safety violations early.

The F1-score of the safety monitoring models for the across all case studies, considering various levels of abstraction, are presented in Figures 3.30, 3.31, 3.32, and 3.33.

As visible, the performance of the safety violation prediction model is highly sensitive to the selected abstraction level during the training phase, especially in the case of *Mountain-Car*. Despite selecting only abstraction levels that maximize the model's per-

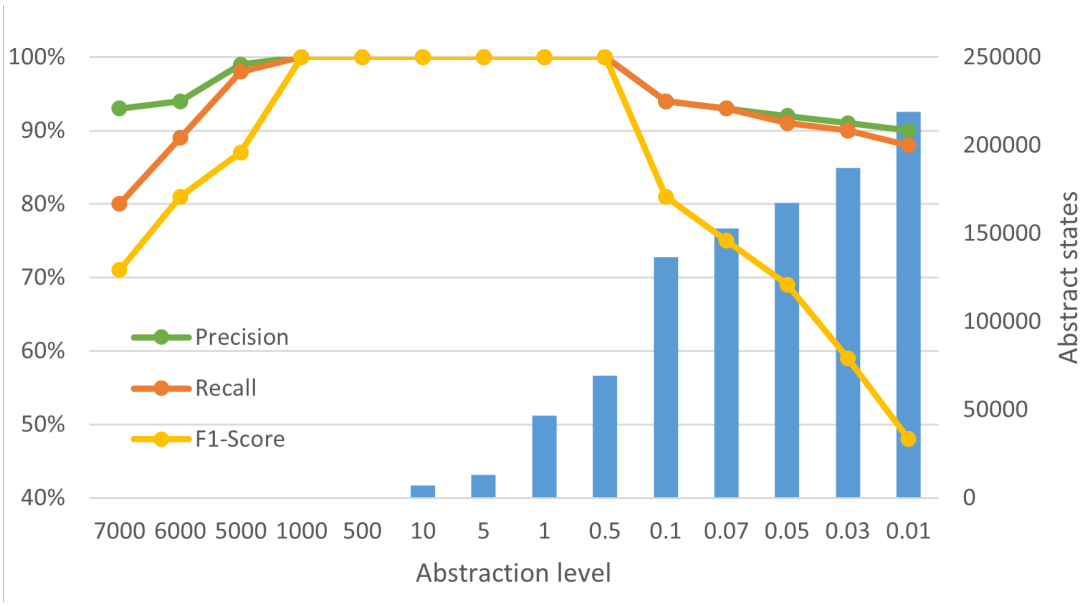

Figure 3.27: Precision, recall, and F1-score achieved after the training of the safety violation prediction model and the number of abstract states for the *Mountain-Car* across different abstraction levels

formance during training, they required different numbers of time steps to achieve the highest accuracy in predicting safety violations. This sensitivity highlights the importance of carefully selecting the appropriate abstraction level for optimal model performance.

Based on Figures 3.30, 3.31, 3.32 and 3.33, we observe that the most suitable abstraction level is $d = 0.11$ for *Cart-Pole*, $d = 5$ for *Mountain-Car*, $d = 0.2$ for *Highway Driving*, and $d = 0.4$ for *Highway Driving* using frequency based features, as they exhibit the most accurate and earliest prediction of safety violations compared to other abstraction levels. This indicates that these abstraction levels are particularly effective at capturing relevant features at the right level of granularity to support learning and the prediction of unsafe episodes.

In summary, in this research question, we studied the performance of *SMARLA* through a systematic two-step process. In the first step, we analyzed the accuracy of *SMARLA* in predicting safety violations post-training, and we derived a range of optimal abstraction levels corresponding to the highest F1-score. Next, we investigated the number of time steps *SMARLA* requires to achieve its peak accuracy. Note that, the ultimate objective is to identify the proper abstraction level which enables the safety monitor to reach its highest accuracy in predicting safety violations at the earliest time step possible.

Answer to RQ4: The accuracy of safety violation prediction models is sensitive to the selected abstraction level and, therefore, the latter should be carefully selected to have optimal monitoring results.

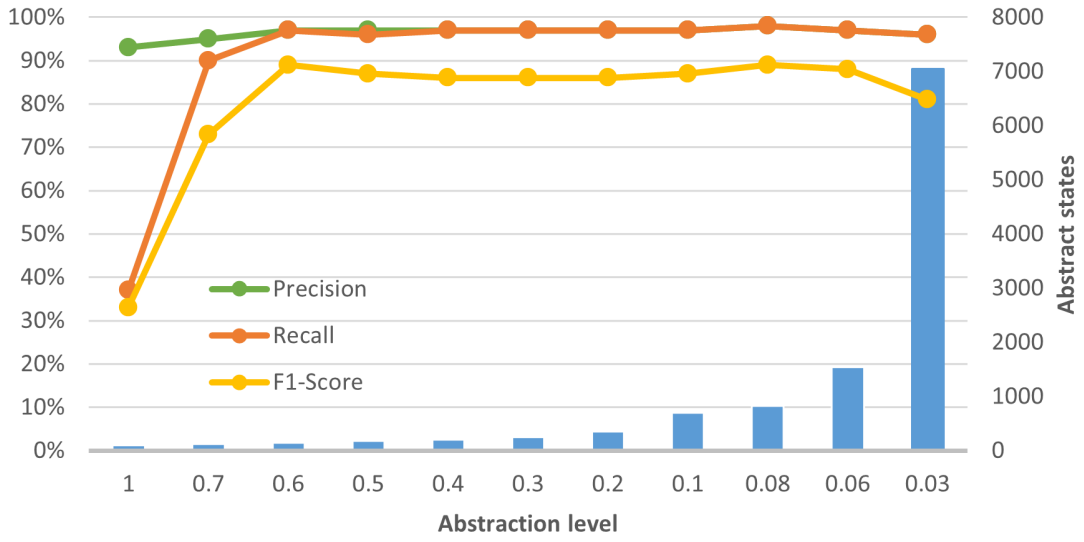

Figure 3.28: Precision, recall, and F1-score achieved after the training of the safety violation prediction model and the number of abstract states for the *Highway Driving* across different abstraction levels

3.4.4.5 Abstraction level selection procedure

To achieve early and accurate predictions of the safety monitor in practice, it is crucial to determine a proper abstraction level. Thus, based on the results of RQ4 3.4.4.4, we propose the following procedure to select a proper abstraction level for our monitoring approach. This procedure is a one-time process that involves a systematic two-step approach. Initially, in the first step, a Coarse-to-Fine search technique [113] is used for finding the optimal range of abstraction levels that maximize the F1-score of the safety violation prediction model. Subsequently, in the second step, within this optimal range, the abstraction level yielding the earliest prediction of safety violation is identified.

In detail, this process starts by training the safety violation prediction models with approximately 70% of the training episodes. During this phase, the safety violation prediction model is trained using a diverse range of abstraction levels, systematically varied in increments. The assessment metric is the F1-score of the safety violation prediction model. It is noteworthy that abstraction levels can be mapped to the number of abstract states, thereby facilitating the determination of the range of abstraction levels to be explored. As a practical guideline, using the Coarse-to-Fine approach, it is recommended to start with a wide range spanning from several hundred to approximately 100,000 states. The process involves iteratively considering new values close to abstraction levels that previously yielded high F1-scores, and therefore gradually narrowing down the range to ultimately identify the optimal range of abstraction levels.

However, the choice of the range of levels to cover depends also on the complexity of the environment; more complex environments may necessitate a larger number of abstract states to ensure accurate prediction of safety violations. Subsequently, within the recom-

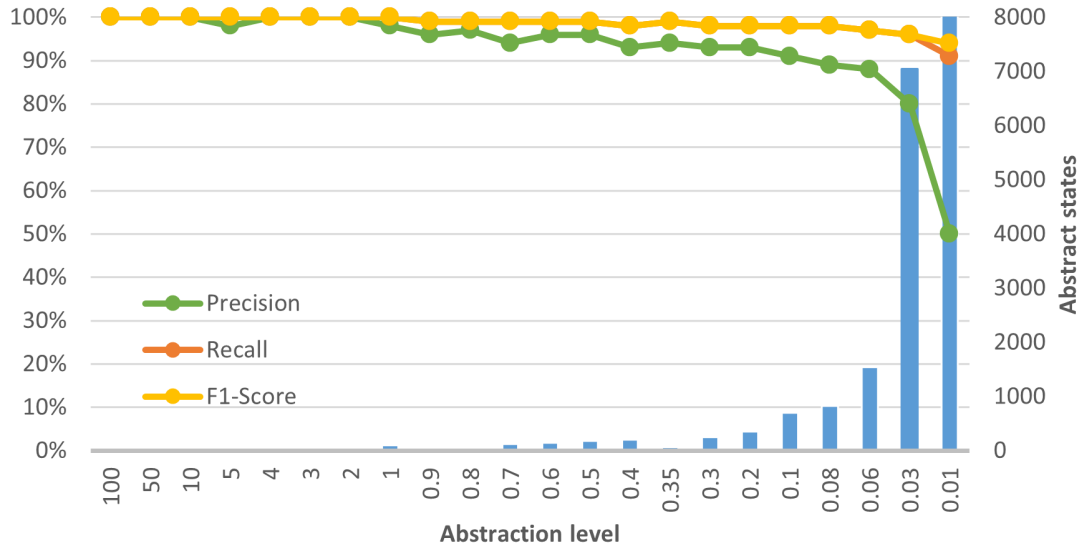

Figure 3.29: Precision, recall, and F1-score achieved after the training of the safety violation prediction model and the number of abstract states for *Highway Driving* across different abstraction levels using frequency of abstract states as features

mended range, the optimal subset of abstraction levels can be determined by selecting levels that maximize the F1-score of the models when evaluated on the remaining 30% of episodes. The results of this step for *Cart-Pole*, *Mountain-Car*, *Highway Driving*, as well as *Highway Driving* with frequency-based features are depicted in Figures 3.26, 3.27, 3.28 and 3.29, respectively. In the *Cart-Pole* case study, the optimal range of abstraction levels lies between 0.1 and 0.3, whereas for the *Mountain-Car*, it is between 0.5 and 1000, and for *Highway Driving* case study, it is between 0.6 and 0.06.

In the second phase, it is imperative to conduct experiments with different abstraction levels within the optimal range when utilizing the safety violation prediction model during the execution of the episode. The primary objective is to pinpoint the level of abstraction that enables the safety monitor to make precise predictions of safety violations at the earliest time step possible. Figures 3.30, 3.31 and 3.32 illustrate the outcomes of the second phase, where we extracted the F1-score of safety monitoring models considering various abstraction levels within the optimal range. As a result, the optimal abstraction levels that enable accurate and early predictions of the safety monitoring approach for the *Cart-Pole*, *Mountain-Car* and *Highway Driving* case studies are $d = 0.11$, $d = 5$ and $d = 0.2$, respectively.

3.4.4.6 Qualitative Analysis

To further shed light on the prediction results of *SMARLA*, we manually sampled representative episodes from each case study and qualitatively analyzed the corresponding risk plots. The primary objective of this analysis was to understand how the probability of

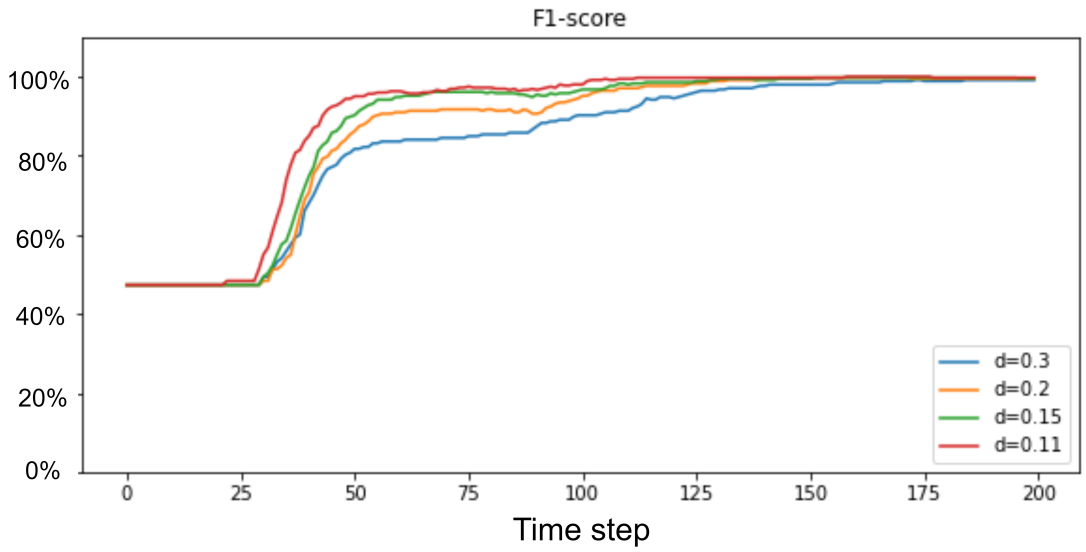

Figure 3.30: Performance of the safety violation prediction models in operation for the *Cart-Pole* case study across different levels of abstraction

safety violations evolves in relation to the dynamics of the agent in each scenario, providing deeper insights into the behavior of the agent and the conditions under which the risk of safety violation is increased. For example, in the *Cart-Pole* case study, we analyzed how the probability of safety violations evolves along with the position and velocity of the cart, as well as the angle and angular velocity of the pole. We provide an example of the safety violation prediction plot for a safe and an unsafe episode in Figure 3.34. This Figure illustrates how the probability of safety violation fluctuates with changes in the cart's dynamics. It is important to recall that an episode is considered unsafe if the cart moves away from the center by a distance above 2.4 units as it crosses the safe operational boundary and can cause damages, regardless of the accumulated reward (detailed in Section 3.4.2.1). As shown in Figure 3.34, the risk plot shows that in the safe episode, the probability of safety violations remains relatively low throughout the episode as the cart stays within the safe boundary. In contrast, in the unsafe episode, the probability of safety violations increases significantly as the cart approaches the safety boundary limit of 2.4. The velocity plot shows that in the safe episode, the velocity of the cart fluctuates within a moderate range, while in the unsafe episode, the cart's velocity reaches higher magnitudes (both positive and negative), indicating more pronounced movements. The angle and angular velocity plots further illustrate that in the unsafe episode, the angle of the pole deviates more significantly, and the angular velocity shows higher variations compared to the safe episode. More specifically the probability of safety violation increases when the angle of the pole is leaning toward the right and the cart is moving to the right to recover the pole from falling (time steps 40 to 50). This is clearly visible when comparing with another safe episode plotted in the same figure where, despite the pole starting to lean toward the right (time step 60), the agent immediately moves the cart to the right and successfully prevents the pole from falling in a much shorter distance from the reference point, so the probability of safety violations is decreased as depicted in Figure 3.34.

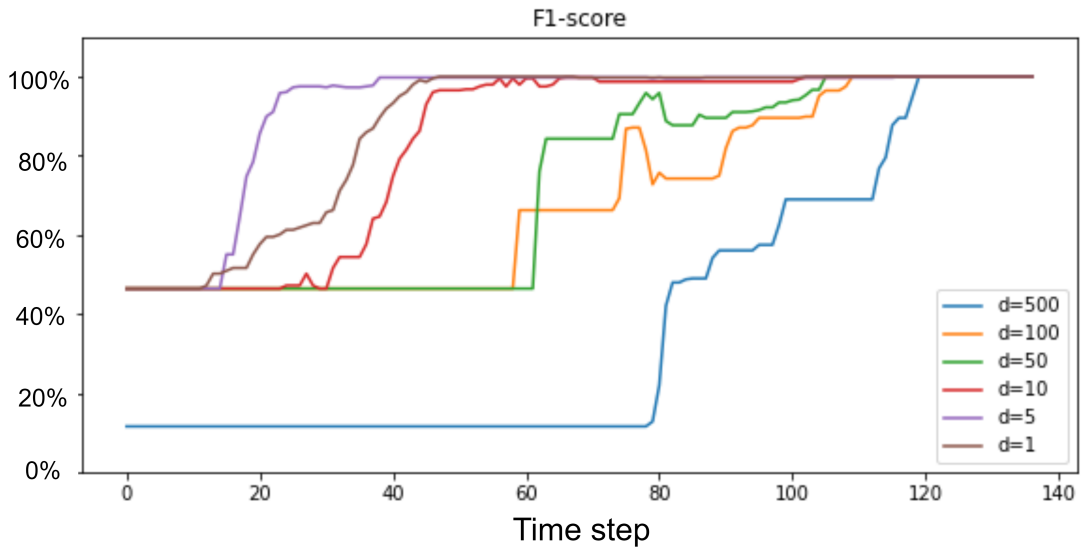

Figure 3.31: Performance of the safety violation prediction models in operation for the *Mountain-Car* case study across different levels of abstraction

Regarding the *Mountain-Car* case study, we also analyzed the probability of safety violations along with the position and velocity of the car. Examples of safe and an unsafe episodes is shown in Figure 3.35. We should recall that a safety violation happens when the car crosses the left border of the environment, as this poses potential damage to the car (detailed in section 3.4.2.2). In the unsafe episode, the agent initially gains momentum in the opposite direction compared to the safe episodes. The probability of safety violation significantly increases when the car is on the right slope, gaining momentum toward the left border. The momentum becomes so high that the car cannot stop before passing the left border. As the car gets closer to this boundary, the probability of safety violation continues to rise, where finally the car passes the left border and a safety violation occurs.

For the third case study, we analyzed the probability of safety violations in the *Highway Driving* scenario where the agent must navigate at high speeds while avoiding other vehicles. We should recall that, in this environment, a safety violation occurs when the agent collides with another vehicle (Section 3.4.2.3).

Figure 3.36 illustrates the probability of safety violation in an unsafe episode along with snapshots of the environment captured during the execution. In this episode, the agent starts driving in the third lane, where there is another actor vehicle at a safe distance from the ego vehicle. The agent decides to move to the rightmost lane to gain reward and increases speed to safely overtake the other vehicles (time steps 0 to 3). Then, another vehicle appears in the rightmost lane, cruising at a lower speed than the ego vehicle. At time step 12, when the ego vehicle is approaching the vehicle in front at high speed, the probability of safety violation begins to rise. At time step 14, the probability exceeds the 50% threshold, and we predict the safety violation as the ego vehicle continues to approach the vehicle in front. The execution continues until the final time step, where a collision occurs between the two vehicles.

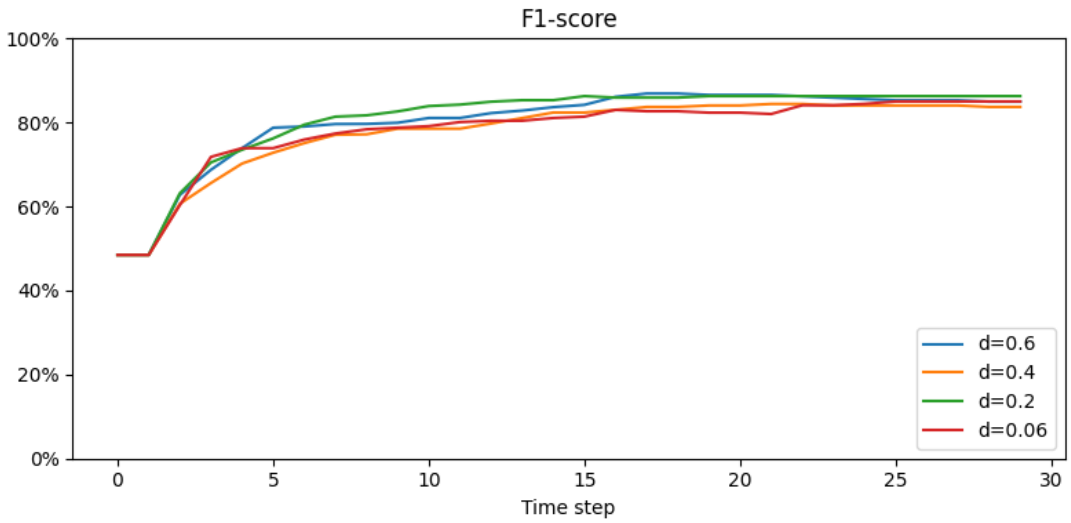

Figure 3.32: Performance of the safety violation prediction models in operation for the *Highway Driving* case study across different levels of abstraction

This qualitative analysis illustrates how *SMARLA* effectively predicts potential safety violations for DRL agents. By examining the agent's dynamics and the probability of safety violations, we provide insights into *SMARLA*'s capability to provide accurate early predictions of safety violations during the execution of episodes. Such early detection allows the system to take proactive measures to prevent or mitigate potential damages, thereby enhancing the overall safety and reliability of DRL agents.

Additionally, it demonstrates the practical application of SMARLA and qualitatively validates its prediction results. Through such a qualitative analysis one can understand the conditions under which the risk of safety violation is increased, and thus providing insights into how these risks might be mitigated in operation.

3.5 Discussion

We propose *SMARLA*, a black-box safety monitoring approach designed for reinforcement learning agents, and demonstrate its high accuracy in predicting safety violations of RL agents during their execution. We also show that such accurate prediction can be made early long before the actual occurrence of violations, allowing for timely incident prevention and mitigation. In our work, we rely on state abstraction to reduce state space and learn change patterns in agent Q-values to predict unsafe episodes. We should emphasize that learning the patterns of unsafe episodes with abstraction does not imply that all episodes following such patterns represent known unsafe concrete states. State abstraction allows us not to depend on the specific occurrence of concrete states and thus generalize patterns to concrete states that may not have been seen during training. Although no approach, including ours, can claim to fully predict or mitigate all unknown unsafe states, the use of

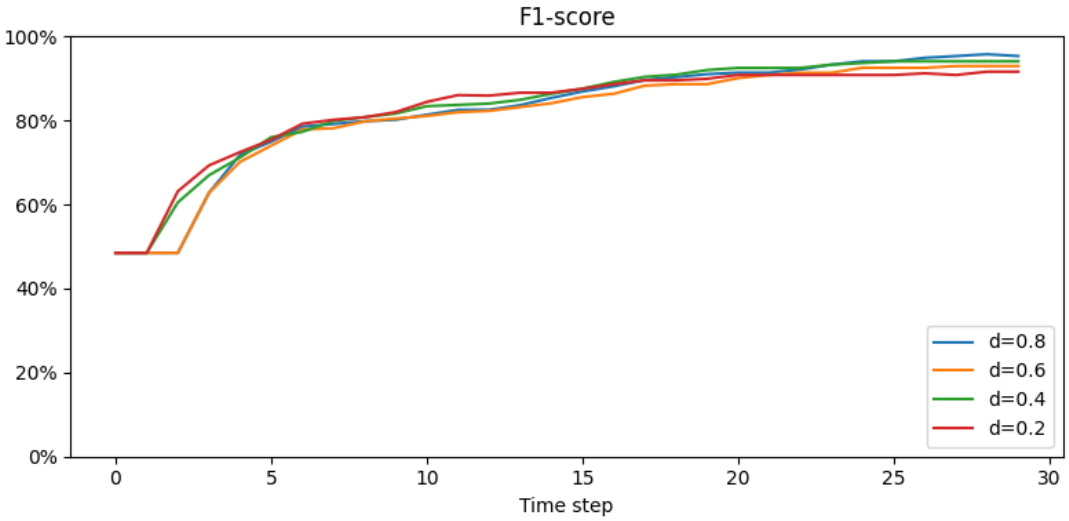

Figure 3.33: Performance of the safety violation prediction models in operation for the *Highway Driving* case study, across different levels of abstraction, using frequency of abstract states as features

state abstraction enhances the safety monitor's capability to recognize potential risks by leveraging similarities in abstract states and learned Q-values from past patterns.

In addition, predicting known unsafe states early is not just valuable, it is essential to achieving acceptable safety levels as recommended by several safety standards for different safety critical domains such as automotive and medical sectors. In these domains, effective risk management depends on the system's ability to detect and address known unsafe states early in operation. For example, in autonomous driving, Tesla's Autopilot must manage a variety of known unsafe states, such as vehicle skidding, excessive braking distances, or unprotected left turns. Even though these conditions are known safety violations, their occurrence depends on dynamic environmental variables. If an autonomous vehicle detects early signs of hydroplaning (a state where the tires lose contact with the road due to water), it can engage preventive measures, such as reducing speed or adjusting the trajectory, long before the vehicle becomes uncontrollable. A delayed response could lead to accidents that might have been prevented with earlier detection. In such scenarios, knowing the unsafe state is not sufficient as early detection of safety violations becomes crucial. In this context, state abstraction techniques, like the Q^*-irrelevance abstraction employed in our approach, allow for generalization over known patterns, enabling the system to detect potentially unsafe behavior that may not match previous episodes exactly but follows recognizable safety violation patterns. This provides a level of generalization over possible unsafe episodes and helps the system adapt to new but related situations.

In short, we offer a light-weight solution to build a safety monitor based on testing results of DRL agents. This is important in practice as testing data contain useful information that can be exploited for monitoring. We should note that the more comprehensive the testing, the better the safety monitoring, as we cannot learn from what we have not observed. However, we alleviate this limitation with state abstraction, although this challenge

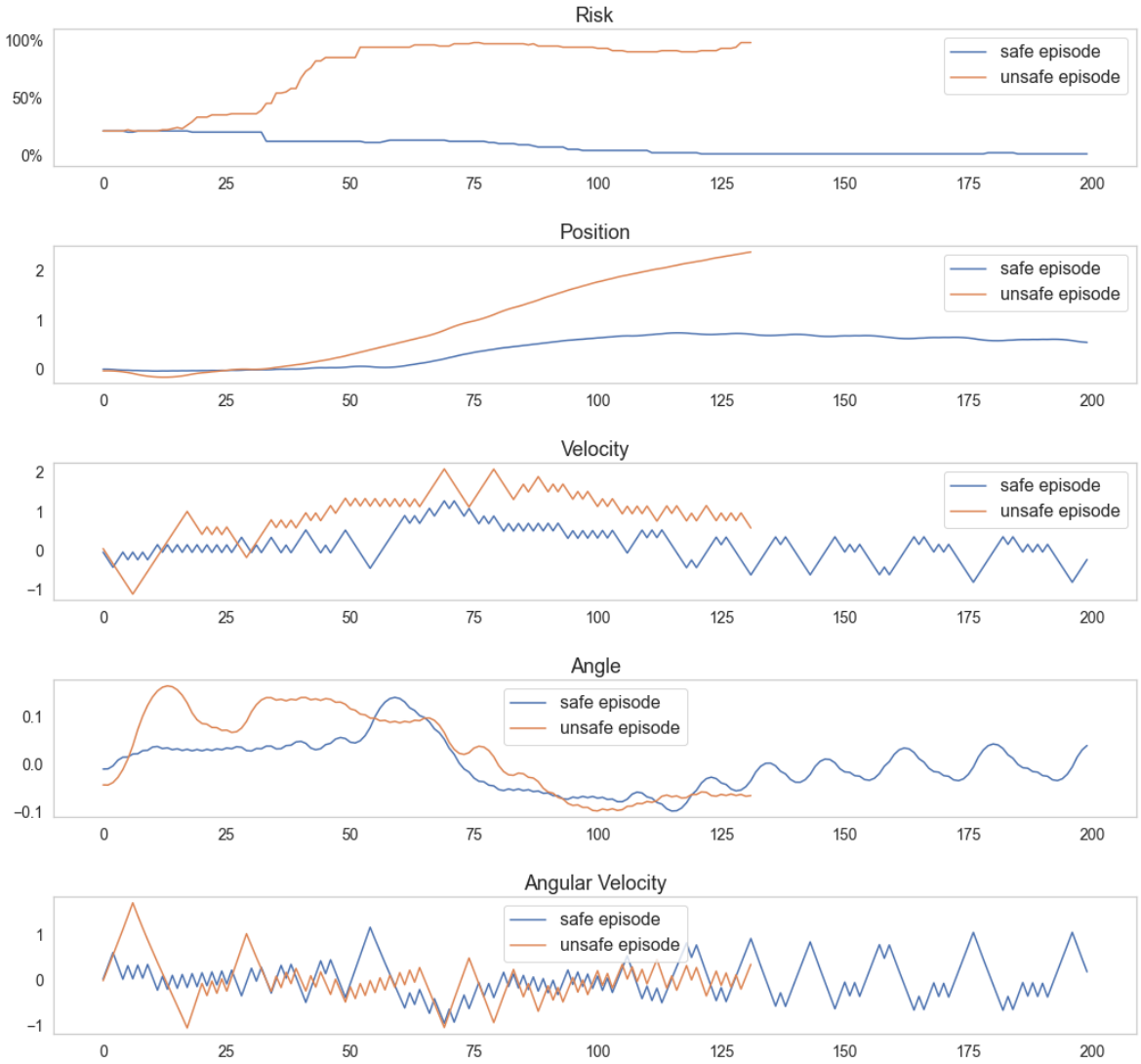

Figure 3.34: Examples for *Cart-Pole*

cannot be fully eliminated.

3.6 Threats to Validity

In this section, we discuss the different threats to the validity of our study and describe how we mitigated them.

Internal threats concern the causal relationship between the treatment and the outcome. One such threat is the choice of an inappropriate state abstraction level. To mitigate this threat, we explored different abstraction levels and identified the ones that (1) significantly reduce the state space, (2) optimize the accuracy of the ML model, and (3) enable the

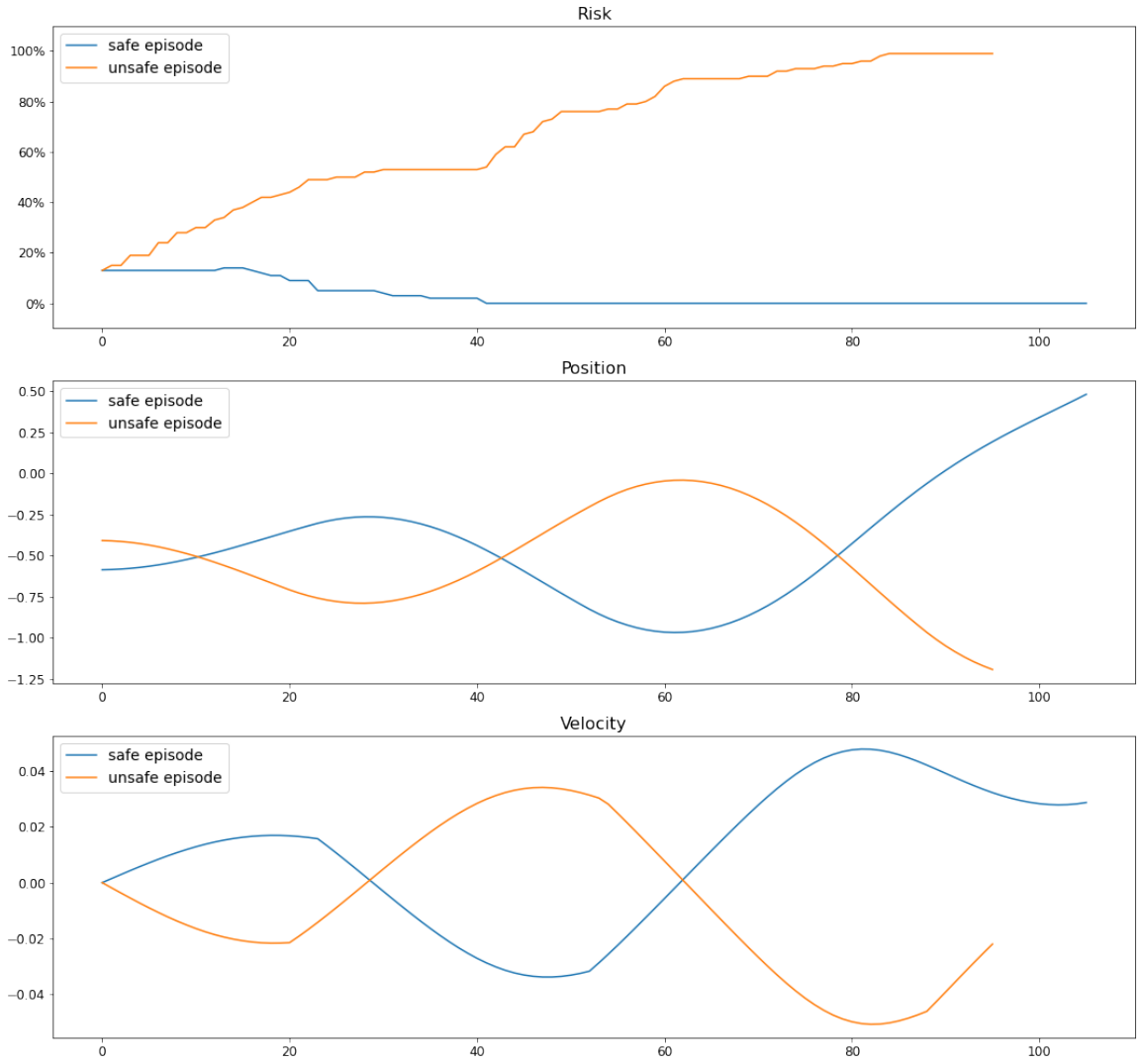

Figure 3.35: Examples for *Mountain-Car*

earliest and most accurate predictions of safety violations during the RL agent execution. Another threat pertains to newly seen abstract states during operation (i.e., abstract states that were not included in the training dataset of the safety violation prediction model), which may impact the accuracy of the safety monitoring system. To mitigate the risk of missing abstract states, we trained our ML model with a large and diverse dataset of RL episodes and use a state abstraction method that considers a large number of concrete states (roughly 450 000 in *Cart-Pole* and 250 000 in *Mountain-Car*). We will study, in future work, the impact of retraining the ML model (to include newly seen abstract states) on the performance of our safety monitoring approach. The scarcity of unsafe episodes poses a threat to accurate safety violation model training. As future work, we will explore extracting unsafe episodes from the later stages of the RL agent's training phase, as they align more closely with the agent's final policy.

Conclusion threats are concerned with the relationship between treatment and outcome.

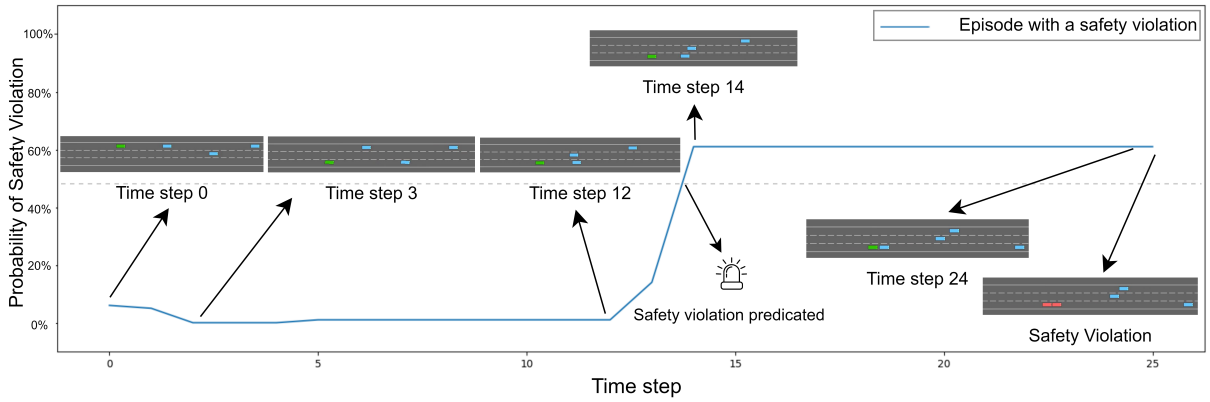

Figure 3.36: Examples for *Highway Driving*

One important threat to consider is the potential impact of a large monitor's overhead on its applicability in real-time scenarios. Such overhead refers to the additional computational resources and time required by safety monitoring. To mitigate this threat, *SMARLA* relies on state abstraction to reduce the state space and computation overhead. It also uses *Random Forest*, which is a lightweight machine learning model known for its fast inference capabilities [22, 25]. This choice helps minimize the computational burden associated with monitoring, allowing for fast processing and response times. Note that, in our case studies, inference times (i.e., safety violation predictions) are in the range of a few milliseconds, thus confirming our expectations. Finally, our monitoring approach is black-box. It does not require access to the internals of the RL agent, which greatly simplifies processing and reduces computation overhead.

Reliability threats concern the replicability of our study results. We rely on publicly available ML models and RL environments. We provide online all the materials required to replicate our study results.

External threats concern the generalizability of our study. Due in part to the high computational expense of our experiments, we relied on three case studies in our work, which may threaten the generalizability of our results. Indeed, experiments for all research questions took about four weeks using a system with a Core-i9 processor, 32GB of memory, and an Nvidia GeForce RTX 3070 GPU with 8GB of memory. However, to mitigate this threat, we relied on benchmark problems widely used in many RL-related studies [16, 17, 21, 29, 107, 110, 146, 151]. Future work includes applying our monitoring approach to different RL problems for broader generalization of the results. Another threat pertains to the ability of *SMARLA* to detect unsafe episodes that are not included in the training set. To mitigate this threat, we use state abstraction and the agent's Q-values to predict safety violations. In our approach, we assume that patterns of Q-value changes observed in training episodes with safety violations can also occur under operational conditions. This allows *SMARLA* not to depend on the specific occurrence of concrete states and extend its predictive capability beyond the concrete states seen during training. This is performed by recognizing and responding to analogous change patterns even in the presence of unseen states, thus generalizing patterns to concrete states that have possibly not been

encountered. Furthermore, it is worth noting that we have explained strategies to deal with unseen abstract states in the deployment environment in Section 3.3.3.1.

3.7 Related Work

This section provides an overview of the most relevant literature on safety monitors. We discuss related works on (1) safety monitors for RL agents, (2) safety monitors for AI/ML-based systems in general , and (3) safety monitoring for Cyber-Physical Systems (CPS). The first topic is obviously directly related to our work, while topic (2) is a bit more general and topic (3) is indirectly related as RL agents are often deployed on Cyber-Physical Systems.

3.7.1 Safety monitors for RL

Several approaches have been proposed in the literature on safe reinforcement learning [44, 47, 92], but only a few of them proposed safety monitors to predict safety violations of the agent at runtime. Junges *et al.* [65] proposed two strategies for monitoring MDPs: forward filtering-based monitoring and model checking-based monitoring. The former involves estimating future possible states of the system based on historical observations. In contrast, the latter uses model-checking techniques to assess the probability of reaching certain states in the MDP. These strategies are only applicable when a detailed model of the environment is available, in contrast to *SMARLA* which works on a broader type of model-free RL algorithms which are more widely used in practice as environment models are challenging to develop and validate [115, 143].

Other approaches have been proposed for the online safety shielding of RL agents where a shield layer is added to the agent and blocks unsafe actions during the execution [69, 87]. In general, online shielding operates by dynamically calculating, during runtime, the collection of potential states that may be reached in the immediate future. Within this set of states, the safety of all feasible actions is assessed and leveraged for shielding purposes as soon as any of the aforementioned states is encountered. In that context, Konighofer *et al.* [69] proposed an online shielding technique for RL agents that employs MDPs and probabilistic model-checking to evaluate all possible actions in the agent's states. By blocking unsafe actions and analyzing the impact of shielding on learning performance and policy safety, they demonstrate improved learning performance with shielding and recommend its application during both the training and execution phases of RL agents. Other shielding approaches have been proposed to increase the safety of RL agents during the training phase [37, 82]. For example, Mallozzi *et al.* [82] introduced *WiseML*, a runtime safety monitoring framework for RL agents. The goal of this framework is to prevent the agent from selecting unsafe actions during training by relying on manual specifications of safety violation patterns using linear-time temporal logic. The monitoring system blocks safety violations, penalizes the agent's reward for unsafe actions, and enhances the learning performance of the agent by accelerating the convergence to its goal. Also, Elsayed *et*

al. [37] proposed a shielding approach for multi-agent reinforcement learning. Their goal is to enforce safety specifications expressed in linear temporal logic and learn safe RL policies in multi-agent environments.

Our research objectives differ from shielding-based studies as we propose a black-box safety monitor for RL agents that does not force the agent to select non-optimal actions (i.e., actions that deviate from the optimal policy) and continue the execution relying on the policy of the agent. Rather, our focus is on the early prediction of safety violations to provide engineers with the flexibility to implement a range of safety mechanisms, including the option to apply early corrective or preventive safety measures.

3.7.2 Safety monitors for AI/ML

Different approaches have been proposed in the literature to monitor the safety of ML-based systems due to concerns about their reliability, especially in safety-critical systems [12, 30, 39, 53, 120]. Some studies focus on detecting distributional shift (between training inputs and data seen in operation time) [12, 30, 39, 53] resulting in DNN mispredictions during operation [59, 119, 120], aiming to maintain accuracy and reduce safety risks. These approaches compare neurons activation patterns [30, 53] or features distributions [12] of inputs during operation with those seen during training. Other approaches involve monitoring DNN outputs and verifying their confidence levels to predict DNNs mispredictions [145].

Other studies such as *Likelihood Regret* [148] detect out-of-distribution DNN input images by reconstructing inputs using autoencoders. However, unlike *SMARLA*, this method does not support the early prediction of safety violations for DRL agents. Furthermore, Stocco *et al.* [120] introduced *SafeOracle*, an unsupervised monitoring tool for autonomous driving systems. *SafeOracle* aims to predict a DNN's misbehavior by building a proxy for the DNN's output confidence level at runtime. The authors have proposed an autoencoder-based image reconstruction technique to forecast the subsequent input images and predict out-of-distribution inputs through a reconstruction error. Additionally, they have trained an anomaly detector on the nominal image inputs of the DNN, which observes the reconstruction error to identify any anomaly. In their approach, they combine confidence estimation, probability distribution fitting, and time series analysis, and show that they were able to identify 77% of safety violations up to six seconds in advance. The same applies for *DeepGuard* [59] which relies on measuring the reconstruction error of input images using autoencoders to predict safety violations. *SelfOracle* and *DeepGuard* are designed for DNN systems primarily focused on classification or regression, where the input spaces consist of images. These methods require image transformations to effectively train the autoencoders and employ distance metrics specific to images to measure reconstruction error and estimate the probability of safety violations.

Our approach, however, is fundamentally different in terms of inputs and methodology. Further, in our case studies, the DRL agent and the underlying DNN do not take images as inputs. Instead, they process state parameters such as the position and velocity of the agent. This distinction is crucial as the nature of the input data in our approach

differs significantly from that in traditional image-based DNN safety monitors. In fact, our approach is agnostic to the type of DRL agent's inputs since we only rely on the agent's Q-values. Furthermore, we monitor the behavior of the agents to predict violations based on a training dataset that contains both safe and unsafe episodes, as opposed to relying solely on out-of-distribution inputs.

3.7.3 Safety monitoring for CPS

Safety monitoring of CPS is crucial for ensuring their reliable and secure operation, particularly when integrating deep learning models. Various approaches have been proposed to address the safety monitoring of CPS [14, 54, 90, 121, 150]. For example, Xie *et al.* [150] proposed *Mosaic*, a model-based safety analysis framework specifically designed for AI-enabled CPSs. This framework addresses the safety challenges that arise from the integration of AI-based controllers in CPSs, which introduce uncertainties and potential safety risks due to their random exploration nature and lack of systematic explanations for their behavior. *Mosaic* constructs an abstract MDP model to represent the AI-CPS, enabling safety analysis through two main components: online safety monitoring and offline model-guided falsification. Online safety monitoring employs probabilistic model checking to predict safety issues during real-time operations and switches control to a predefined safety controller if necessary. Offline model-guided falsification searches for counterexamples that violate safety specifications using a combination of global and local search strategies. Our method, *SMARLA*, differs from *Mosaic* in several aspects. *Mosaic* is model-dependent, requiring the construction of an MDP model. This process can be complex and computationally expensive, especially for high-dimensional systems. In contrast, *SMARLA* is model-free, eliminating the need for constructing abstract models, thereby simplifying the safety analysis process and reducing computational overhead. Furthermore, *Mosaic* focuses on general AI-CPS applications but does not specifically address RL environments. *SMARLA*, on the other hand, is specifically designed for RL environments, providing tailored safety solutions for RL applications. In summary, while *Mosaic* offers a framework for the safety analysis of AI-CPS through model-based methods, *SMARLA* addresses key limitations by being model-free, specifically designed for RL environments, and suitable for black-box RL agents. This provides a more versatile and efficient solution for ensuring safety in reinforcement learning applications.

Henzinger *et al.* [54] proposed an abstraction-based monitoring framework for neural networks, focusing on novelty detection during runtime. Their method utilizes box abstractions to monitor the behavior of neural networks by observing hidden layers. Specifically, they define intervals (or boxes) around the values seen during training for each neuron in the monitored layers. If an input causes the network to produce values outside these pre-defined boxes, it is flagged as a novelty. This method balances false positives and novel input detection through adjustable parameters. However, it requires access to the internal structure of the neural network, making it a white-box approach. This limitation restricts its applicability to scenarios where such access is available. In contrast, *SMARLA* is black-box and does not require knowledge of the model's internals, making it more versatile.

While the proposed framework is primarily designed for classification tasks, *SMARLA* is tailored to reinforcement learning environments. Furthermore, Strickland *et al.* [121] proposed an LSTM-based safety monitoring approach for CPS. The model analyses the driving environment images as well as the vehicle's state to predict potential collisions. However, this method is tailored to image inputs, whereas *SMARLA* is versatile and not restricted to any specific type of input data. Michelmore *et al.* [89, 90] relied on the evaluation of uncertainty measures, in real-time within end-to-end controllers, and Bayesian inference methods for autonomous driving to predict safety violations. However, these approaches require access to the internal weights of the DNN models, and are thus white-box.

Several other approaches have been proposed in the literature on the runtime verification of CPS to check the compliance of such systems with safety requirements. For example, Grundt *et al.* [46] presented a formalization approach for spatio-temporal requirements in autonomous driving systems and proposed a runtime verification method to ensure these systems comply with complex requirements during validation runs. Their monitoring system continuously evaluates the behavior of the autonomous driving system to provide real-time compliance verdicts with the system's requirements. The same applies to Zapridou *et al.* [152] where the authors propose a runtime verification approach to ensure that autonomous driving systems, using an adaptive cruise control system as a case study, operate safely and effectively within simulated urban driving environments. This is achieved by continuously monitoring the system's behavior against predefined safety and performance specifications expressed in signal temporal logic. The objectives of the above approaches substantially differ from our work because they concentrate on the runtime verification of autonomous driving systems. Unlike *SMARLA*, which predicts the probability of safety violations at each time step during the operation of RL agents, these methodologies focus on verifying system compliance with predefined requirements at runtime. Therefore, we do not include these approaches as a baseline for comparison.

Overall, *SMARLA* stands out by providing a novel black-box safety monitoring approach that is tailored to RL agents and prioritizes early prediction of safety violations, thus providing the flexibility to implement a wide range of safety mechanisms. The use of state abstraction and ML models allows for efficient and accurate safety violation prediction, without requiring detailed environment models, making it well-suited for a broader range of model-free RL algorithms that are commonly used.

3.8 Conclusion

In this chapter, we propose *SMARLA*, a black-box safety monitoring approach for reinforcement learning agents. We rely on a machine learning model to predict safety violations of RL agents early during their execution. Our approach relies on Q-values and is agnostic to the input of the RL agent. We employ state abstraction to reduce the state space, enabling improved learnability to predict violations. We quantitatively and qualitatively evaluate our safety monitoring approach on three widely used RL benchmarks. Our results demonstrate the high accuracy of *SMARLA* in predicting the safety violations of RL agents during their execution. Additionally, our empirical results show that such accurate prediction can be made early long before the actual occurrence of violations, allowing for timely

damage prevention and mitigation. In future work, we intend to expand our evaluation to include additional RL case studies. Furthermore, to further improve accuracy and early predictions, our aim is to investigate the use of other types of features that consider temporal information regarding the agent's states and actions in RL episodes. For instance, we can consider the sequence of abstract states instead of only considering their presence or absence as features.

Chapter 4

Conclusion and Future Research Directions

This thesis presents two novel approaches *STARLA* and *SMARLA*, aiming at improving the safety and reliability of DRL agents through testing and runtime monitoring. The following key findings and contributions summarize the research:

We introduced *STARLA* as a data-box search-based testing approach for detecting functional-faulty episodes in DRL agents. Through machine learning-based classification, we were able to predict these faulty episodes with high accuracy, thus guiding and improving the efficiency of the testing process. Empirical evaluations on two well-known RL environments, demonstrated that *STARLA* significantly outperforms random testing methods, detecting more faults within the same testing budget. This approach is particularly valuable for pre-deployment testing of DRL agents in safety-critical domains like autonomous driving and healthcare. The open-source prototype and datasets provided will allow researchers and practitioners to replicate and build upon these findings.

We proposed *SMARLA* as a black-box, runtime safety monitoring approach that predicts safety violations during the operation of DRL agents. By leveraging state abstraction and Q-values, *SMARLA* is capable of detecting potential safety issues early, providing sufficient time for proactive interventions. Our empirical evaluation showed that *SMARLA* was highly effective across widely-known RL case studies, such as Cart-Pole and Highway Driving, with early and accurate safety violation predictions. The versatility of *SMARLA*, which operates without internal access to the DRL agent, makes it applicable to a wide range of real-world systems.

Together, these contributions demonstrate that combining search-based testing with runtime monitoring can significantly help enhance the safety and reliability of DRL agents in both pre-deployment and operational phases.

4.1 Future Research Directions

The research opens several avenues for future work:

- **Generalization to Complex RL Environments**: While *STARLA* and *SMARLA* were evaluated on well-known benchmark problems, future research can explore their application to more complex and diverse RL environments to assess their scalability and effectiveness across a wider range of industrial safety-critical systems.
- **Fine-tuning of Parameters**: *STARLA*'s performance can be further enhanced by investigating automated methods for fine-tuning its parameters for different environments. Additionally, future work can explore the optimization of *SMARLA*'s prediction thresholds to balance early detection and false positive rates.
- **Dynamic Parameter Selection**: The current implementation of *SMARLA* relies on fixed parameters for aspects such as the prediction thresholds and abstraction levels. A promising direction for future work is to explore dynamic parameter selection strategies. This could involve adapting parameters monitoring phases based on real-time feedback from the system. For example, adjusting the safety violation prediction thresholds dynamically based on the agent's behavior could improve the balance between early detection and false positives. Such adaptive methods could increase the overall robustness and efficiency.
- **Integration into Real-World Systems**: Both approaches can be integrated into real-world systems, such as autonomous driving platforms, to assess their practical impact on real-time decision-making processes. These real-world trials would provide valuable insights into the deployment challenges and potential adjustments needed for industrial applications.
- **Handling Unseen States**: Future work could focus on expanding the training data for the ML models used in *STARLA* and *SMARLA* to incorporate a wider range of abstract states or use adaptive retraining methods to further improve the performance of *SMARLA*.
- **Integrating Curiosity-Driven Testing**: Future work could explore combining curiosity-driven testing [52] with *STARLA* to explore less visited or uncertain regions of the state-action space, potentially uncovering rare faults. This hybrid approach could enhance fault detection while addressing challenges related to large state spaces and computational costs.

4.2 Broader Impact

The proposed approaches have the potential to make a significant impact on the increasingly numerous safety-critical applications that employ DRL agents, such as autonomous

vehicles, healthcare systems, and industrial robots. By addressing both pre-deployment and runtime safety concerns, *STARLA* and *SMARLA* contribute to the trustworthiness of DRL systems, reducing the risks associated with their deployment in real-world environments. The frameworks and methodologies developed in this thesis serve as foundational tools for ensuring the safe operation of such intelligent systems, paving the way for more robust and safer artificial intelligence solutions.

References

[1] David Abel, Dilip Arumugam, Lucas Lehnert, and Michael Littman. State abstractions for lifelong reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 10–19. PMLR, 10–15 Jul 2018.

[2] Zohreh Aghababaeyan, Manel Abdellatif, Lionel Briand, S Ramesh, and Mojtaba Bagherzadeh. Black-box testing of deep neural networks through test case diversity. *IEEE Transactions on Software Engineering*, 2023.

[3] Zohreh Aghababaeyan, Manel Abdellatif, Mahboubeh Dadkhah, and Lionel Briand. Deepgd: A multi-objective black-box test selection approach for deep neural networks, 2024.

[4] Carlos Aguilar-Ibáñez, Julio Mendoza-Mendoza, and Jorge Dávila. Stabilization of the cart pole system: by sliding mode control. *Nonlinear Dynamics*, 78(4):2769–2777, 2014.

[5] Riad Akrour, Filipe Veiga, Jan Peters, and Gerhard Neumann. Regularizing reinforcement learning with state abstraction. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 534–539. IEEE, 2018.

[6] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. Measuring and maintaining population diversity in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 153–168. Springer, 2020.

[7] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[8] Izzat Alsmadi. Using genetic algorithms for test case generation and selection optimization. In *CCECE 2010*, pages 1–4. IEEE, 2010.

[9] Vincent J. Amuso and Jason Enslin. The strength pareto evolutionary algorithm 2 (spea2) applied to simultaneous multi- mission waveform design. In *2007 International Waveform Diversity and Design Conference*, pages 407–417, 2007.

[10] Snobin Antony, Raghi Roy, and Yaxin Bi. Q-learning: Solutions for grid world problem with forward and backward reward propagations. In Max Bramer and Frederic

Stahl, editors, *Artificial Intelligence XL*, pages 266–271, Cham, 2023. Springer Nature Switzerland.

[11] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[12] Koorosh Aslansefat, Ioannis Sorokos, Declan Whiting, Ramin Tavakoli Kolagari, and Yiannis Papadopoulos. Safeml: safety monitoring of machine learning classifiers through statistical difference measures. In *Model-Based Safety and Assessment: 7th International Symposium, IMBSA 2020, Lisbon, Portugal, September 14–16, 2020, Proceedings 7*, pages 197–211. Springer, 2020.

[13] Ahmad Taher Azar, Anis Koubaa, Nada Ali Mohamed, Habiba A Ibrahim, Zahra Fathy Ibrahim, Muhammad Kazim, Adel Ammar, Bilel Benjdira, Alaa M Khamis, Ibrahim A Hameed, et al. Drone deep reinforcement learning: A review. *Electronics*, 10(9):999, 2021.

[14] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. *Lectures on Runtime Verification: Introductory and Advanced Topics*, pages 135–175, 2018.

[15] Vahid Behzadan and Munir Arslan. Whatever does not kill deep reinforcement learning, makes it stronger. *arXiv preprint arXiv:1712.09344*, 2017.

[16] Vahid Behzadan and William Hsu. Sequential triggers for watermarking of deep reinforcement learning policies. *ArXiv*, abs/1906.01126, 2019.

[17] Vahid Behzadan and William H. Hsu. Adversarial exploitation of policy imitation. *ArXiv*, abs/1906.01121, 2019.

[18] Vahid Behzadan and Arslan Munir. Mitigation of policy manipulation attacks on deep q-networks with parameter-space noise. *ArXiv*, abs/1806.02190, 2018.

[19] Ali Beikmohammadi and Sindri Magnússon. Comparing nars and reinforcement learning: An analysis of ona and q-learning algorithms. In Patrick Hammer, Marjan Alirezaie, and Claes Strannegård, editors, *Artificial General Intelligence*, pages 21–31, Cham, 2023. Springer Nature Switzerland.

[20] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013.

[21] Francesco Bellotti, Luca Lazzaroni, Alessio Capello, Marianna Cossu, Alessandro De Gloria, and Riccardo Berta. Explaining a deep reinforcement learning (drl)-based automated driving agent in highway simulations. *IEEE Access*, 11:28522–28550, 2023.

[22] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[23] Cornelius Buerkle, Florian Geissler, Michael Paulitsch, and Kay-Ulrich Scholl. Fault-tolerant perception for automated driving a lightweight monitoring approach. *arXiv preprint arXiv:2111.12360*, 2021.

[24] Taejoon Byun, Sanjai Rayadurgam, and Mats PE Heimdahl. Black-box testing of deep neural networks. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 309–320. IEEE, 2021.

[25] Jie Cao, Zhaohui Guo, Yongjun Lv, Man Xu, Chiyue Huang, and Huizhi Liang. Pollution risk prediction for cadmium in soil from an abandoned mine based on random forest model. *International Journal of Environmental Research and Public Health*, 20(6):5097, Mar 2023.

[26] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[27] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[28] Thomas Chaffre, Julien Moras, Adrien Chan-Hon-Tong, and Julien Marzat. Sim-to-real transfer with incremental environment complexity for reinforcement learning of depth-based robot navigation. *arXiv preprint arXiv:2004.14684*, 2020.

[29] Kangjie Chen, Tianwei Zhang, Xiaofei Xie, and Yang Liu. Stealing deep reinforcement learning models for fun and profit. *CoRR*, abs/2006.05032, 2020.

[30] Chih-Hong Cheng, Georg Nührenberg, and Hirotoshi Yasuoka. Runtime monitoring neuron activation patterns. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 300–303. IEEE, 2019.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[32] Kenji Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245, 2000.

[33] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

[34] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[35] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning, 2019.

[36] Valerie J. Easton and John H. McColl. Statistics glossary v1.1. In *Statistics Glossary v1.1*, 1997.

[37] Ingy ElSayed-Aly, Suda Bharadwaj, Christopher Amato, Rüdiger Ehlers, Ufuk Topcu, and Lu Feng. Safe multi-agent reinforcement learning via shielding. *arXiv preprint arXiv:2101.11196*, 2021.

[38] Shuo Feng, Haowei Sun, Xintao Yan, Haojie Zhu, Zhengxia Zou, Shengyin Shen, and Henry X Liu. Dense reinforcement learning for safety validation of autonomous vehicles. *Nature*, 615(7953):620–627, 2023.

[39] Raul Sena Ferreira, Jean Arlat, Jérémie Guiochet, and Hélène Waeselynck. Benchmarking safety monitors for image classifiers with machine learning. In *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 7–16. IEEE, 2021.

[40] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *ArXiv*, abs/1706.10295, 2017.

[41] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[42] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International conference on machine learning*, pages 2052–2062. PMLR, 2019.

[43] Jasmina Gajcin, James McCarthy, Rahul Nair, Radu Marinescu, Elizabeth Daly, and Ivana Dusparic. Iterative reward shaping using human feedback for correcting reward misspecification, 2023.

[44] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[45] Salah Ghamizi, Maxime Cordy, Martin Gubri, Mike Papadakis, Andrey Boystov, Yves Le Traon, and Anne Goujon. Search-based adversarial testing and improvement of constrained credit scoring systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1089–1100, 2020.

[46] Dominik Grundt, Anna Köhne, Ishan Saxena, Ralf Stemmer, Bernd Westphal, and Eike Möhlmann. Towards runtime monitoring of complex system requirements for autonomous driving functions. *arXiv preprint arXiv:2209.14032*, 2022.

[47] Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, Yaodong Yang, and Alois Knoll. A review of safe reinforcement learning: Methods, theory and applications. *arXiv preprint arXiv:2205.10330*, 2022.

[48] Gym library. `https://github.com/openai/gym`, 2020.

[49] Yi Han, Benjamin I. P. Rubinstein, Tamas Abraham, Tansu Alpcan, Olivier Y. de Vel, Sarah Monazam Erfani, David Hubczenko, Christopher Leckie, and Paul Montague. Reinforcement learning for autonomous defence in software-defined networking. *ArXiv*, abs/1808.05770, 2018.

[50] K.M. Hansen, A.P. Ravn, and V. Stavridou. From safety analysis to software requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, 1998.

[51] Fitash Ul Haq, Donghwan Shin, Lionel C. Briand, Thomas Stifter, and Jun Wang. Automatic test suite generation for key-points detection dnns using many-objective search (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 91–102, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Junda He, Zhou Yang, Jieke Shi, Chengran Yang, Kisub Kim, Bowen Xu, Xin Zhou, and David Lo. Curiosity-driven testing for sequential decision-making process. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[53] Thomas A Henzinger, Anna Lukina, and Christian Schilling. Outside the box: Abstraction-based monitoring of neural networks. *arXiv preprint arXiv:1911.09032*, 2019.

[54] Thomas A Henzinger, Anna Lukina, and Christian Schilling. Outside the box: Abstraction-based monitoring of neural networks. In *ECAI 2020*, pages 2433–2440. IOS Press, 2020.

[55] M Hessel, J Modayil, H van Hasselt, T Schaul, G Ostrovski, W Dabney, and D Silver. Rainbow: Combining improvements in deep reinforcement learning, corr abs/1710.02298. *arXiv preprint arXiv:1710.02298*, 2017.

[56] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[57] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[58] Sandy H. Huang, Nicolas Papernot, Ian J. Goodfellow, Yan Duan, and P. Abbeel. Adversarial attacks on neural network policies. *ArXiv*, abs/1702.02284, 2017.

[59] Manzoor Hussain, Nazakat Ali, and Jang-Eui Hong. Deepguard: A framework for safeguarding autonomous driving systems from inconsistent behaviour. *Automated Software Engineering*, 29(1):1, 2022.

[60] Inaam Ilahi, Muhammad Usama, Junaid Qadir, Muhammad Umar Janjua, Ala Al-Fuqaha, Dinh Thai Huang, and Dusit Niyato. Challenges and countermeasures for adversarial attacks on deep reinforcement learning, 2021.

[61] Road vehicles — safety of the intended functionality. ISO/PAS 21448:2019, 2019.

[62] Road vehicles – functional safety. ISO 26262:2018, 2018.

[63] Beakcheol Jang, Myeonghwi Kim, Gaspard Harerimana, and Jong Wook Kim. Q-learning algorithms: A comprehensive classification and applications. *IEEE access*, 7:133653–133667, 2019.

[64] Nan Jiang. Notes on state abstractions, 2018.

[65] Sebastian Junges, Hazem Torfah, and Sanjit A Seshia. Runtime monitors for markov decision processes. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*, pages 553–576. Springer, 2021.

[66] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pages 97–117. Springer, 2017.

[67] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 23(6):4909–4926, 2021.

[68] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[69] Bettina Könighofer, Julian Rudolf, Alexander Palmisano, Martin Tappler, and Roderick Bloem. Online shielding for reinforcement learning. *Innovations in Systems and Software Engineering*, pages 1–16, 2022.

[70] Jernej Kos and Dawn Song. Delving into adversarial attacks on deep policies. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

[71] Alex Kulesza and Ben Taskar. Determinantal point processes for machine learning. *arXiv preprint arXiv:1207.6083*, 2012.

[72] Xian Yeow Lee, Yasaman Esfandiari, Kai Liang Tan, and Soumik Sarkar. Query-based targeted action-space adversarial policies on deep reinforcement learning agents. In *Proceedings of the ACM/IEEE 12th International Conference on Cyber-Physical Systems*, ICCPS '21, page 87–97, New York, NY, USA, 2021. Association for Computing Machinery.

[73] Chen Lei. Deep reinforcement learning. In *Deep Learning and Practice with MindSpore*, pages 217–243. Springer, 2021.

[74] Edouard Leurent. An environment for autonomous driving decision-making. `https://github.com/eleurent/highway-env`, 2018.

[75] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. *ISAIM*, 4(5):9, 2006.

[76] Zhuo Li, Xiongfei Wu, Derui Zhu, Mingfei Cheng, Siyuan Chen, Fuyuan Zhang, Xiaofei Xie, Lei Ma, and Jianjun Zhao. Generative Model-Based Testing on Decision-Making Policies . In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 243–254, Los Alamitos, CA, USA, September 2023. IEEE Computer Society.

[77] Junyu Lin, Lei Xu, Yingqi Liu, and Xiangyu Zhang. Black-box adversarial sample generation based on differential evolution. *Journal of Systems and Software*, 170:110767, 2020.

[78] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. Tactics of adversarial attack on deep reinforcement learning agents. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, page 3756–3762. AAAI Press, 2017.

[79] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. Tactics of adversarial attack on deep reinforcement learning agents, 2019.

[80] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, and et al. Deepgauge: multi-granularity testing criteria for deep learning systems. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Sep 2018.

[81] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

[82] Piergiuseppe Mallozzi, Ezequiel Castellano, Patrizio Pelliccione, Gerardo Schneider, and Kenji Tei. A runtime monitoring framework to enforce invariants on reinforcement learning agents exploring complex environments. In *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, pages 5–12. IEEE, 2019.

[83] Enrico Marchesini, Luca Marzari, Alessandro Farinelli, and Christopher Amato. Safe deep reinforcement learning by verifying task-level properties. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1466–1475, 2023.

[84] Alonso Marco, Felix Berkenkamp, Philipp Hennig, Angela P Schoellig, Andreas Krause, Stefan Schaal, and Sebastian Trimpe. Virtual vs. real: Trading off simulations and physical experiments in reinforcement learning with bayesian optimization. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1557–1563. IEEE, 2017.

[85] Antoine Marot, Isabelle Guyon, Benjamin Donnot, Gabriel Dulac-Arnold, Patrick Panciatici, Mariette Awad, Aidan O'Sullivan, Adrian Kelly, and Zigfried Hampel-Arias. L2rpn: Learning to run a power network in a sustainable world neurips2020 challenge design, 2020.

[86] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[87] Daniel Melcer, Christopher Amato, and Stavros Tripakis. Shield decentralization for safe multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 35:13367–13379, 2022.

[88] Lingheng Meng, Rob Gorbet, and Dana Kulić. Memory-based deep reinforcement learning for pomdps. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5619–5626. IEEE, 2021.

[89] Rhiannon Michelmore, Marta Kwiatkowska, and Yarin Gal. Evaluating uncertainty quantification in end-to-end autonomous driving control. *arXiv e-prints*, pages arXiv–1811, 2018.

[90] Rhiannon Michelmore, Matthew Wicker, Luca Laurenti, Luca Cardelli, Yarin Gal, and Marta Kwiatkowska. Uncertainty quantification with statistical guarantees in end-to-end autonomous driving control. In *2020 IEEE international conference on robotics and automation (ICRA)*, pages 7344–7350. IEEE, 2020.

[91] Brad L. Miller and David E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evol. Comput.*, 4(2):113–131, jun 1996.

[92] Paulina Stevia Nouwou Mindom, Amin Nikanjam, Foutse Khomh, and John Mullins. On assessing the safety of reinforcement learning algorithms using formal methods. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 260–269. IEEE, 2021.

[93] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[94] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[95] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, Computer Laboratory, 1990.

[96] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *CoRR*, abs/1610.08401, 2016.

[97] Ronald W Morrison and Kenneth A De Jong. Measurement of population diversity. In *International conference on artificial evolution (evolution artificielle)*, pages 31–41. Springer, 2001.

[98] Tadahiko Murata, Hisao Ishibuchi, et al. Moga: multi-objective genetic algorithms. In *IEEE international conference on evolutionary computation*, volume 1, pages 289–294. IEEE Piscataway, NJ, USA, 1995.

[99] Savinay Nagendra, Nikhil Podila, Rashmi Ugarakhod, and Koshy George. Comparison of reinforcement learning algorithms applied to the cart-pole problem. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 26–32. IEEE, 2017.

[100] Ali Nasr, Keaton Inkol, and John McPhee. Safety in wearable robotic exoskeletons: Design, control, and testing guidelines. *Journal of Mechanisms and Robotics*, pages 1–13, 2024.

[101] Amin Nikanjam, Mohammad Mehdi Morovati, Foutse Khomh, and Houssem Ben Braiek. Faults in deep reinforcement learning programs: a taxonomy and a detection approach. *Automated Software Engineering*, 29(1):1–32, 2022.

[102] Yoshihiro Okawa, Tomotake Sasaki, Hitoshi Yanami, and Toru Namerikawa. Safe exploration method for reinforcement learning under existence of disturbance, 2023.

[103] Loïc Omnes, Antoine Marot, and Benjamin Donnot. Adversarial training for a continuous robustness control problem in power systems. In *2021 IEEE Madrid PowerTech*, pages 1–6. IEEE, 2021.

[104] Openai. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`, 2018. [Accessed 24 Jan. 2022.].

[105] Alexander Pan, Yongkyun Lee, Huan Zhang, Yize Chen, and Yuanyuan Shi. Improving robustness of reinforcement learning for power system control with adversarial training, 2021.

[106] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[107] Anay Pattanaik, Zhenyi Tang, Shuijing Liu, Gautham Bommannan, and Girish Chowdhary. Robust deep reinforcement learning with adversarial attacks. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 2040–2042, 2018.

[108] Martin Pecka and Tomas Svoboda. Safe exploration techniques for reinforcement learning–an overview. In *Modelling and Simulation for Autonomous Systems: First International Workshop, MESAS 2014, Rome, Italy, May 5-6, 2014, Revised Selected Papers 1*, pages 357–375. Springer, 2014.

[109] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. *Commun. ACM*, 62(11):137–145, October 2019.

[110] Alessandro Pighetti, Francesco Bellotti, Changjae Oh, Luca Lazzaroni, Luca Forneris, Matteo Fresta, and Riccardo Berta. Investigating adversarial policy learning for robust agents in automated driving highway simulations. In Francesco Bellotti, Miltos D. Grammatikakis, Ali Mansour, Massimo Ruo Roch, Ralf Seepold, Agusti Solanas, and Riccardo Berta, editors, *Applications in Electronics Pervading Industry, Environment and Society*, pages 124–129, Cham, 2024. Springer Nature Switzerland.

[111] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & amp; Sons, Inc., USA, 1st edition, 1994.

[112] Amin Rakhsha, Xuezhou Zhang, Xiaojin Zhu, and Adish Singla. Reward poisoning in reinforcement learning: Attacks against unknown learners in unknown environments, 2021.

[113] Jonathan Schaeffer, Nathan Sturtevant, Robert Holte, and Ken Anderson. Coarse-to-fine search techniques. In *Coarse-to-Fine Search Techniques*, 2008.

[114] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[115] Francesco Semeraro, Alexander Griffiths, and Angelo Cangelosi. Human–robot collaboration and machine learning: A systematic review of recent research. *Robotics and Computer-Integrated Manufacturing*, 79:102432, 2023.

[116] Mohit Sewak. *Deep reinforcement learning*. Springer, 2019.

[117] Qunying Song, Kaige Tan, Per Runeson, and Stefan Persson. Critical scenario identification for realistic testing of autonomous driving systems. *Software Quality Journal*, 31(2):441–469, 2023.

[118] Denis Steckelmacher, Hélène Plisnier, Diederik M. Roijers, and Ann Nowé. Sample-efficient model-free reinforcement learning with off-policy critics. In Ulf Brefeld, Elisa Fromont, Andreas Hotho, Arno Knobbe, Marloes Maathuis, and Céline Robardet, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 19–34, Cham, 2020. Springer International Publishing.

[119] Andrea Stocco, Paulo J Nunes, Marcelo D'Amorim, and Paolo Tonella. Thirdeye: Attention maps for safe autonomous driving systems. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[120] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, pages 359–371, 2020.

[121] Mark Strickland, Georgios Fainekos, and Heni Ben Amor. Deep predictive models for collision risk assessment in autonomous driving. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4685–4692. IEEE, 2018.

[122] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Testing Deep Neural Networks. *arXiv e-prints*, page arXiv:1803.04792, March 2018.

[123] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Deepconcolic: Testing and debugging deep neural networks. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 111–114, 2019.

[124] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks, 2018.

[125] Richard S Sutton. Andrew g barto. reinforcement learning: An introduction. *MIT press*, 2018.

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

[127] Phillip Swazinna, Steffen Udluft, Daniel Hein, and Thomas Runkler. Comparing model-free and model-based algorithms for offline reinforcement learning. *arXiv preprint arXiv:2201.05433*, 2022.

[128] Kai Liang Tan, Yasaman Esfandiari, Xian Yeow Lee, Aakanksha, and Soumik Sarkar. Robustifying reinforcement learning agents via action space adversarial training. *CoRR*, abs/2007.07176, 2020.

[129] Kai Liang Tan, Yasaman Esfandiari, Xian Yeow Lee, Soumik Sarkar, et al. Robustifying reinforcement learning agents via action space adversarial training. In *2020 American control conference (ACC)*, pages 3959–3964. IEEE, 2020.

[130] R. Tanabe, H. Ishibuchi, and A. Oyama. Benchmarking multi- and many-objective evolutionary algorithms under two optimization scenarios. *IEEE Access*, 5:19597–19619, 2017.

[131] Chen Tang, Ben Abbatematteo, Jiaheng Hu, Rohan Chandra, Roberto Martín-Martín, and Peter Stone. Deep reinforcement learning for robotics: A survey of real-world successes. *arXiv preprint arXiv:2408.03539*, 2024.

[132] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pages 5981–5988, 2020.

[133] Martin Tappler, Filip Cano Córdoba, Bernhard K Aichernig, and Bettina Könighofer. Search-based testing of reinforcement learning. *arXiv preprint arXiv:2205.04887*, 2022.

[134] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[135] Arash Tavakoli, Fabio Pardo, and Petar Kormushev. Action branching architectures for deep reinforcement learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 32, 2018.

[136] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars, 2018.

[137] Miller Trujillo, Mario Linares-Vásquez, Camilo Escobar-Velásquez, Ivana Dusparic, and Nicolás Cardozo. Does neuron coverage matter for deep reinforcement learning? a preliminary study. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 215–220, 2020.

[138] Matteo Turchetta, Andrey Kolobov, Shital Shah, Andreas Krause, and Alekh Agarwal. Safe reinforcement learning via curriculum induction. *Advances in Neural Information Processing Systems*, 33:12151–12162, 2020.

[139] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[140] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[141] Eugene Vinitsky, Yuqing Du, Kanaad Parvate, Kathy Jang, Pieter Abbeel, and Alexandre M. Bayen. Robust reinforcement learning using adversarial populations. *CoRR*, abs/2008.01825, 2020.

[142] Dongshu Wang, Dapei Tan, and Lei Liu. Particle swarm optimization algorithm: an overview. *Soft computing*, 22:387–408, 2018.

[143] Hao-nan Wang, Ning Liu, Yi-yun Zhang, Da-wei Feng, Feng Huang, Dong-sheng Li, and Yi-ming Zhang. Deep reinforcement learning: a survey. *Frontiers of Information Technology & Electronic Engineering*, 21(12):1726–1744, 2020.

[144] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2016.

[145] Michael Weiss and Paolo Tonella. Fail-safe execution of deep learning based systems through uncertainty monitoring. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 24–35. IEEE, 2021.

[146] Lu Wen, Eric H. Tseng, Huei Peng, and Songan Zhang. Dream to adapt: Meta reinforcement learning by latent context imagination and mdp imagination. *IEEE Robotics and Automation Letters*, pages 1–8, 2024.

[147] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[148] Zhisheng Xiao, Qing Yan, and Yali Amit. Likelihood regret: An out-of-distribution detection score for variational auto-encoder. *Advances in neural information processing systems*, 33:20685–20696, 2020.

[149] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 146–157, 2019.

[150] Xuan Xie, Jiayang Song, Zhehua Zhou, Fuyuan Zhang, and Lei Ma. Mosaic: Model-based safety analysis framework for ai-enabled cyber-physical systems, 2023.

[151] Xu Yan, Haiming Zhang, Yingjie Cai, Jingming Guo, Weichao Qiu, Bin Gao, Kaiqiang Zhou, Yue Zhao, Huan Jin, Jiantao Gao, Zhen Li, Lihui Jiang, Wei Zhang, Hongbo Zhang, Dengxin Dai, and Bingbing Liu. Forging vision foundation models for autonomous driving: Challenges, methodologies, and opportunities, 2024.

[152] Eleni Zapridou, Ezio Bartocci, and Panagiotis Katsaros. Runtime verification of autonomous driving systems in carla. In *International Conference on Runtime Verification*, pages 172–183. Springer, 2020.

[153] Huan Zhang, Hongge Chen, Chaowei Xiao, Bo Li, Mingyan Liu, Duane Boning, and Cho-Jui Hsieh. Robust deep reinforcement learning against adversarial perturbations on state observations. *Advances in Neural Information Processing Systems*, 33:21024–21037, 2020.

[154] Xuezhou Zhang, Yuzhe Ma, Adish Singla, and Xiaojin Zhu. Adaptive reward-poisoning attacks against reinforcement learning, 2020.

[155] Amirhossein Zolfagharian, Manel Abdellatif, Lionel Briand, and Ramesh S. Safety monitoring of deep reinforcement learning agents. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 286–287, New York, NY, USA, 2024. Association for Computing Machinery.

[156] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C Briand, Mojtaba Bagherzadeh, and S Ramesh. A search-based testing approach for deep reinforcement learning agents. *IEEE Transactions on Software Engineering*, 2023.

[157] Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, and Ramesh S. SMARLA: A Safety Monitoring Approach for Deep Reinforcement Learning Agents . *IEEE Transactions on Software Engineering*, (01):1–25, November 5555.