A Search-Based Testing Approach for Deep Reinforcement Learning Agents

Amirhossein Zolfagharian, Manel Abdellatif, Lionel C. Briand, *Fellow, IEEE*, Mojtaba Bagherzadeh, and Ramesh S

***Abstract*—Deep Reinforcement Learning (DRL) algorithms have been increasingly employed during the last decade to solve various decision-making problems such as autonomous driving, trading decisions, and robotics. However, these algorithms have faced great challenges when deployed in safety-critical environments since they often exhibit erroneous behaviors that can lead to potentially critical errors. 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. This raises the question of how we can efficiently test DRL policies to ensure their correctness and adherence to safety requirements. Most existing works on testing DRL agents use adversarial attacks that perturb states or actions of the agent. However, such attacks often lead to unrealistic states of the environment. Furthermore, their main goal is to test the robustness of DRL agents rather than testing the compliance of the agents' policies with respect to requirements. Due to the huge state space of DRL environments, the high cost of test execution, and the black-box nature of DRL algorithms, exhaustive testing of DRL agents is impossible. In this paper, we propose a Search-based Testing Approach of Reinforcement Learning Agents (STARLA) to test the policy of a DRL agent by effectively searching for failing executions of the agent within a limited testing budget. We rely on machine learning models and a dedicated genetic algorithm to narrow the search toward faulty episodes (i.e., sequences of states and actions produced by the DRL agent). We apply STARLA on Deep-Q-Learning agents trained on two different RL problems widely used as benchmarks and show that STARLA significantly outperforms Random Testing by detecting more faults related to the agent's policy. We also investigate how to extract rules that characterize faulty episodes of the DRL agent using our search results. Such rules can be used to understand the conditions under which the agent fails and thus assess the risks of deploying it.**

***Index Terms*—Genetic algorithm, machine learning, reinforcement learning, state abstraction, testing.**

I. Introduction

Reinforcement Learning (RL) algorithms have seen tremendous research advances in recent years, both from a theoretical standpoint and in their applications to solve real-world problems. Reinforcement Learning [1] trains an agent to make a sequence of decisions to reach a final goal and is therefore a technique gaining increased interest in many application contexts, such as autonomous driving and robotics. Deep Reinforcement Learning (DRL) techniques [2], [3], [4], a branch of deep learning where RL policies are learned using Deep Neural Networks (DNNs), have gained attention in recent years. However, like DNN components, their application in production environments requires effective and systematic testing, especially when used in safety-critical applications. For instance, 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 [5].

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) [6] 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.

Furthermore, in a DRL context, two types of faults can be defined: functional and reward faults. The former happens when an RL agent takes an action that leads to an unsafe state (e.g., a driverless car does not stop at a stop sign). The latter occurs when an agent does not reach the desired reward (e.g., when a

Manuscript received 15 June 2022; revised 6 April 2023; accepted 15 April 2023. Date of publication 25 April 2023; date of current version 18 July 2023. This work was supported by a research Grant from General Motors as well as the Canada Research Chair and Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC). Recommended for acceptance by D. Lo. *(Corresponding author: Amirhossein Zolfagharian.)*

Amirhossein Zolfagharian is with the School of Electrical Engineering and Computer Science (EECS), University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: A.zlf@uottawa.ca).

Manel Abdellatif is with the Software and Information Technology Engineering Department, École de Technologie Supérieure, Montreal, QC H3C 1K3, Canada, and also with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: Manel.abdellatif@etsmtl.ca).

Lionel C. Briand is with the School of Electrical Engineering and Computer Science (EECS), University of Ottawa, Ottawa, ON K1N 6N5, Canada, and also with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, 4365 Esch-sur-Alzette, Luxembourg (e-mail: Lbriand@uottawa.ca).

Mojtaba Bagherzadeh is with Cisco, Ottawa, ON K1N 6N5, Canada, and also with the School of EECS, University of Ottawa, Ottawa, ON K1N 6N5, Canada (e-mail: Mbagherz@cisco.com).

Ramesh S is with the Department of Research and Development, General Motors, Warren, MI 48092 USA (e-mail: Ramesh.s@gm.com).

Digital Object Identifier 10.1109/TSE.2023.3269804

driverless car reaches its destination an hour late). Functional and reward faults are often in tension, as we can obtain a high reward while observing a functional fault. For example, an unsafe action can help the agent reach the desired state faster (e.g., when not stopping at stop signs, the driverless car may reach its destination sooner). This makes the detection of such types of faults challenging, especially when the agent is well-trained and failures are rare. We should note 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 paper.

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 [7], [8]. 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 [9], [10], [11], [12], [13], [14], [15], 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 [11], [13], [15], [16]. 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 [17].

Most existing works on testing DRL agents are based on adversarial attacks that aim to perturb states of the DRL environment [18]. 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 [11]. 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 paper, 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 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 [19]. We rely on state abstraction techniques [20], [21] 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 [22]. We show that our testing approach outperforms Random Testing, as we find significantly more faults.

Overall, the main contributions of our paper 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 paper is structured as follows. Section II presents the required background and establishes the context of our research. Section III describes our research problem. Section IV presents our testing approach. Section V reports our empirical evaluation and results. Section VI discusses the practical implications of our results. Section VII analyzes the threats to validity. Finally, Sections VIII and IX contrast our work with related work and conclude the paper, respectively.

II. 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.

A. 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

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

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 [6] $\langle S, A, T, R, \gamma \rangle$ where S and A denote a set of possible states and actions accordingly, $T : S \times A \times S \to [0, 1]$ 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, $R : S \times A \to [0, R_{max}]$ is a reward function that determines the immediate reward for a pair of an action and a state, and $\gamma \in [0, 1]$ is the discount factor indicating the difference of short-term and long-term reward [23].

The solution of an MDP is a policy $\pi : S \to A$ 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 paper and is left for future work.

B. 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 [20], [21]. State abstraction can be defined as a mapping from an original state $s \in S$ to an abstract state $s^\phi \in S^\phi$

$$\phi : S \to S^\phi \tag{1}$$

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 [24], [25]:

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 [21]. 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:

$$\phi_d(s_1) = \phi_d(s_2) \equiv \forall a \in A : \left\lceil \frac{Q^*(s_1, a)}{d} \right\rceil = \left\lceil \frac{Q^*(s_2, a)}{d} \right\rceil \quad (2)$$

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.

III. Problem Definition

In this paper, 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.

A. RL Agent Testing Challenges

Since DRL techniques use DNNs, they inherit the advantages and drawbacks of DNNs, making their testing challenging and time-consuming [26], [27], [28], [29]. 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 [7], [30].
- *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 [13], [30], [31].
- *Focus on adversarial attacks:* Most existing works on testing DRL agents use adversarial attacks that are focused on perturbing states [14]. However, such attacks lead to unrealistic states and episodes [32]. The main goal of such attacks is to test the robustness of RL policies rather than the agents' functionality [13], [33], [34].

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.

B. 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 [36], [37] (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 [38], [39].

IV. 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 [40]. 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.

A. 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 IV-C.
- *Operators:* Genetic operators include crossover, mutation, and selection [41], 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 IV-E2.

  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 IV-E3.
- *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 [42] function that we use to compare episodes and select the fittest ones. As further explained in Section IV-D, 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.

Algorithm 1: High-Level Genetic Algorithm.

```
Input: A set of episodes ($P$) as the initial population,
       solution archive $\alpha$
Output: The updated archive $\alpha$ containing faulty
        episodes
1  $gen \leftarrow 0$
2  $\alpha \leftarrow \emptyset$
3  while $FitSatisfied = False \ \& \ gen \leq g$ do
4      $Fitness = Fit(P)$
5      $P_{new} \leftarrow \emptyset$
6      $rand = Random(0, 1)$
7      if $rand < c$ then
8          $t_1, t_2 \leftarrow Cross(P)$
9          $P_{new} \leftarrow P_{new} \cup \{t_1, t_2\}$
10     $t_m \leftarrow Mut(P \cup \{t_1, t_2\}, m)$
11     $P_{new} \leftarrow P_{new} \cup t_m$
12     $Fitness = Fit(P_{new})$
13     $Update(FitSatisfied)$
14     $Update(\alpha)$
15     $P \leftarrow Select(P, P_{new})$
16     $gen \leftarrow gen + 1$
17 return $\alpha$
```

B. Overview of the Approach

As depicted in Fig. 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).

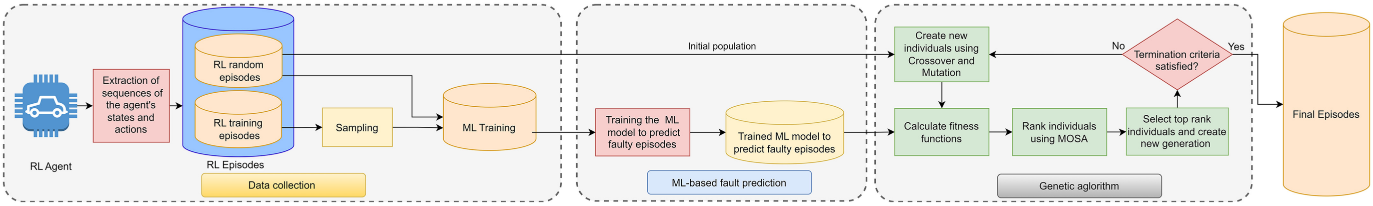

RL Agent
Extraction of sequences of the agent's states and actions
RL random episodes
RL training episodes
RL Episodes
Sampling
ML Training
Data collection
Initial population
Training the ML model to predict faulty episodes
Trained ML model to predict faulty episodes
ML-based fault prediction
Create new individuals using Crossover and Mutation
No
Termination criteria satisfied?
Yes
Calculate fitness functions
Rank individuals using MOSA
Select top rank individuals and create new generation
Genetic aglorithm
Final Episodes

Fig. 1. Overview of STARLA.

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 [43]. In the following, we discuss each step of the search in detail.

C. 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.

D. 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 III-A 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 IV-D1, IV-D2, and IV-D3). Specifically, we explain how each fitness function contributes to guiding the search toward faulty episodes. We then explain in Sections IV-D4, IV-D5, IV-D6, and IV-D7 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 IV-D8.

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.

$$f_1(e) = r'_e \tag{3}$$

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 IV-E) 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) 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:

$$f_2(e) = 1 - Pr_f[e] \tag{4}$$

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 IV-D4. 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.

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 [44].

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 (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.

$$f_3(e) = \frac{\sum_{i=1}^{|e|} \left(P_r(a_i|s_i) - \max\limits_{a_j \in A_i \ \& \ j \neq i} P_r(a_j|s_i)\right)}{|e|} \tag{5}$$

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 IV-E2). 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.

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 [45], [46]. 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 the paper.

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.

$$P_r[E_i] = \frac{i}{\Sigma_{j=1}^{n} j} \tag{6}$$

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

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

Algorithm 2: High-Level Algorithm to Create Abstract States.

```
Input: Set of states $S$, abstraction level $d$
Output: Abstract states $S^{\phi_d}$
$S^{\phi_d} \leftarrow \emptyset$
$len \leftarrow 0$
for $s_i \in S$ do
    if $S^{\phi_d} = \emptyset$ then
        $len \leftarrow len + 1$
        append $s_i$ to $S_1^{\phi_d}$
    $Found = False$
    for $j\ in\ range(1, len)$ do
        if $\phi(s_i) = S_j^{\phi_d}$ then
            append $s_i$ to $S_j^{\phi_d}$
            $Found = True$
    if $Found = False$ then
        $len \leftarrow len + 1$
        append $s_i$ to $S_{len}^{\phi_d}$
return $S^{\phi_d}$
```

created with random executions of the RL agent (Section IV-C), thus minimizing the number of simulations (and therefore the testing budget).

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 II-B.

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 II-B. 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.

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.

$$\begin{array}{cccccc} & S_1^\phi & S_2^\phi & \cdots & S_i^\phi & \cdots & S_n^\phi \\ episode_i & 0 & 1 & \cdots & 0 & \cdots & 1 \end{array}$$

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.

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:

$$\min_{x \in E} \quad F(x) = (f_1(x), f_2(x), f_3(x)) \tag{7}$$

where E is the set of possible episodes in the search space, $F : E \to \mathbf{R}^3$ 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.

E. 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.

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 [47]. 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 II-B). 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.

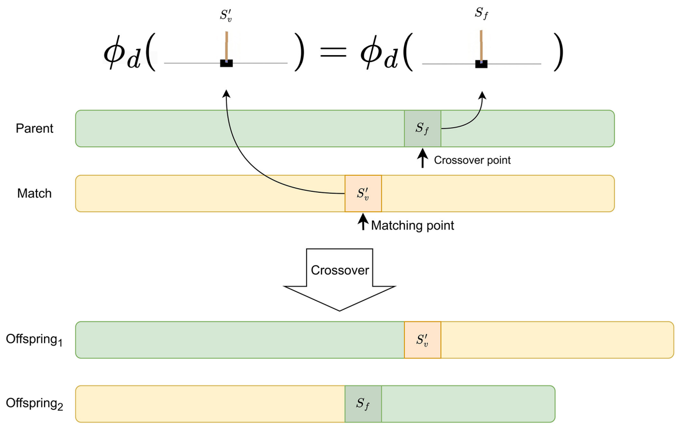


Fig. 2. The crossover operator.

The crossover process is illustrated in Fig. 2. Let us assume that the selected parent is as follows:

$$Parent = [(s_1, a_1), (s_2, a_2), \ldots, (s_{f-1}, a_{f-1}), \boldsymbol{(s_f, a_f)}, (s_{f+1}, a_{f+1}), \ldots, (s_m, a_m)] \tag{8}$$

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.

$$Match = [(s_1', a_1'), (s_2', a_2'), \ldots, (s_{v-1}', a_{v-1}'), \boldsymbol{(s_v', a_v')}, \ldots, (s_n', a_n')] \tag{9}$$

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:

$$Offspring_1 = [(s_1, a_1), \ldots, (s_{f-1}, a_{f-1}), (s_v', a_v'), \ldots, (s_n', a_n')] \tag{10}$$

$$Offspring_2 = [(s_1', a_1'), \ldots, (s_{v-1}', a_{v-1}'), (s_f, a_f), \ldots, (s_m, a_m)] \tag{11}$$

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 II-B), 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 IV-F.

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) 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 [10], [48]. 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 V-B), 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:

$$e_h = [(s_1, a_1), \ldots, \boldsymbol{(s_c, a_c)}, (s_{c+1}, a_{c+1}), \ldots, (s_m, a_m)] \tag{12}$$

$$e_h^m = [(s_1, a_1), \ldots, \boldsymbol{(s_c^t, a_c^t)}, \ldots] \tag{13}$$

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

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) [49], or SPEA-II [50] because they show poor performance in problems with many objectives [51] and they target Pareto front optimization. Additionally, to build a generalizable and extensible testing approach, we opt for the Many Objective Sorting Algorithm (MOSA) [43] 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 [52]. Although traditional dominance-based algorithms like NSGA-II and SPEA-II [50] show poor performance in problems with many objectives [51], 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 IV-D. 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.

F. 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 V-B1, have a cosine distance lower than 0.25. Similarly, we observed that 99% of the deviations in the *Mountain Car* environment (see Section V-B2 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.

V. Empirical Evaluation

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

A. Research Questions

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

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) 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

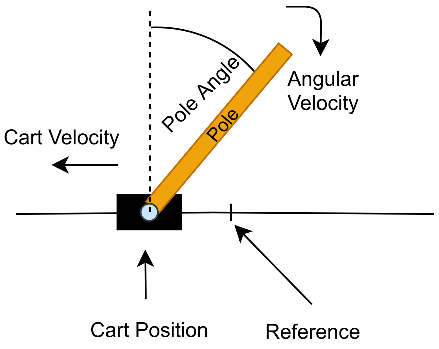


Fig. 3. *Cart-Pole* balancing problem.

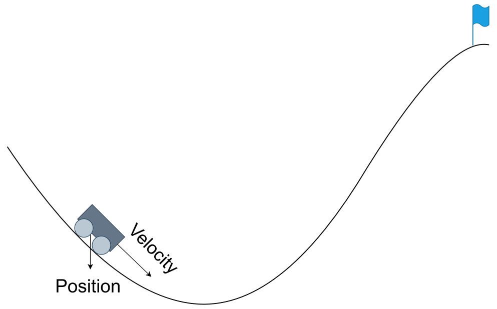


Fig. 4. *Mountain Car* problem.

an ML classifier as a fitness function to guide our search toward finding faulty episodes.

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.

B. 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 [53], [54], [55]. 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.

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 Fig. 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) 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 Fig. 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.

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* (Fig. 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.

C. Implementation

We used Google Colab and stable baselines [56] to implement RL agents for both case studies (see Section V-B). Our RL agents are based on a DQN policy network [57] using standard setting of stable baselines (i.e., Double Q-learning [4], and dueling DQN [58]). 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] gymlibrary.dev/environments/classic_control/cart_pole/
[3] gymlibrary.dev/environments/classic_control/mountain_car/

D. Evaluation and Results

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 [59]. 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 [59], [60]. 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.

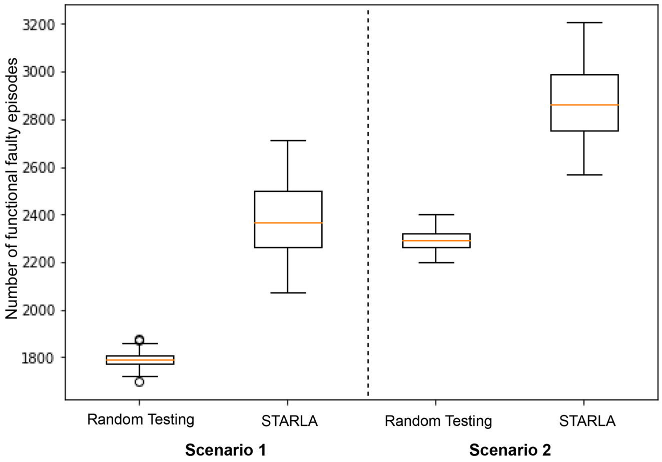


Fig. 5. Number of functional-faulty episodes generated with STARLA compared to Random Testing in the *Cart-Pole* problem.

Because of randomness in our search approach and its significant execution time (Section V-C), 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 study$M_c = 128$ and in *Mountain Car* $M_m = 139$) over the 20 runs. 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 Figs. 5 and 6, 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.

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.

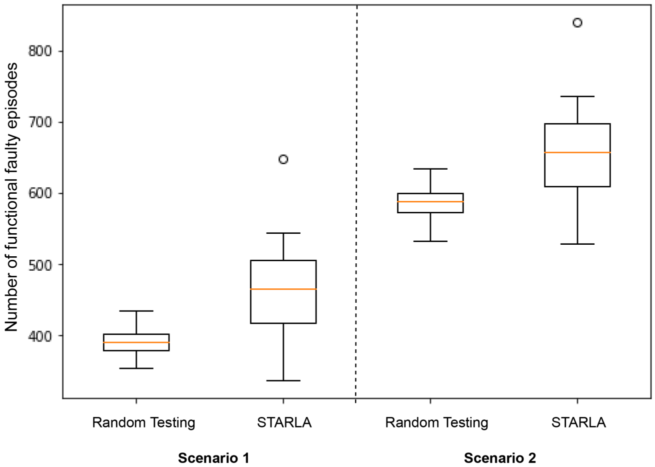


Fig. 6. Number of functional-faulty episodes generated with STARLA compared to Random Testing in the *Mountain Car* problem.

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 [61] 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)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 IV-C). 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.

TABLE I
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

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 II
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

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%

In both case studies, we trained *Random Forest* models 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 IV-D6, we used the Q^*-irrelevance state abstraction technique [21] to reduce the state space. We experimented with several values for the abstraction level *d* (Section II-B) and reported the prediction results in terms of precision, recall, and accuracy in Tables I and II. 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 I, the best results for the prediction of functional faults yield a precision and a recall of 97% in the *Cart-Pole* problem. Similarly, a precision and a recall of 99% was achieved in the *Mountain Car* problem as depicted in Table II.

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.

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 IV-D7). 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).

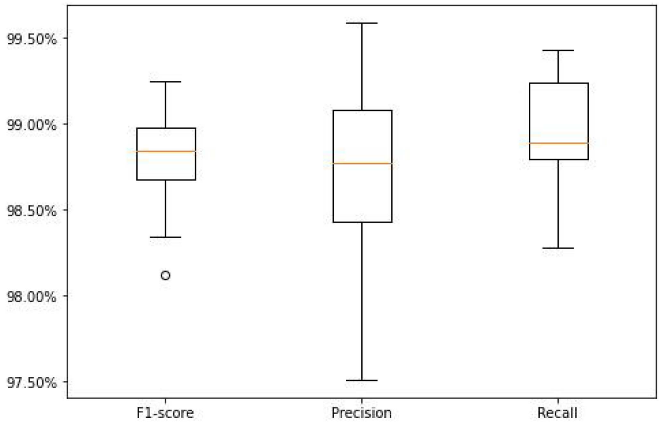


Fig. 7. Accuracy of rules predicting faults in *Cart-Pole*.

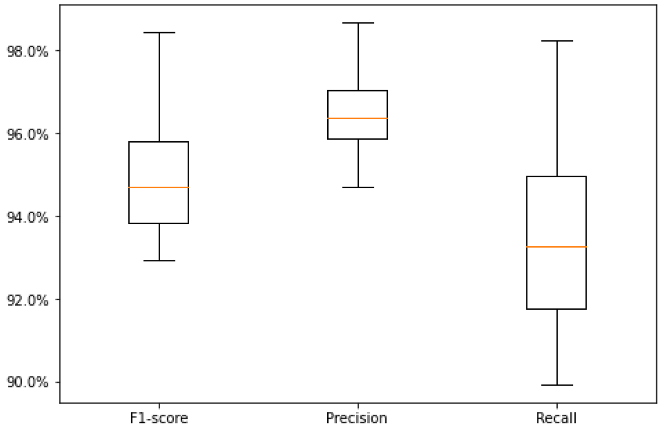


Fig. 8. Accuracy of rules predicting faults in *Mountain Car*.

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 Figs. 7 and 8.

As shown in Figs. 7 and 8, 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.

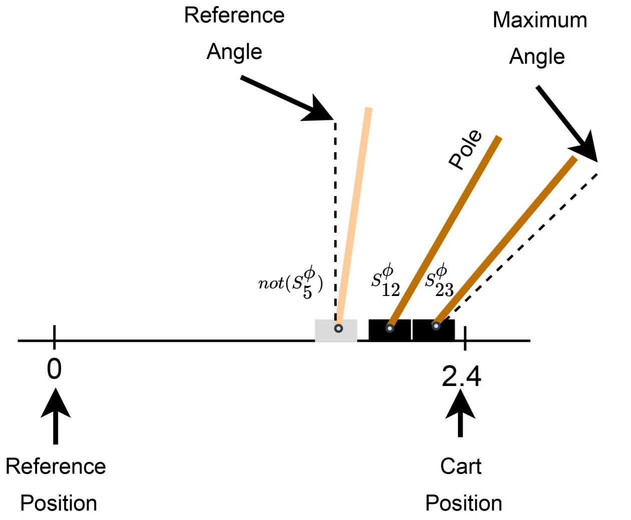


Fig. 9. Interpretation of Rule *R1*. Each cart represents one abstract state. The gray cart depicts the state of the system in abstract state S^{ϕ}_{5}, which should be absent in the episode. The black carts represent the presence of abstract states S^{ϕ}_{12} and S^{ϕ}_{23}, respectively. Having both latter states appearing in an episode and not having the state on the left is highly likely to lead to a fault.

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

$$\textbf{R1}: not(S^{\phi}_{5})\ and\ S^{\phi}_{12}\ and\ S^{\phi}_{23}$$

where rule *R1* states that an episode is faulty if there are no concrete states in the episode that belong to abstract state S^{ϕ}_{5} and we have at least two concrete states matching abstract state S^{ϕ}_{12} and S^{ϕ}_{23}, 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^{ϕ}_{12} and S^{ϕ}_{23}, respectively). For abstract states that the rule specifies as absent (abstract state S^{ϕ}_{5} 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 Fig. 9. Our analysis shows that the presence of abstract states S^{ϕ}_{12} and S^{ϕ}_{23} 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^{ϕ}_{5} 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. 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.

VI. Discussions

In this paper 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 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 IV-E1, 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 IV-F. 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 V-B, 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 III-A, 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 IV-E3).

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 IV-C, 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 [62], entropy measure [63], and hamming distance [64].

Rules Characterizing Faulty Episodes: In this paper, 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 paper 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.

VII. 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 paper. However, we mitigated this threat by using widely studied RL tasks which are considered as valid benchmark problems in many RL-related studies [55], [65], [66], [67]. 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.

VIII. 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. [68] presented a search-based testing method for RL agents. The method relies on a backtracking-based, depth-first search algorithm [69] 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. [11] 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. [7] studied the reliability of neuron coverage [70] 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 [71]. 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 [13], [14], [15]. For example, Ilahi et al. [13] 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. [72] 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. [14] 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 [57], TRPO [73], and A3C [74], trained on Atari games [75]. They showed that adversarial attacks can significantly degrade the performance of the agent, even with small, imperceptible perturbations.

Pan et al. [15] 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 [76], [77]. Moreover, they studied the robustness improvement of the agent trained with adversarial training.

Other existing approaches in the literature [16], [55], [78], [79], [80], [81], [82], [83], [84] have proposed adversarial training techniques to increase the robustness of RL agents to adversarial attacks. For example, Pattanaik et al. [55] 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. [16] 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.

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.

IX. Conclusion

In this paper, 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.

Acknowledgment

We express our gratitude to Ul Haq et al. [85] for generously sharing the implementation of their search algorithm, which greatly facilitated our study.

References

[1] M. A. Wiering and M. Van Otterlo, "Reinforcement learning," *Adapt. Learn. Optim.*, vol. 12, no. 3, 2012, Art. no. 729.

[2] M. Sewak, *Deep Reinforcement Learning*. Berlin, Germany: Springer, 2019.

[3] C. Lei, "Deep reinforcement learning," in *Deep Learning and Practice With MindSpore*. Berlin, Germany: Springer, 2021, pp. 217–243.

[4] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. AAAI Conf. Artif. Intell.*, 2016, vol. 30, no. 1.

[5] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," 2019, *arXiv: 1904.12901*.

[6] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. New York, NY, USA: Wiley, 1994.

[7] M. Trujillo, M. Linares-Vásquez, C. Escobar-Velásquez, I. Dusparic, and N. Cardozo, "Does neuron coverage matter for deep reinforcement learning? a preliminary study," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng. Workshops*, 2020, pp. 215–220.

[8] J. Lin, L. Xu, Y. Liu, and X. Zhang, "Black-box adversarial sample generation based on differential evolution," *J. Syst. Softw.*, vol. 170, 2020, Art. no. 110767.

[9] Z. Aghababaeyan, M. Abdellatif, L. Briand, R. S, and M. Bagherzadeh, "Black-box testing of deep neural networks through test case diversity," *IEEE Trans. Softw. Eng.*, pp. 215–220, 2023. [Online]. Available: http://dx.doi.org/10.1109/TSE.2023.3243522

[10] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 303–314.

[11] A. Nikanjam, M. M. Morovati, F. Khomh, and H. Ben Braiek, "Faults in deep reinforcement learning programs: A taxonomy and a detection approach," *Automated Softw. Eng.*, vol. 29, no. 1, pp. 1–32, 2022.

[12] X. Xie et al., "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 146–157.

[13] I. Ilahi et al., "Challenges and countermeasures for adversarial attacks on deep reinforcement learning," *IEEE Trans. Artif. Intell.*, vol. 3, no. 2, pp. 90–109, 2021.

[14] S. H. Huang, N. Papernot, I. J. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," 2017, *arXiv: 1702.02284*.

[15] A. Pan, Y. Lee, H. Zhang, Y. Chen, and Y. Shi, "Improving robustness of reinforcement learning for power system control with adversarial training," 2021, *arXiv:2110.08956*.

[16] K. L. Tan et al., "Robustifying reinforcement learning agents via action space adversarial training," in *Proc. IEEE Amer. Control Conf.*, 2020, pp. 3959–3964.

[17] T. Byun, S. Rayadurgam, and M. P. Heimdahl, "Black-box testing of deep neural networks," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng.*, 2021, pp. 309–320.

[18] Y.-C. Lin et al., "Tactics of adversarial attack on deep reinforcement learning agents," 2019.

[19] J. H. Holland, "Genetic algorithms," *Sci. Amer.*, vol. 267, no. 1, pp. 66–73, 1992.

[20] R. Akrour, F. Veiga, J. Peters, and G. Neumann, "Regularizing reinforcement learning with state abstraction," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2018, pp. 534–539.

[21] D. Abel, D. Arumugam, L. Lehnert, and M. Littman, "State abstractions for lifelong reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 10–19.

[22] Gym library, 2020. [Online]. Available: https://github.com/openai/gym

[23] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[24] N. Jiang, "Notes on state abstractions," 2018. [Online]. Available: http://nanjiang.web.engr.illinois.edu/files/cs598/note4.pdf

[25] L. Li, T. J. Walsh, and M. L. Littman, "Towards a unified theory of state abstraction for MDPs," *Proc. Int. Symp. Artif. Intell. Math.*, vol. 4, no. 5, pp. 1–9, 2006.

[26] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Testing deep neural networks," 2018, *arXiv: 1803.04792*.

[27] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: Testing and debugging deep neural networks," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: Companion Proc.*, 2019, pp. 111–114.

[28] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proc. IEEE/ACM 33rd Int. Conf. Automat. Softw. Eng.*, 2018, pp. 109–119.

[29] L. Ma et al., "DeepGauge: Multi-granularity testing criteria for deep learning systems," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 120–131, doi: 10.1145/3238147.3238202.

[30] G. Dulac-Arnold et al., "Challenges of real-world reinforcement learning: Definitions, benchmarks and analysis," *Mach. Learn.*, vol. 110, no. 9, pp. 2419–2468, 2021.

[31] T. Chaffre, J. Moras, A. Chan-Hon-Tong, and J. Marzat, "Sim-to-real transfer with incremental environment complexity for reinforcement learning of depth-based robot navigation," 2020, *arXiv: 2004.14684*.

[32] S. Ghamizi et al., "Search-based adversarial testing and improvement of constrained credit scoring systems," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1089–1100.

[33] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 39–57.

[34] H. Zhang et al., "Robust deep reinforcement learning against adversarial perturbations on state observations," in *Proc. Adv. Neural Inf. Process. Syst.*, 2020, pp. 21024–21037.

[35] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," 2019, *arXiv:1904.12901*.

[36] X. Zhang, Y. Ma, A. Singla, and X. Zhu, "Adaptive reward-poisoning attacks against reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 11 225–11 234.

[37] A. Rakhsha, X. Zhang, X. Zhu, and A. Singla, "Reward poisoning in reinforcement learning: Attacks against unknown learners in unknown environments," 2021, *arXiv:2102.08492*.

[38] P. Swazinna, S. Udluft, D. Hein, and T. Runkler, "Comparing model-free and model-based algorithms for offline reinforcement learning," 2022, *arXiv:2201.05433*.

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

[40] I. Alsmadi, "Using genetic algorithms for test case generation and selection optimization," in *Proc. IEEE Can. Conf. Elect. Comput. Eng.*, 2010, pp. 1–4.

[41] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, 1994.

[42] T. Murata et al., "MOGA: Multi-objective genetic algorithms," in *Proc. IEEE Int. Conf. Evol. Comput.*, Piscataway, NJ, USA, 1995, pp. 289–294.

[43] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proc. IEEE 8th Int. Conf. Softw. Testing Verification Validation*, 2015, pp. 1–10.

[44] Y.-C. Lin, Z.-W. Hong, Y.-H. Liao, M.-L. Shih, M.-Y. Liu, and M. Sun, "Tactics of adversarial attack on deep reinforcement learning agents," in *Proc. 26th Int. Joint Conf. Artif. Intell.*, 2017, pp. 3756–3762.

[45] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001, doi: 10.1023/A:1010933404324.

[46] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 2001.

[47] B. L. Miller and D. E. Goldberg, "Genetic algorithms, selection schemes, and the varying effects of noise," *Evol. Comput.*, vol. 4, no. 2, pp. 113–131, Jun. 1996, doi: 10.1162/evco.1996.4.2.113.

[48] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, pp. 137–145, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3361566

[49] D. Wang, D. Tan, and L. Liu, "Particle swarm optimization algorithm: An overview," *Soft Comput.*, vol. 22, pp. 387–408, 2018.

[50] V. J. Amuso and J. Enslin, "The strength pareto evolutionary algorithm 2 (SPEA2) applied to simultaneous multi- mission waveform design," in *Proc. IEEE Int. Waveform Diversity Des. Conf.*, 2007, pp. 407–417.

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

[52] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[53] S. Nagendra, N. Podila, R. Ugarakhod, and K. George, "Comparison of reinforcement learning algorithms applied to the cart-pole problem," in *Proc. IEEE Int. Conf. Adv. Comput. Commun. Inform.*, 2017, pp. 26–32.

[54] C. Aguilar-Ibáñez, J. Mendoza-Mendoza, and J. Dávila, "Stabilization of the cart pole system: By sliding mode control," *Nonlinear Dyn.*, vol. 78, no. 4, pp. 2769–2777, 2014.

[55] A. Pattanaik, Z. Tang, S. Liu, G. Bommannan, and G. Chowdhary, "Robust deep reinforcement learning with adversarial attacks," in *Proc. 17th Int. Conf. Auton. Agents MultiAgent Syst.*, 2018, pp. 2040–2042.

[56] A. Hill et al., "Stable baselines," 2018. [Online]. Available: https://github.com/hill-a/stable-baselines

[57] V. Mnih et al., "Playing atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

[58] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1995–2003.

[59] A. Marco et al., "Virtual versus real: Trading off simulations and physical experiments in reinforcement learning with Bayesian optimization," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2017, pp. 1557–1563.

[60] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *Int. J. Robot. Res.*, vol. 32, no. 11, pp. 1238–1274, 2013.

[61] P. E. McKnight and J. Najab, "Mann-whitney U test," *Corsini Encyclopedia Psychol.*, pp. 1–1, 2010. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0524

[62] A. Kulesza and B. Taskar, "Determinantal point processes for machine learning," 2012, *arXiv:1207.6083*.

[63] N. Albunian, G. Fraser, and D. Sudholt, "Measuring and maintaining population diversity in search-based unit test generation," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2020, pp. 153–168.

[64] R. W. Morrison and K. A. D. Jong, "Measurement of population diversity," in *Proc. Int. Conf. Artif. Evol.*, 2001, pp. 31–41.

[65] V. Behzadan and W. H. Hsu, "Adversarial exploitation of policy imitation," 2019, *arXiv: 1906.01121*.

[66] V. Behzadan and W. Hsu, "Sequential triggers for watermarking of deep reinforcement learning policies," 2019, *arXiv: 1906.01126*.

[67] K. Chen, T. Zhang, X. Xie, and Y. Liu, "Stealing deep reinforcement learning models for fun and profit," 2006, *arXiv: 2006.05032*.

[68] M. Tappler, F. C. Cordoba, B. Aichernig, and B. Könighofer, "Search-based testing of reinforcement learning," in *Proc. 31st Int. Joint Conf. Artif. Intell.*, 2022, pp. 503–510.

[69] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[70] L. Ma et al., "DeepGauge: Multi-granularity testing criteria for deep learning systems," in *Proc. IEEE/ACM 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 120–131.

[71] R. S. Sutton, *Andrew G Barto Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[72] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," 2016, *arXiv:1610.08401*.

[73] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1889–1897.

[74] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.

[75] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res.*, vol. 47, pp. 253–279, Jun. 2013, doi: 10.1613/jair.3912.

[76] A. Marot et al., "L2rpn: Learning to run a power network in a sustainable world neurips2020 challenge design," Réseau de Transport d'Électricité, Paris, France, White Paper, 2020.

[77] L. Omnes, A. Marot, and B. Donnot, "Adversarial training for a continuous robustness control problem in power systems," in *Proc. IEEE Madrid PowerTech.*, 2021, pp. 1–6.

[78] J. Kos and D. Song, "Delving into adversarial attacks on deep policies," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, Apr. 24–26, 2017, pp. 1–6. [Online]. Available: https://openreview.net/forum?id=BJcib5mFe

[79] X. Y. Lee, Y. Esfandiari, K. L. Tan, and S. Sarkar, "Query-based targeted action-space adversarial policies on deep reinforcement learning agents," in *Proc. IEEE/ACM 12th Int. Conf. Cyber-Phys. Syst.*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 87–97, doi: 10.1145/3450267.3450537.

[80] E. Vinitsky, Y. Du, K. Parvate, K. Jang, P. Abbeel, and A. M. Bayen, "Robust reinforcement learning using adversarial populations," 2020, *arXiv: 2008.01825*.

[81] K. L. Tan, Y. Esfandiari, X. Y. Lee Aakanksha, and S. Sarkar, "Robustifying reinforcement learning agents via action space adversarial training," 2007, *arXiv: 2007.07176*.

[82] V. Behzadan and A. Munir, "Mitigation of policy manipulation attacks on deep q-networks with parameter-space noise," 2018, *arXiv: 1806.02190*.

[83] V. Behzadan and M. Arslan, "Whatever does not kill deep reinforcement learning, makes it stronger," 2017, *arXiv: 1712.09344*.

[84] Y. Han et al., "Reinforcement learning for autonomous defence in software-defined networking," 2018, *arXiv: 1808.05770*.

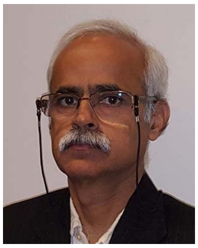
[85] F. U. Haq, D. Shin, L. C. Briand, T. Stifter, and J. Wang, "Automatic test suite generation for key-points detection dnns using many-objective search (experience paper)," in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, New York, NY, USA, 2021, pp. 91–102, doi: 10.1145/3460319.3464802.

Amirhossein Zolfagharian is a member of Nanda Lab and is currently working toward the PhD degree in the School of EECS, University of Ottawa. He gained valuable practical experience from his internship with General Motors' research and development lab in the United States. Throughout his academic career, he has been the recipient of several academic awards, including a PhD admission scholarship, an international doctoral scholarship from the University of Ottawa, and an honorable award for admission to the master's program in computer science with Amirkabir University of Technology. In 2017, he was ranked as the 4th-best student among all computer science students at Amirkabir University, placing him in the top 5% of his class in GPA. His research interests primarily focus on machine learning, empirical software engineering, and testing and verification of RL-based systems.

Manel Abdellatif received the PhD degree in computer science from Polytechnique Montréal, Canada. She is a faculty member at École de Technologie Supérieure. She was a postdoctoral fellow at the School of EECS, University of Ottawa. She has actively contributed to the research community, serving as a program committee member and a reviewer for numerous journals and conferences. Her research interests include testing machine learning-based systems, service computing, and empirical software engineering.

Lionel C. Briand (Fellow, IEEE) is a professor of software engineering and has shared appointments between (1) School of EECS, University of Ottawa, Canada and (2) SnT centre for Security, Reliability, and Trust, University of Luxembourg. He is the head of the SVV department at the SnT Centre and a Canada Research Chair in Intelligent Software Dependability and Compliance (Tier 1). He has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. He is a fellow of the IEEE and ACM. He was also granted the IEEE Computer Society Harlan Mills award (2012), the IEEE Reliability Society Engineer-of-the-year award (2013), and the ACM SIGSOFT Outstanding Research Award (2022) for his work on software testing and verification.

Mojtaba Bagherzadeh received the PhD degree in computer science from Queen's University, Canada, in 2019. He is a highly experienced software engineer with a proven track record of success in both industry and academia. He currently works as a software engineer with Cisco Systems and has previously worked as a software developer with IBM and a startup company. He has contributed to this research during his tenure as a postdoctoral researcher with the University of Ottawa. His research interests include testing and debugging machine learning-based systems, model-driven engineering, software testing, and empirical software engineering.



Ramesh S senior technical fellow, has been with General Motors Research and Development (R&D) for more than 15 years conducting and leading advanced research projects in the areas of model-based development of embedded systems and software, rigorous verification and validation, and more recently, AI/ML based systems. Prior to joining GM R&D, he was a full professor with the Department of Computer Science and Engineering, Indian Institute of Technology Bombay, India, where he co-founded a Centre for Formal Design and Verification of Software. He has published more than 125 research papers in international journals and conferences and authored many patents in the areas of modeling, analysis, and verification of embedded systems and software. He has been on the program committees of several international research conferences and on the editorial boards of journals. He leads an USCAR committee and serves as an expert in ISO and SAE committees to develop guidelines for AI/ML based systems.