Exploring Fault Parameter Space Using Reinforcement Learning-based Fault Injection

A Preprint

Mehrdad Moradi[1], **Bentley James Oakes**[1], **Mustafa Saraoglu**[2],
Andrey Morozov[2], Klaus Janschek[2], and Joachim Denil[1]
[1]University of Antwerp and Flanders Make vzw, Belgium
{Mehrdad.Moradi, Bentley.Oakes, Joachim.Denil}@uantwerpen.be
[2]Technische Universität Dresden
{Mustafa.Saraoglu, Andrey.Morozov, Klaus.Janschek}@tu-dresden.de

June 14, 2020

Abstract

Assessing the safety of complex Cyber-Physical Systems (CPS) is a challenge in any industry. Fault Injection (FI) is a proven technique for safety analysis and is recommended by the automotive safety standard ISO 26262. Traditional FI methods require a considerable amount of effort and cost as FI is applied late in the development cycle and is driven by manual effort or random algorithms. In this paper, we propose a Reinforcement Learning (RL) approach to explore the fault space and find critical faults. During the learning process, the RL agent injects and parameterizes faults in the system to cause catastrophic behavior. The fault space is explored based on a reward function that evaluates previous simulation results such that the RL technique tries to predict improved fault timing and values. In this paper, we apply our technique on an Adaptive Cruise Controller with sensor fusion and compare the proposed method with Monte Carlo-based fault injection. The proposed technique is more efficient in terms of fault coverage and time to find the first critical fault.

***K*eywords** Fault injection, reinforcement learning, safety assessment, cyber-physical systems, machine learning

1 Introduction

Cyber-physical systems (CPSs) are heterogeneous systems in which computer-based components interact with the physical components of the system. CPSs therefore involve multiple domains such as electronics, mechanics, computing, networking, etc [1]. Each component can consist of many modules and subsystems, which raises the complexity of the total system and increases the possibility of errors. Along with this heterogeneous complexity, CPSs must also perform in a dynamic, possibly real-time environment where the CPS must perform intended functions safely. For this purpose, vendors perform safety assessments on their products.

A safety assessment aims to assure that the system will perform the intended function properly in every operating scenario. Different methods have been proposed for safety assessment. Fault Injection (FI) is a well-known method which is based on experimenting and simulation. FI accelerates the occurrences of faults in the system in order to evaluate system behavior [2]. In FI, the fault space is induced by the three main properties of faults: *type*, *location* and *time*. The complexity of the system under the test may also lead to a system which is very sensitive to slight perturbations, expanding the fault space exponentially [3]. Therefore, it is crucial to efficiently explore the fault space to find and parameterize the most critical fault.

As the fault space could be extremely large (or infinite), exploring it with traditional FI is very inefficient in terms of time and effort. We require a methods to explore this space automatically and efficiently. However, it is unclear how to exploit domain knowledge to properly set-up these automatic searches. Our research question is about efficiently and (semi-) automatically finding critical fault amplitudes (referring to the value of the fault) and the time at which they should occur. We will strive to find an optimal fault amplitude and time pair that violate the safety specifications for a given system.

In this paper, our approach is to use a Reinforcement Learning (RL) algorithm for fault space exploration. RL is one of the main categories in Machine Learning (ML). The aim of using RL is to find these fault pairs by rewarding the set of actions that lead to the collection of most rewards, namely the most critical faults [4]. The RL algorithm will learn by dynamic interaction with the environment. The purpose of using RL in this dynamic environment is to understand the edge-case scenarios in automated driving.

In autonomous driving, safety is easy to verify given a fully-deterministic environment for many of the state-of-the-art systems. The challenge lies in fault detection mechanisms for the real-world, which are often the causes of fatalities in current autonomous driving systems. Therefore, the usage of RL makes it possible to reach solutions for highly complicated models. The entity that interacts with the environment to explore the fault space and seeks to maximize its rewards is called the *agent*. Here the term environment refers to everything outside of the agent and the fault injector, namely the system under the study.

We demonstrate our technique on an Adaptive Cruise Controller (ACC) with a sensor fusion model available from The Mathworks Inc. In our use case, the *ego vehicle* uses ACC to keep a safe distance away from the *lead vehicle* in front by adapting the acceleration of the ego vehicle. Our approach will inject faults in the velocity sensor which determines the relative velocity between the ego and lead vehicles. This velocity data is input for the ACC and so directly impacts the acceleration of the ego vehicle. Therefore, we will introduce faults into this sensor with the aim of finding critical fault parameters which provoke a catastrophic situation.

Our core contributions are the following: a) an accurate description of our application of RL for fault space exploring in the ACC example, b) a discussion of the *reward functions* employed, which guide the RL search, and c) lessons learned about this approach, including an examination of the limitations. The proposed approach explores the fault space and progress towards critical faults. It is therefore more efficient than random fault space exploration in terms of fault coverage and performance. We also briefly compare our approach to a randomized Monte Carlo approach.

This paper is organized as follows. In section 2, we provide some background knowledge and related works in the literature. Section 3 describe our motivating example, the ACC system. Our technique and methodology are explained in section 4 and the results are demonstrated in section 5. Section 6 describes challenges and lesson learned in the techniques, and section 7 concludes the paper and states future work.

2 Background and Related Works

In this section, we provide a brief introduction about fundamental topics discussed throughout this paper as well as related work in the state-of-the-art.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a type of Machine Learning (ML) algorithm which is used to tackle hard-to-solve control problems. It focuses on finding the best sequence of actions that generate the intended behavior. This section will briefly describe the three components of an RL approach: *environment*, *agent*, and *reward function*. Each component will then be further explained in context of our use case in section 4.3. The *environment* contains the model of the system, and the *agent* tries solves the problem by exploring, interacting with, and learn from the environment. Each simulation in RL is named an *episode*, and the agent in RL learns from the previous episode to maximize the rewards by changing its actions for the next episode [5]. The biggest difference between RL and other ML methods is that RL deals with a changing dynamic environment, whereas supervised and unsupervised ML deals with static data sets.

The agent interact with the environment with the *observation* and the *action* signal as shown in Fig. 1. The observation signal indicates the state of the environment and the action signal modifies or controls the environment. The agent's action is based on the amplitude of the *reward function*. The reward function guides the agent towards exploring or exploiting the search space in the environment [6]. The reward function must be defined in a way that directs the agent toward the maximum reward and the intended state. The observation-action-reward cycle continues until finishing the space exploration.

The states of the system are the observations for the agent, but how they change is unknown. This refers to the *model-free* setting, where the model of the system that the agent interacts is unknown. The agent gets feedback solely through the reward function(s). This may seem like an oversimplification, but the ability to use such a setting is the strength of RL, as to solve for every and all possible states and actions would not be feasible. Furthermore, it may even not be possible for models with incomplete knowledge or for systems that have a highly unpredictable nature. Inside the

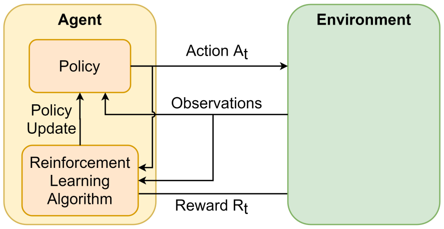


Figure 1: RL agents

agent, there is a *policy* and an *RL algorithm*. The policy maps the observation signals to actions, and the RL algorithm tries to improve the policy in a way that the agent obtains a higher reward value. There are two main algorithms for the agent. The first type is *policy-based*, where the algorithm knows in which way it must explore the space. The second type is *value-based*, where the algorithm must decide about the direction of exploring the space. For using agents, there are two main approaches. The first is a *model-free* approach, where the agent does not need to know anything about the system. The second approach is a *model-based* technique, where we provide information from the system for the agent, so it does not need to explore the whole space.

2.2 Fault Injection

Fault injection (FI) is a testing method that stresses systems in unusual ways by injecting faults (manipulating the values of the signals) to evaluate system behaviors and system dependability under the uncertainties and rare events that create errors in any system. Faults can be defined as either perturbance from a normal distribution or clearly defined disturbances. However, most faults are piece-wise, nonlinear, and unpredictable, such as saturation, bit-flips as hardware faults, or communications faults such as disconnections in data transmissions. Note that a linear fault model would not pose much of a great challenge because of its predictable nature. Error detection and disturbance rejection mechanisms can tolerate such faults straightforwardly. However, that is not the case in most systems involving various software and hardware components. A nonlinear fault would have different erroneous values, disregarding a linear correlation with the actual value of the signal. Therefore, the most viable verification solution is to apply an experimental approach, i.e., to use simulation-based testing for FI [2, 3, 7].

FI has traditionally been applied to software and hardware prototypes, testing the fault-tolerance capabilities of the designed product and validating and verifying [8]. Faults are mainly injected randomly or based on user experience, and the fault's parameters are selected from a uniform or predefined distribution. Faults have three main parameters as *type*, *location* and *time*, and the fault type can be divided into many different categories, such as *data latency*, or *data change*. Faults could also have a transient effect in the system, which means that it will be active in the system for a short period, or be present from the beginning of the system's operation until the end.

2.3 Related Works

Applying a traditional approach without any domain knowledge is not efficient for exploring the fault space of a system [9]. A more mature approach is searching in a predetermined fault space. This approach employs the gridding of the FI parameters to determine the critical fault values inside a predetermined parameter space. The larger the violation is, the finer the gridding of the FI parameters until the boundary between safe state and the failure state is determined. This approach is a more guided approach than a random search approach but requires the specific ranges of the parameters.

Previous literature has also proposed approaches to prune the fault space by deriving equivalence classes for (possibly) effective faults and cutting off points that beforehand can be proven to be benign [9–11]. CriticalFault [12] uses the Architectural Vulnerability Factor (AVF) to measure the probability that a fault will affect the output of the program. The authors in [13] perform pre-injection analysis to know the data flow in the system and how a possible fault can propagate through components.

ML methods can also be used for optimizing FI in different application and perspectives. In [14], a ML algorithm is utilized to reduce the computational cost for Functional De-Rating of individual flip-flops. They trained an algorithm for one basic circuit then extend it to another sequential circuit with more complexity. In [15], the Learn-Based Technique (LBT) is used by applying the LBT on a black-box model by interacting with the model. Eventually, a simpler model of the model-under-test is built, and then verified with model checking approaches. The authors in [16] propose a learning-based algorithm to find the faults that have a high impact on the safety of the vehicle. They use

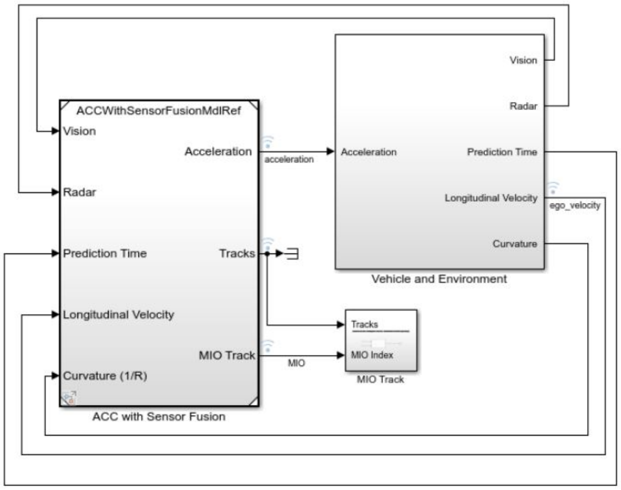


Figure 2: Adaptive cruise control schematic

a Bayesian network by considering domain knowledge to predict when the vehicle will be close to an accident, and they activate the fault at that time. [17] analyzes the impact of the soft error on virtual platforms by randomly injecting faults based on uniform distribution. Supervised and unsupervised ML techniques are then used to eliminate non-relevant information. That work also tries to identify the correlation between FI results and application and platform characteristics. The authors in [18] propose an Efficient Fault Injection System for Transient Fault (ESIFT), and it provides some statistics about the fault in the hardware under the test.

There are many studies on exploring and pruning the fault space in software-based FI, and hardware-based FI, especially on digital circuits, use cases. In addition, most of the ML techniques are developed for a homogeneous environment and use cases. However, there are not enough studies focusing on model-based techniques which covers broader applications and domains. Model-based FI is one way to ease the complexity of the evaluation process, as it performs at a higher level of abstraction [19–21].

3 Motivating Example

This section will describe the motivating example for our approach which is the Adaptive Cruise Control (ACC) with sensor fusion from a Mathworks Inc. example[1].This example has a reasonable level of complexity as it contains continuous and discrete parts with four vehicles travelling on a roadway, where vehicles are changing lanes.

In Fig. 2 we can see the model of the *ego vehicle*, which is the vehicle under study[2]. The model of the ego vehicle consists of the *ACC with sensor fusion* component and the *vehicle and environment*.

The vehicle which is in the same lane and in front of the ego vehicle is named the *lead vehicle*. In Fig. 3 we can see the ego vehicle follow the lead vehicle. The user of the ACC defines a safe distance and the ACC in the model tries to keep the relative distance equal or larger than the safe distance. The ACC has two modes based on relative distance and safe distance. If relative distance is larger than safe distance, the ACC increases the acceleration to reach the predefined speed. However if the relative distance is smaller than the safe distance, the ACC will decrease acceleration to increase the relative distance. In the following parts we describe each component of the example further, including the driving scenario.

3.1 ACC with Sensor Fusion

The *ACC with sensor fusion* controls the acceleration of vehicle by gathering sensor data. In this subsystem, there is a tracking and a sensor fusion model. This model receives the information from three sensors: *radar*, *vision*, and *velocity*. The sensor fusion block combines the radar sensor and vision sensor data to increase the environmental

[1] `https://www.mathworks.com/help/driving/examples/adaptive-cruise-control-with-sensor-fusion.html`.

[2] Model figure from `https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html`.

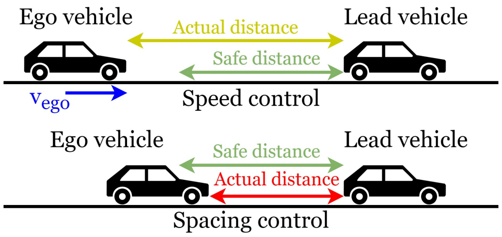


Figure 3: Adaptive cruise control using sensor fusion

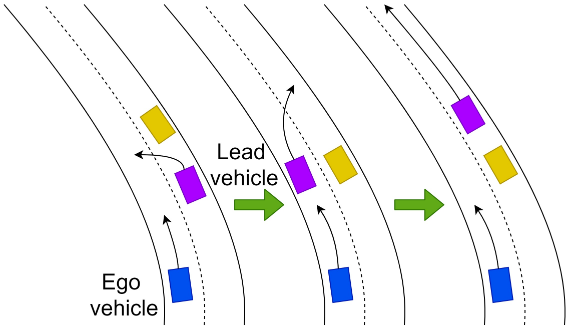


Figure 4: Driving scenario

perception. The ego vehicle based on this sensor data detects the other objects in the scenario, such as the lead vehicle and the relative velocity and distance to it. These data together with predefined safe distance and vehicle's speed provides the necessary information for ACC. The ACC will then increase or decrease acceleration of the ego vehicle to satisfy both the predefined velocity and safe distance, and avoid an accident between the two vehicles.

For ACC implementation there are two options in the model. One is based on Proportional-Integral–Derivative (PID) and another is based on Model Predictive Control (MPC). For this paper we selected the ACC with the PID controller.

3.2 Vehicle and Driving Scenario

In this subsystem there are two main components, the *vehicle dynamics* and the *actor and sensor simulation*. The vehicle dynamics is based on the bicycle model which is controlled by acceleration and steering signal. This component provides position and velocity data for actors and sensor data.

Within the actor and sensor subsystem, there is a driving scenario where the road and other vehicles are defined. In the Fig. 4 we can see the driving scenario. At first, the ego vehicle is moving with predefined speed within its lane. Then the *purple vehicle* cuts into the ego vehicle lane. Then the purple vehicle becomes the lead vehicle and the distance between the ego vehicle and lead vehicle becomes smaller than the safe distance. In a safe simulation, the ego vehicle must decrease its velocity to avoid an accident. Then the purple vehicle returns to its lane and the ego vehicle is able to move with its own speed. For the other vehicles in the scenario, no models are defined and they only follow a pre-defined path and speed in each moment.

3.3 Fault Possibilities

Based on the driving scenario and the position of the ego vehicle, the ACC receives data from the radar and vision sensor, and the vehicle dynamics provides a velocity value for ACC. Then the ACC uses these information to control the acceleration of the ego vehicle. If the measured data is not correct, it affects the vehicle safety. In this scenario, the ego vehicle may crash into the back of the lead vehicle entering its lane.

The most common fault type is a sensor fault [7]. The accuracy of a sensor reading is easily affected by a large number of environmental factors. A typical factor is the process noise, which is usually defined as a disturbance addition to the real value.

In this paper we check the safety of the vehicle when there is a fault in the velocity data. Therefore, we have fixed the location of the fault for the sake of simplicity and focus on exploring fault space for finding the critical fault amplitude

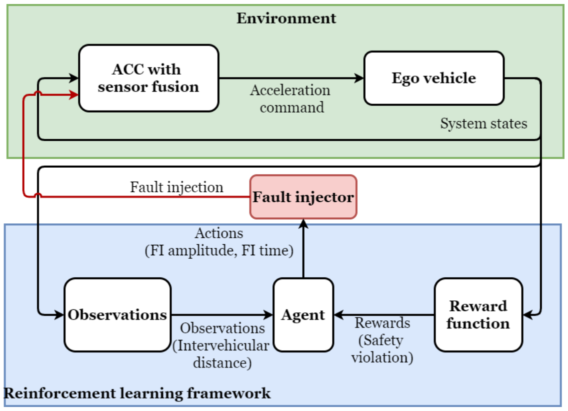


Figure 5: Connecting reinforcement learning, fault injection, and the environment

and fault injection times. A small example of a fault activation is defined as the following:

$$v_{ego} \begin{cases} v_{ego}, & t < f_t \\ v_{ego} = f_v, & t \geq f_t \end{cases} \tag{1}$$

where t denotes the real-time and f_t denotes the fault activation time. In this simple example, the fault activation sets the velocity of the ego vehicle v_{ego} to the defined fault value f_v as the time t passes f_t.

In the next section, we will demonstrate a RL-based technique for exploring the fault space in the determined use case regarding the ACC safety specification.

4 Technique

In this section we describe the proposed technique step-by-step. The focus of the technique is on using this method to explore the fault space and derive the critical fault parameters that can cause an accident.

4.1 Technique Framework

The proposed technique is based on model-based FI and uses an RL agent as a fault injector. For using RL in fault space exploration, we change the model-under-test to the setup in Fig. 5. All the models of the vehicles and the scenario become an environment for the new setup. The proposed approach defines the fault injector as the agent, and the actions are fault properties such as fault injection time and amplitude in a specific signal. By starting the training, the fault injector starts with initial values for the fault, and it eventually optimizes the value in each new episode.

4.2 Relevant Properties in the Use Case

In the ACC with a sensor fusion model, the requirement is to satisfy the user-defined speed while keeping the relative distance larger than a specific value between the ego vehicle and the lead vehicle. These two parameters guarantee the safety of the vehicle. Another critical aspect of reasoning is the different modes of the system or any fault-tolerant mechanism. For example, in our use case, we know the time and the condition that the system switches modes (or goes into safety mode). Also, the boundary conditions of the system and developer assumptions in system development are critical information as well as the testing scenario. In our use case, the test engineer must know how the ego vehicle and other vehicles behave in the simulation. Based on our knowledge about our use case, we consider a stuck-to-value fault in the velocity sensor to be the fault of interest. Then, we need to adjust and configure our RL agent.

4.3 Applying Reinforcement Learning

We must use the RL in a way that it exploits fault space for the highest long-term reward. In our case, as we explore the fault parameter space, we need to explore all fault value pairs. Therefore, a special emphasis on exploration vs. exploitation is required for the execution of the policy. At time t this action A_t can be denoted as:

$$A_t = \arg\underset{F^*}{max}\ R_t(F^*) \tag{2}$$

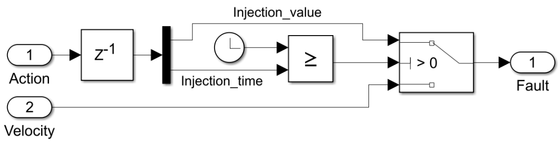


Figure 6: Determination of the agent action function as described in (1)

Where arg max denotes the value of F^* at which the received award R_t is maximized. In the use case, we add an RL agent, reward function, stopping criteria, and observation signal to the original model. The RL agent will then work as a fault injector to provoke the system to create an accident.

4.3.1 Environment

In our use case, all the models-under-test are our environment. In the environment, there are many signals and parameters. The tester does have to know about the model structure to provide some beneficial signals to the RL agent because the observation signals and action signals are crucial for the agent. Observation signals must represent the system state, especially its safety. In our use case, the observation signals are *acceleration* and *relative distance* as they determine the ego vehicle safety and position. The action signals as shown in Fig. 6, are the fault's parameters of *amplitude* and *injection time*.

4.3.2 Policy

An appropriate policy algorithm is critical for the use case. There are many algorithms based on the following parameters:

- Continuous or discrete action space
- Continuous or discrete observation spaces
- Policy-based or value-based algorithm or both
- Model-free or model-based method or both

For the policy in our use case, we use the Deep Deterministic Policy Gradient (DDPG) algorithm. It is a model-free and actor-critic reinforcement learning method [22] chosen because of the following reasons:

- Our use case is a hybrid and non-linear model. Therefore, the action space and observation space is infinite.
- While we provide some knowledge about the system, we want to be less dependent on user knowledge about the model-under-test.
- We do not know the best direction of exploring the action space. Hence, the agent should compute an optimal policy that maximizes the long-term reward.

We combine the benefits of direct policy mapping and value-based mapping in a third method called *actor-critic method* that guides the agent efficiently toward high reward states. In the actor-critic agent, the actor is the policy, and the critic is a function that improves the actor in each episode. In both critic and actor function, we use a Neural Network (NN) inspired by [22]. Our critic network consists of four fully-connected layers, and between each layer, we use the Rectified Linear Unit (ReLU) function. Our actor consists of two fully connected layers and a hyperbolic tangent layer between them. As with other ML techniques, there is no one right approach for settling on a NN structure. A lot of it comes down to starting with a structure that has already worked for the type of problem to solve and tweaking it from there.

We also provide the agent some information about the use case, so our agent is a combination of model-free and model-based approach. With this custom agent, we have the benefits of two methods together.

4.3.3 Reward Function

The reward function is made concerning the system's requirements. The reward function is defined in a negative way; namely, in our use case, the agent is rewarded for actions that lead to an accident. Therefore, the agent is encouraged to commit actions that yield unsafe states, and by analyzing the action signals, we are able to determine the most critical faults. For this purpose we make the reward function R_t at time t as:

$$R_t = ((max(0, min(200, x_r))^{-1})10)^2 + 0.1v_{ego} - t \tag{3}$$

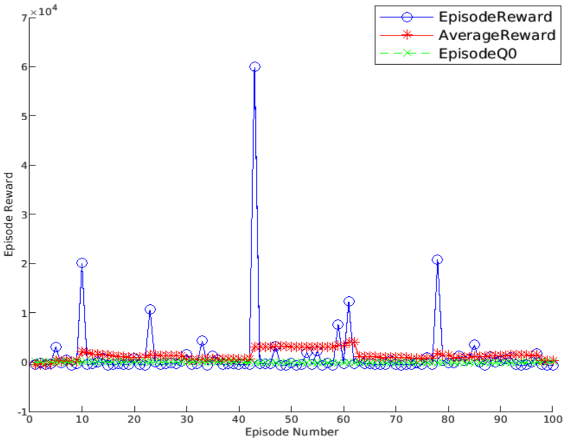


Figure 7: The training result

Our reward function is based on three parameters as follows:

- time (t): By advancing time, the reward value will decrease. This motivates the RL agent to inject a fault as soon as possible.
- Velocity of ego vehicle (v_{ego}): The reward must increase when the velocity is higher. This counters the speed limit that the user defines and motivates the ego vehicle to go faster and create an accident.
- Relative distance (x_r): The relative distance identifies the safety severity of the situation. When the ego vehicle crashes or the relative distance is less than the predefined margin, the reward value increases. This motivates the vehicle to violate the safe distance.

The amplitude of each factor must be set in a way to increase the reward value when the ego vehicle acts in an unsafe way. For example, if we decrease the impact of the relative distance, the agent is less likely will try to violate the safe distance. On the other hand, if we increase its impact too much, our results suffer as we are decreasing the impact of other factors like velocity.

In our experiments, when neglecting the velocity term, the agent finds less critical faults in the fault space. It is because the velocity itself motivates the vehicle toward the beginning of an unsafe state while the relative distance will be most useful when the vehicle is closer to the unsafe situation.

In Eq. (3), mathematical operations and functions such as square function, multiplication, and inverse function magnify the relative distance impact compared to the other parameters as described. Choosing those operations are based on the domain knowledge, our expertise and some trial and error simulation.

In this configuration, the RL agent tweaks the NN after each episode to increase the reward. Hence, it explores the fault space at a fixed location (velocity sensor) to find both critical fault amplitude and injection time. To explore a broader fault space for the system, we must change the injection location in the model and perform the aforementioned steps. For example, we could change the injection location (action signal of RL) to be the radar sensor. Then in the highest rewarded episode, we obtain the final parameters for use in our fault testing.

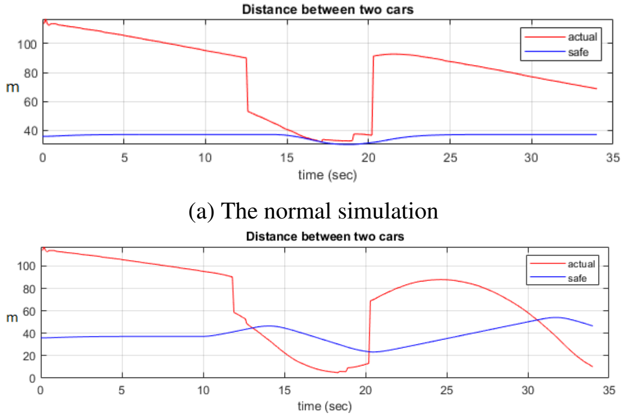
5 Results

This section will describe our obtained results and their verification, as well as present a brief comparison of this technique to a Monte Carlo approach.

5.1 Results and their Verification

Fig. 7 shows the result of the training. The peaks in the diagram show the maximum reward reached over the different episodes. The reward function was tested in a non-faulty model where there is no accident. If the episode reward is greater than the reward in the non-faulty model, we conclude that the ego vehicle violated the safe distance or that an accident was caused.

To select the faults, we examine the peaks in Fig. 7. The different amplitudes of the peaks indicates that we have different unsafe episode with varying severity. In one of the unsafe episode (one peak), we choose the value that appears most often in that episode for both fault amplitude and fault injection time. This fault is then injected into the model, and the simulation is run. The visualization of the result with three dimensional (3D) animation or by plotting the simulation traces allows us to validate whether the fault produces unsafe vehicle behavior or not.



(a) The normal simulation

(b) The faulty simulation

Figure 8: Simulation results with and without a fault

In Fig. 8, we can see the relative distance in the red line and the safe distance in the blue line. In the fault-free simulation in Fig. 8a, the relative distance is always larger than the safe distance. On the other hand, with a faulty simulation in Fig. 8b, we see that between 12 seconds and 20 seconds, the relative velocity is lower than the safe distance. It indicates that the ego vehicle violates safety requirements.

5.2 Comparison

We also briefly compared our method with Monte-Carlo fault injection (MCFI) in Table 1, which is based on randomly choosing fault parameters. The data is averaged between two experiments for MCFI. In MCFI, a uniform distribution is used for selecting fault amplitude and time of injection. Our performance metric was the number of simulations that can find a critical scenario. Hence, we do not consider the personal computer's (PC) configuration. We ran MCFI twice, and the average of the minimum number of simulations to obtain the first critical fault was fifteen, while our approach reached a critical fault after three simulations. Our method also found nineteen critical faults in a hundred simulations while MCFI finds three. Thus we have promising results that our approach is more efficient than a Monte Carlo approach.

Table 1: Fault space exploration result comparison

Approach	**Num. Of Sims. for First Critical Fault**	**Total Num. of Critical Faults**
MCFI	15	3
RL	3	19

6 Discussion

In this section we will describe challenges encountered during this research and some lessons learned.

6.1 Challenges

These challenges mainly revolve around two principal questions essential to ML and Artificial Intelligence (AI): a) *how to ensure that the technique will provide a good solution?*, and b) *how to improve the policy and handle imperfections?*

The first challenge is that there is no straight forward way to define a proper RL structure. Each of the agent structure (for example hidden layers), agent parameters, training parameters, observation signal, reward function, action signal, and simulation parameters all affect the training result. In addition, as RL is a black-box (like all ML techniques) for the users, we can only validate the technique and the results by simulating of the obtained critical faults.

A second challenge is that training process requires time and effort, especially in the cycle of redesigning, training, and testing. For each iteration, the deep learning algorithm has to perform a lot of exploration. Hence, finding a proper RL configuration requires manual work and trial-and-error. As the require time for this cycle is highly dependant on user expertize, we skipped to include this parameter in Table 1.

Related open questions are how to map the domain knowledge to the problem, and use the domain knowledge to direct the exploration not to lose test coverage but reduce the computation time. The domain knowledge must guide us to a proper definition of a reward function and observation signals. As well, the principles are unclear as to how to extend the same RL structure and function to other use cases. We cannot use one RL configuration in different applications or even different scenarios. The RL results are dependent on scenario related parameters and its setting.

For example, the reward function has a critical rule for finding the most destructive scenario. If the reward function is not appropriately defined, the RL algorithm will not explore the fault space in the right way. If the RL can then not find the solution, we cannot extrapolate that there is no solution to the problem.

Another challenge can be not observing convergence in the trace (c.f. Fig. 7) with an evolving training process, while having a convergence trace is mandatory in many applications of RL. Convergence means that we have reached an optimal parameter. In fault space exploration, a converged curve would provide us one number for each parameter instead of multiple, and it will not continue to explore other areas. Therefore in this paper, we are not interested in a converged curve. Instead, to avoid getting stuck at a local maximum, the RL approach should continuously explore the other possibilities by trying different fault value/time pairs.

6.2 Lessons Learned

In the proposed technique, the reward function plays a key role. It formalizes the goal of the agent. The method to make the agent do the described task or achieve a goal depends on how the reward function is defined for it. With experimenting in the use case, we learned how to shape the reward function effectively. Any discontinuity in the reward function makes fault space exploration very hard and time-consuming for the agent. It must also be defined in a way that it will gradually increase when the agent is close to its goal and it must avoid to get stuck to local maximum.

This definition must take into account domain knowledge, so the test engineer must be familiar with the model-under-test and how it operates in each test scenario. Hence, we propose to check for proper operation of the reward function before running the learning process by running the simulation without any RL agent and examining the output of the reward function.

The RL algorithm in the policy also is essential. There are many algorithms based on the nature of the problem that we want to solve. Algorithms have been categorized based on continuity/discontinuity, off-policy/on-policy, deterministic/stochastic, etc. Therefore, we note that choosing a proper algorithm has a significant impact on the result.

Another critical factor is the observation signals chosen, as they must represent the crucial states of the system. Hence, choosing too many signals will not be profitable for the agent. Using domain knowledge and understanding the model-under-test will help to determine the highly effective signals.

The agent setting also is important. For example, we can determine ranges for observations and action signals, which has a significant impact on finding faults in the space. Typically users do not define any limits for them, and the agent must explore infinite numbers, decreasing the fault coverage. Instead, by considering the ranges of real-world faults, the user may be able to determine the limits more efficiently.

The last lesson learned is related to interpreting the training result. For extracting the critical fault from training result, we use the values that appears must often as the result of training does not consist of fixed value for fault parameters. In all of the critical faults we observe that the fault value is frequently repeated.

7 Conclusion and Future Works

This work explores the use of reinforcement learning (RL) for optimizing fault injection (FI) in terms of performance and fault coverage. On our use case, we converted the problem to a RL algorithm where we defined a reward function and the observation signals using knowledge about the model-under-test. The RL agent can then move toward the intended behaviour instead of exploring the fault space randomly, and this guided-search provides us a sequence of fault injection parameters which could lead to hazardous situations in autonomous driving.

This approach can be applied to a gray-box systems or black-box systems, both in hardware-based FI and software-based FI, because we do not need to know about the system in detail. The proposed approach interacts with the system with some defined signals. Therefore, the high-level knowledge about the system under the test is sufficient.

This work also demonstrates the effectiveness of intelligent fault space exploration mechanisms in contrast to the random parametrization of faults. We have explored how a guided search with a well-defined reward function can prove more useful than Monte Carlo-based FI techniques.

For our future work, we plan to generalize the methodology to apply it to a variety of different problems. For this purpose we will work on modelling the domain knowledge in the process and using a mature RL agent for fault space exploration. In this paper we have used a deterministic policy for the RL agent while using a stochastic policy will be more beneficial as it will explore fault space based on the probability of all next actions. It also provides an interpretable result through an examination of the final trained network.

Acknowledgment

This work was partly funded by Flanders Make vzw, the strategic research centre for the Flemish manufacturing industry; and by the Flanders Make project aSET (grant no. HBC.2017.0389) of the Flanders Innovation and Entrepreneurship agency (VLAIO). The authors thank Moharram Challenger for his useful suggestions.

References

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363–369.

[2] J. Arlat *et al.*, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb 1990.

[3] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer Science & Business Media, 2003, vol. 23.

[4] A. S. Polydoros and L. Nalpantidis, "Survey of model-based reinforcement learning: Applications on robotics," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 2, pp. 153–173, 2017.

[5] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement learning and dynamic programming using function approximators*. CRC press, 2017.

[6] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. Eslami *et al.*, "Emergence of locomotion behaviours in rich environments," *arXiv preprint arXiv:1707.02286*, 2017.

[7] M. Saraoğlu, A. Morozov, M. T. Söylemez, and K. Janschek, "Errorsim: A tool for error propagation analysis of Simulink models," in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2017, pp. 245–254.

[8] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[9] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 123–134, 2012.

[10] X. Meng, Q. Tan, Z. Shao, N. Zhang, J. Xu, and . Zhang, "Optimization methods for the fault injection tool seinjector," in *2018 International Conference on Information and Computer Technologies (ICICT)*, March 2018, pp. 31–35.

[11] C. Dietrich, A. Schmider, O. Pusz, G. P. Vayá, and D. Lohmann, "Cross-layer fault-space pruning for hardware-assisted fault injection," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

[12] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.

[13] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *Dependable Computing - EDCC 5*, M. Dal Cin, M. Kaâniche, and A. Pataricza, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 246–262.

[14] T. Lange, A. Balakrishnan, M. Glorieux, D. Alexandrescu, and L. Sterpone, "On the estimation of complex circuits functional failure rate by machine learning techniques," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, June 2019, pp. 35–41.

[15] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Virtualized-fault injection testing: A machine learning approach," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 297–308.

[16] S. Jha, S. Banerjee, T. Tsai, S. K. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "Ml-based fault injection for autonomous vehicles: A case for Bayesian fault injection," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2019, pp. 112–124.

[17] F. R. da Rosa, R. Garibotti, L. Ost, and R. Reis, "Using machine learning techniques to evaluate multicore soft error reliability," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 6, pp. 2151–2164, June 2019.

[18] N. Tian, D. Saab, and J. A. Abraham, "Esift: Efficient system for error injection," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, July 2018, pp. 201–206.

[19] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "Modifi: a model-implemented fault injection tool," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2010, pp. 210–222.

[20] M. Moradi, B. Van Acker, K. Vanherpen, and J. Denil, "Model-implemented hybrid fault injection for Simulink (tool demonstrations)," in *Cyber Physical Systems. Model-Based Design*, R. Chamberlain, W. Taha, and M. Törngren, Eds. Cham: Springer International Publishing, 2019, pp. 71–90.

[21] M. Moradi, C. Gomes, B. J. Oakes, and J. Denil, "Optimizing fault injection in FMI co-simulation through sensitivity partitioning," in *Proceedings of the 2019 Summer Simulation Conference*, ser. SummerSim '19. San Diego, CA, USA: Society for Computer Simulation International, 2019.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.