Search-Based Repair of DNN Controllers of AI-Enabled Cyber-Physical Systems Guided by System-Level Specifications

Deyun Lyu
lyu.deyun.107@s.kyushu-u.ac.jp
Kyushu University
Fukuoka, Japan

Zhenya Zhang
zhang@ait.kyushu-u.ac.jp
Kyushu University
Fukuoka, Japan

Paolo Arcaini
arcaini@nii.ac.jp
National Institute of Informatics
Tokyo, Japan

Fuyuki Ishikawa
f-ishikawa@nii.ac.jp
National Institute of Informatics
Tokyo, Japan

Thomas Laurent
tlaurent@tcd.ie
SFI Lero@Trinity College Dublin
Dublin, Ireland

Jianjun Zhao
zhao@ait.kyushu-u.ac.jp
Kyushu University
Fukuoka, Japan

ABSTRACT

In *AI-enabled CPSs, DNNs* are used as controllers for the physical system. Despite their advantages, DNN controllers can produce wrong control decisions, which can lead to safety risks for the system. Once wrong behaviors are detected, the DNN controller should be fixed. DNN repair is a technique that allows to perform this fine-grained improvement. However, state-of-the-art DNN repair techniques require ground-truth labels to guide the repair. For AI-enabled CPSs, these are not available, as it is not possible to assess whether a specific control decision is correct. Nevertheless, it is possible to assess whether the DNN controller leads to wrong behaviors of the controlled system by considering system-level requirements. In this paper, following this observation, we propose a novel DNN repair approach that is guided by system-level specifications. The approach takes in input a system-level specification, some tests violating the specification, and some faulty DNN weights. The approach searches for alternative weight values with the goal of fixing the behavior on the failing tests without breaking the passing tests. We also propose a heuristic that allows us to accelerate the search by avoiding the execution of some tests. Experiments on real-world AI-enabled CPSs show that the approach effectively repairs their controllers.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Software testing and debugging**.

KEYWORDS

DNN repair, Neural network controllers, Cyber-physical systems

ACM Reference Format:
Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Fuyuki Ishikawa, Thomas Laurent, and Jianjun Zhao. 2024. Search-Based Repair of DNN Controllers of AI-Enabled Cyber-Physical Systems Guided by System-Level Specifications. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

GECCO '24, July 14–18, 2024, Melbourne, VIC, Australia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Cyber-physical systems (CPS) are systems in which physical plants are controlled by computer systems, and they have been successfully applied in different domains, even safety-critical, like transportation, healthcare, etc. In order to improve their capabilities, CPSs are increasingly adopting *deep neural networks (DNNs)* as controllers, leading to the emergence of a new class of systems called *AI-enabled CPS* [20, 27, 32, 45]. For example, in autonomous driving systems, DNN controllers permit to process complex environmental data collected by sensors and produce intelligent control decisions.

Despite their numerous benefits, DNN controllers can be faulty and can lead to failures of their AI-enabled CPSs; in safety-critical domains, such as autonomous driving systems, such failures can constitute serious safety issues. Debugging and improving DNNs is particularly difficult due to their data-driven computation paradigm and to the fact that the decision logic is not explicit but encoded in the setting of the weights. In order to tackle this problem, *DNN repair* is an emerging technique that allows performing localized modifications to a DNN with the goal to improve its performance [7, 17–19, 31, 34, 36, 42, 43].

However, the application of existing DNN repair approaches to DNN controllers of AI-enabled CPSs is not straightforward. Indeed, state-of-the-art repair techniques are usually applied for classification problems and require to have ground-truth labels for the DNN inputs used during repair. In an AI-enabled CPS, such ground-truth information regarding the DNN controller is not available; indeed, given a control decision of the DNN controller, it is not possible to decide whether it is correct or not. On the other hand, requirements are available regarding the expected behavior of the controlled physical plant. Therefore, it is possible to specify *system-level specifications* predicting the behavior of the controlled physical plant: if these specifications are satisfied, we can (indirectly) conclude that the DNN controller is behaving correctly; instead, a violation of the system specification is an indication that the controller provided a wrong control decision.

In this paper, based on this intuition, we propose ContrRep, an approach to repair DNN controllers of AI-enabled CPSs that is guided by the evaluation of a system-level specification. Specifically,

ContrRep takes as input an AI-enabled CPS $\mathcal{M}^{C_{orig}}$ composed of a physical plant $\mathcal{M}$ and a DNN controller C_{orig}, a system specification φ over $\mathcal{M}^{C_{orig}}$ written in Signal Temporal Logic (STL), and a test suite TS, in which some tests in TS satisfy the specification (called TS^{+}_{orig}) and some do not (called TS^{-}_{orig}). ContrRep also takes in input a set of *suspicious weights* SW of the DNN controller C_{orig}, that have been identified by some fault localization technique [5, 6, 31]; these weights are those responsible for the failures in TS^{-}_{orig}. ContrRep adopts a search-based approach to find an alternative setting for weights SW, with the goal of making the system pass the tests in TS^{-}_{orig}, and not break those in TS^{+}_{orig}.

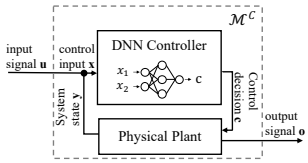
During the search, ContrRep requires, for each fitness function evaluation, to execute the candidate AI-enabled CPS $\mathcal{M}^{C_{SW\leftarrow\overline{v}}}$ over all the tests in TS. This can be computationally expensive, in particular, if the test execution relies on some expensive simulation. To tackle this problem, we propose a version of the repair approach (called ContrRep$_{\text{FAST}}$), that adopts a heuristic to compute the fitness function faster. The heuristic method exploits the STL *quantitative robustness* that, for a given system execution, not only tells *whether* the specification is satisfied or not, but also provides a real value that tells *how robustly* it does so (i.e., "how far" it is from violation/satisfaction): negative robustness means that the specification is violated, with lower values indicating more severe violations; positive robustness, instead, means that the specification is satisfied, with higher values indicating stronger satisfaction. The heuristic method sorts the failed tests TS^{-}_{orig} by decreasing robustness and evaluates them in order: as soon as a test is not repaired (i.e., it is not satisfied by the candidate AI-enabled CPS $\mathcal{M}^{C_{SW\leftarrow\overline{v}}}$), the remaining tests with lower robustness are not evaluated and considered as failing, as it is unlikely that they could be repaired (as they are more difficult to repair). Similarly, the passing tests TS^{+}_{orig} are sorted by increasing robustness and evaluated in order: as soon as a test passes (i.e., it is also satisfied by the candidate AI-enabled CPS $\mathcal{M}^{C_{SW\leftarrow\overline{v}}}$), the remaining tests with higher robustness are not evaluated and it is assumed that they will also pass (as they are more difficult to break).

Paper structure. §2 introduces necessary background, and §3 presents the proposed approach. §4 introduces the design of the experiments, and §5 presents the experimental results. Finally, §6 discusses threats that may affect the validity of the approach, §7 reviews related work, and §8 concludes the paper.

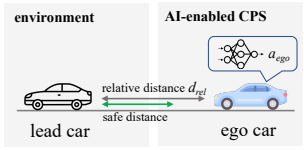
2 PRELIMINARIES

2.1 AI-Enabled Cyber-Physical Systems

Definition 1 (AI-enabled CPS)**.** As shown in Fig. 1, an AI-enabled CPS consists primarily of a DNN controller C and a physical plant $\mathcal{M}$. At a time instant t, the DNN controller C gives a control decision $\mathbf{c}(t)$ that decides the evolution of the state of the physical plant, based on external input signal $\mathbf{u}(t)$ and the state of the plant $\mathbf{y}(t)$. The dynamics of the plant is formulated as follows: $\dot{\mathbf{y}}(t) = \mathcal{M}(\mathbf{y}(t), \mathbf{c}(t))$, where $\mathcal{M}$ is a black-box function such that it can accommodate complex dynamics and third-party confidential components. Overall, the whole system $\mathcal{M}^{C}$ can be viewed as a function that maps an input signal $\mathbf{u}$ to an output signal $\mathbf{o}$. ◁



(a) AI-enabled CPS



(b) Adaptive Cruise Control (ACC)

Figure 1: AI-enabled CPS model and an example ACC

The following example uses the ACC system (later introduced in §4.2) to illustrate the operation process of a CPS system.

Example 1. *Adaptive Cruise Control* (ACC), as shown in Fig. 1b, is an advanced control system that aims to maintain a safe distance of a vehicle (*ego car*) from the preceding vehicle (*lead car*). In Fig. 1b, the ego car is considered as the physical plant, under the control of a DNN controller. The lead car is considered as the surrounding environment, and its speed and position detected by the sensors of the ego car are treated as the external input signal $\mathbf{u}(t)$ to the ACC. The signal $\mathbf{y}(t)$ fed back from the plant is the speed and position of the ego car. The two signals are combined into a vector $\mathbf{x}(t)$ and sent to the controller C as its input for decision-making. Based on the input, the DNN controller C computes a control signal, i.e., the acceleration of the ego car, to control the motion of the plant. ◁

As shown in Def. 1, the state evolution of the whole AI-enabled CPS is primarily decided by the DNN controller, and therefore, we take it as our main target for repairing unsafe system behavior. While our technique is applicable to various types of neural networks, we elaborate on the widely-adopted fully-connected DNNs [31, 45].

Definition 2 (DNN controller)**.** A DNN controller C is a function that, at each timestamp, maps an input vector $\vec{x}$ to an output scalar c (i.e., a control decision). It consists of L hidden layers and an output layer. The i-th hidden layer computes an output vector $\vec{x}_i$ by $\vec{x}_i = \sigma(\mathcal{W}_i\vec{x}_{i-1} + \vec{b}_i)$, where $\vec{x}_{i-1}$ is the output of the $(i-1)$-th hidden layer, $\mathcal{W}_i$ is a matrix of *weights* and $\vec{b}_i$ is a vector of *bias* at the i-th layer, and σ is a non-linear *activation function*, such as *ReLU*, *sigmoid* and *tanh*. The output layer computes the output c of the DNN controller, by taking the weighted sum over different components of the output $\vec{x}(i)$ of the last hidden layer. ◁

In this paper, we follow the literature [18, 31, 34, 36] and consider the *weights* as the target of repair. Specifically, we identify by $\mathcal{W}$ the set of weights across all the hidden layers in a DNN controller.

2.2 System Specification

In AI-enabled CPSs, temporal logics, especially *signal temporal logic (STL)*, are often used as specification language that formalizes the expected system behaviors and properties. In the following, we review the STL syntax and semantics in [8].

Definition 3 (STL syntax)**.** Let $\vec{o} \in \mathbb{R}^d$ be a vector. In STL, an *atomic proposition* is represented as $\alpha :\equiv (f(\vec{o}) > 0)$, in which $f : \mathbb{R}^d \to \mathbb{R}$ is a function that maps $\vec{o}$ to a real number. The syntax of an STL formula φ is defined as follows:

$$\varphi :\equiv \alpha \mid \bot \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box_I\varphi \mid \Diamond_I\varphi \mid \varphi_1\mathcal{U}_I\varphi_2$$

Here, I is a time interval $[a, b]$, where $a, b \in \mathbb{R}_{\geq 0}$ and $a < b$. $\Box_I$, $\Diamond_I$ and $\mathcal{U}_I$ are temporal operators *always*, *eventually* and *until*, that allow to express complex temporal properties. ◁

The semantics of STL characterizes the extent to which a signal $\mathbf{o}$ satisfies a specification φ. Formally, it is defined as follows.

Definition 4 (STL robust semantics)**.** Given a system output signal $\mathbf{o}$ and an STL specification φ, the *robust semantics* returns a quantity $[\![\mathbf{o}, \varphi]\!] \in \mathbb{R} \cup \{\infty, -\infty\}$ that indicates *how robustly* $\mathbf{o}$ satisfies or violates φ. The formal definition of STL semantics is as below:

$$[\![\mathbf{o}, \alpha]\!] := f(\mathbf{o}(0)) \qquad [\![\mathbf{o}, \neg\varphi]\!] := -[\![\mathbf{o}, \varphi]\!]$$
$$[\![\mathbf{o}, \varphi_1 \wedge \varphi_2]\!] := \min\left([\![\mathbf{o}, \varphi_1]\!], [\![\mathbf{o}, \varphi_2]\!]\right)$$
$$[\![\mathbf{o}, \Box_I \varphi]\!] := \inf_{t \in I}\left([\![\mathbf{o}^t, \varphi]\!]\right)$$
$$[\![\mathbf{o}, \varphi_1 \,\mathcal{U}_I\, \varphi_2]\!] := \sup_{t \in I}\left(\min\left([\![\mathbf{o}^t, \varphi_2]\!], \inf_{t' \in [0,t)} [\![\mathbf{o}^{t'}, \varphi_1]\!]\right)\right)$$

where $\mathbf{o}^t$ denotes the t-shift of $\mathbf{o}$, namely, for an arbitrary $t' \in [0, T-t]$, it holds that $\mathbf{o}^t(t') = \mathbf{o}(t+t')$. The semantics of the omitted operators can be derived from those of the existing operators, based on their corresponding syntactic equivalence relations.

Through the robust semantics, we can infer the satisfaction of $\mathbf{o}$ to φ: a positive $[\![\mathbf{o}, \varphi]\!]$ implies that $\mathbf{o}$ satisfies φ (i.e., $\mathbf{o} \models \varphi$), while a negative $[\![\mathbf{o}, \varphi]\!]$ implies that $\mathbf{o}$ violates φ (i.e., $\mathbf{o} \not\models \varphi$); ◁

3 PROPOSED REPAIR APPROACH

In this section, we describe ContrRep, our proposed approach for repairing DNN controllers of AI-enabled CPSs. Note that state-of-the-art DNN repair approaches are not applicable to DNN controllers, as they require the ground truth of the DNN behaviors that are not available for DNN controllers. We share with existing DNN repair approaches the basic assumption that failures are due to a few components of the DNN that are not properly tuned, and only these should be changed to fix the failures.

ContrRep takes in input an AI-enabled CPS $\mathcal{M}^{C_{orig}}$, a system specification φ, and a test suite TS of input sequences for $\mathcal{M}^{C_{orig}}$. TS should be generated with some generation technique [30, 44] that guarantees enough coverage of the input space. Some tests in the test suite satisfy φ and some do not; for convenience, we partition the test suite as $TS^+_{orig} \cup TS^-_{orig}$, where:

$$TS^+_{orig} = \{t \in TS \mid \mathcal{M}^{C_{orig}}(t) \models \varphi\}$$

$$TS^-_{orig} = \{t \in TS \mid \mathcal{M}^{C_{orig}}(t) \not\models \varphi\}$$

The correctness of the DNN controller C_{orig} depends on how many tests satisfy the specification. We introduce a *correctness measure* defined as:

$$CM(\mathcal{M}^{C_{orig}}, \varphi, TS) = \frac{|TS^+_{orig}|}{|TS|} \tag{1}$$

In the following, in §3.1, we explain how the weights to repair are selected; in §3.2, we introduce our proposed repair approach, and in §3.3, we present a heuristic to speed up the repair process.

3.1 Identification of the weights to repair

The components to modify during repair are usually identified by some fault localization (FL) technique, inspired by fault localization for classic code [41]. Different FL approaches have been proposed for DNNs [4, 6, 31], mainly targeting the weights of the network; in our work, we follow this trend, and we also consider the weights as the target of the repair. The main intuition behind these approaches is that the weights that are activated during failures are more likely to be faulty; FL approaches identify these weights and return them as *suspicious weights* SW.

Our approach assumes that some FL approach has been applied to the weights of C_{orig}, considering the results of the execution of the test suite TS (i.e., TS^+_{orig} and TS^-_{orig}), and has identified a set of *suspicious weights*. We want to point out that, similarly to FL for classic code, DNN FL is usually able to correctly identify the faulty weights in SW, but also identifies other additional weights that are not responsible for the failures. However, it is not known which of the returned weights are the faulty ones.

In our experiments, we are interested in assessing the capabilities of the repair approach and not of the FL approach. Therefore, in order to perform a proper assessment, we need to know the quality of the FL results. To do so, we proceed as usually done in the controlled experiments for automatic program repair [9]. Namely, we inject faults in the weights of the controller C_{orig}; then, we build the set of suspicious weights SW, by including these faulty weights plus others that are actually correct. In this way, we reproduced in a controlled way the noise that can be introduced by an FL approach in the suspicious weights SW. We provide more details in §4.

3.2 ContrRep – Search-based repair of the DNN controller

In this section, we present our proposed repair approach. We cast the repair problem as a search problem, whose goal is to find better values for the suspicious weights SW. Specifically, we define a single-objective search problem as follows.

The *search variables* $\overline{x}$ of the approach are the possible alternative values for SW, i.e., :

$$\overline{x} = [x_1, \ldots, x_{|SW|}]$$

A *search individual* $\overline{v}$ is an assignment to the search variables that can be generated during the search. A search individual identifies a new DNN controller whose weights SW are set to the values $\overline{v}$ and the other weights keep the same value as in C_{orig}. We will identify it as $C_{SW \leftarrow \overline{v}}$; we will also identify as $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$ the AI-enabled system equipped with the modified controller.

In order to define the *search space* for the search variables $\overline{x}$, we proceed as follows. The search space of a variable x depends on the original value v_{orig} of the corresponding weight w in C_{orig}. Namely, we allow to modify the weight in a way that its contribution to the activation function is at most δ times the original one or at least a $1/\delta$ fraction of it, where $\delta \in \mathbb{R}_{>0}$ is a parameter of the search. This is obtained by defining the search space of x as $\left[v_{orig} \cdot \delta^{-sign(v_{orig})}, v_{orig} \cdot \delta^{sign(v_{orig})}\right]$, where $sign(v_{orig}) \in \{-1, 1\}$ identifies the sign of v_{orig}. In the experiments, we set $\delta = 2$.

The goal of the search is to find an alternative controller $C_{SW \leftarrow \overline{v}}$ that can fix (i.e., make them pass) the tests failing with the original controller (i.e., TS^-_{orig}) and not break those that were passing (i.e., TS^+_{orig}). So, given an individual $\overline{v}$, the *fitness function* that must be

maximized is:

$$fit(\overline{v}) = CM(\mathcal{M}^{C_{SW \leftarrow \overline{v}}}, \varphi, TS) \quad (2)$$

We adopt Differential Evolution (DE) [33] as the underlying search algorithm as it has been successfully used in other DNN repair approaches for image classifiers like Arachne [31]. However, any other population-based algorithm could be adopted.

At the end of the search, the approach returns the individual that obtained the best performance. We will identify the corresponding controller and AI-enabled CPS as C_{best} and $\mathcal{M}^{C_{best}}$.

3.3 ContrRep$_{\text{FAST}}$ – Speeding Up the Fitness Computation

The calculation of the fitness as described in §3.2 can be quite expensive, in particular, if the test suite TS contains several tests and the cost of executing one test is high (e.g. when relying on simulators). Therefore, in this section, we propose a modified version of the approach (called ContrRep$_{\text{FAST}}$) that adopts a fitness function that still tries to compute the correctness measure, but in a faster way:

$$fit_{\text{FAST}}(\overline{v}) = \text{ApproxFit}(TS, TS^{-}_{orig}, TS^{+}_{orig}, \mathcal{M}^{C_{SW \leftarrow \overline{v}}}) \quad (3)$$

The fitness computation relies on the heuristic implemented in the method ApproxFit that tries to avoid running all the tests by *estimating* which is the possible assessment (i.e., pass or fail) of some of them. The heuristics makes use of the quantitative robustness value associated with each test execution (see Def. 4) and it is based on the following intuitions:

- let us consider a failing test t in TS^{-}_{orig} (i.e., failing with the original system $\mathcal{M}^{C_{orig}}$) with negative robustness; if t is not repaired with $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$ (i.e., the robustness $[\![\mathcal{M}^{C_{SW \leftarrow \overline{v}}}(t), \varphi]\!]$ of the test t is still negative when executed with $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$), all the tests in TS^{-}_{orig} having lower robustness will likely be not repaired as well. The intuition is that the robustness value identifies the "difficulty to repair" and if it is not possible to repair "easy tests" (i.e., tests close to positive robustness), it is probably also not possible to repair more difficult tests;
- let us consider a passing test t in TS^{+}_{orig} (i.e., passing with the original system $\mathcal{M}^{C_{orig}}$) with positive robustness; if t is still satisfied (i.e., the robustness $[\![\mathcal{M}^{C_{SW \leftarrow \overline{v}}}(t), \varphi]\!]$ of the test is still positive with $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$), all the tests in TS^{+}_{orig} having higher robustness will likely be still passing as well. The intuition is that if the repair action does not break tests that are close to being violated (i.e., tests close to negative robustness), most probably also tests with higher robustness (i.e., farther from violation) will be preserved.

Based on the previous intuitions, the heuristic ApproxFit is defined as presented in Alg. 1. It works as follows:

- it sorts the tests failing with $\mathcal{M}^{C_{orig}}$ by decreasing robustness (line 2) and those passing with $\mathcal{M}^{C_{orig}}$ by increasing robustness (line 3);
- it visits the failing tests in decreasing order of robustness (line 5). For each test t^{-}_{i}:
  - if the test passes with the AI-enabled CPS $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$ under evaluation, i.e., the test is repaired (line 6), the counter of passing tests TP is increased (line 7);
  - otherwise, if the test fails, the approach assumes that also the remaining tests with lower robustness will fail, and so it stops the iteration (line 9);
- it visits the passing tests in increasing order of robustness (line 10). For each test t^{+}_{i}:
  - if the test passes with the AI-enabled CPS $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$ under evaluation, i.e., the test has not been broken (line 11), the approach assumes that also all the other tests with higher robustness will not be broken, and so the counter of passing tests TP is increased accordingly without executing those tests (line 12), and the iteration is stopped (line 13);
  - otherwise, the iteration continues by executing the next test with higher robustness.
- the ratio of passing tests over the total number of tests is returned as computed approximated fitness of $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$ (line 14).

Algorithm 1 ApproxFit – Heuristic for fitness computation

Require: TS: test suite
Require: TS^{-}_{orig}: tests failing in $\mathcal{M}^{C_{orig}}$
Require: TS^{+}_{orig}: tests passing in $\mathcal{M}^{C_{orig}}$
Require: $\mathcal{M}^{C_{SW \leftarrow \overline{v}}}$: AI-enabled CPS under assessment
Ensure: approximated fitness

```
1:  function ApproxFit(TS, TS^-_orig, TS^+_orig, M^{C_{SW←v̄}})
2:      [t^-_1, ..., t^-_{|TS^-_orig|}] ← sortDecrRob(TS^-_orig)
3:      [t^+_1, ..., t^+_{|TS^+_orig|}] ← sortIncrRob(TS^+_orig)
4:      TP ← 0                                  ▷ # of tests passing with M^{C_{SW←v̄}}
5:      for i ← {1, ..., |TS^-_orig|} do
6:          if M^{C_{SW←v̄}}(t^-_i) ⊨ φ then
7:              TP ← TP + 1
8:          else
9:              break
10:     for i ← {1, ..., |TS^+_orig|} do
11:         if M^{C_{SW←v̄}}(t^+_i) ⊨ φ then
12:             TP ← TP + ((|TS^+_orig| − i) + 1)
13:             break
14:     return TP / |TS|
```

4 EXPERIMENT DESIGN

We here present the design of the experiments. All the code, benchmarks, and experimental results are reported online [21].

4.1 Research questions

We identified the following research questions (RQs) to assess the proposed approach:

RQ1: How is the repair performance of ContrRep? Is it affected by the quality of the fault localization results?
In this RQ, we want to assess if ContrRep can actually repair DNN controllers. Moreover, we want to assess whether having imprecise fault localization results (i.e., containing weights that are not faulty) affects the repair effectiveness.

RQ2: Does ContrRep$_{\text{FAST}}$ reduce the execution time of the repair approach? Does it affect the repair performance?
In this RQ, we want to assess whether the heuristic implemented by ContrRep$_{\text{FAST}}$ is indeed effective in reducing the

Table 1: AI-enabled CPSs $\mathcal{M}^C$ and correctness measure of its faulty versions $\mathcal{M}^{C_{fault}}$ (CM ($\mathcal{M}^{C_{fault}}$, φ, TS) (%))

$\mathcal{M}^C$	ACC#1	ACC#2	AFC#1	AFC#2
Structure of C	[15 15 15]	[30 30 30]	[15 15 15]	[15 15 15 15]
#weights of C	450	1800	450	675
#blocks of $\mathcal{M}$	49	49	153	153
$CM(\mathcal{M}^{C_{fault}}, \varphi, TS)$	20	24	52	35

repair time, and which is the reduction that must be paid in terms of repair performance.

RQ3: To what extent does ContrRep fix failing tests and break passing tests?
In this RQ, we want to assess whether, while trying to repair the failing tests, ContrRep also breaks some passing tests.

4.2 Benchmarks

To assess the effectiveness of ContrRep, we consider two industrial-level CPSs to be used as plant $\mathcal{M}$, taken from the automotive domain. Both CPSs have been considered in previous works: *Adaptive Cruise Control* [12, 22, 23, 37, 38] and *Abstract Fuel Control* [11, 25]. The CPS is developed in Simulink [24], which in the industry is the de facto standard formalism for modeling control systems.

For each CPS, we have created two different AI-enabled CPS $\mathcal{M}^C$, by empowering them with a different DNN controller C. We employed the approach described in [38, 45] to train the controllers.

Therefore, there is a total of four benchmarks, as shown in Table 1. We also report the complexity of the AI-enabled CPS in terms of: (i) number of Simulink blocks *#blocks* of $\mathcal{M}$, (ii) *structure* of C as the number of neurons in each layer, and (iii) number of weights *#weights* of C.

Adaptive Cruise Control (ACC). ACC [23], as introduced in Example 1, controls the acceleration of the ego car to keep a safe distance of it from a lead car. We consider the following specification, i.e., the relative distance d_{rel} between the two cars should always be greater than the sum of a constant d_{safe} and the braking distance $1.4 \cdot v_{ego}$ of the ego car during $[0, 50]$, and moreover, the speed v_{ego} of the ego car should be lower than 30.

$$\varphi_{ACC} \equiv \square_{[0,50]}(d_{rel} \geq d_{safe} + 1.4 \cdot v_{ego} \wedge v_{ego} \leq 30)$$

Abstract Fuel Control (AFC). AFC is a powertrain control system developed by Toyota [11]. It takes two signals, pedal angle, and engine speed, as the external inputs and produces two output signals AF that indicate the *air-to-fuel* ratio and AF_{ref} that is the reference value of AF. The specification requires that AF should not deviate from AF_{ref} too much, during $[0, 30]$.

$$\varphi_{AFC} \equiv \square_{[0,30]}\left(\left|AF - AF_{ref}\right| \leq 0.2 \cdot AF_{ref}\right)$$

4.2.1 Building faulty benchmarks. In order to properly assess the capability of the compared approaches to repair a DNN controller, we produce faulty versions of the trained controllers. Specifically, for each AI-enabled CPS $\mathcal{M}^C$ and its corresponding specification φ, we proceed as follows.

We generate a test suite TS of 100 tests for $\mathcal{M}^C$ that uniformly cover the input space; for all the benchmarks, all the tests pass, i.e., $CM(\mathcal{M}^C, \varphi, TS) = 1$. Then, we randomly sample 10 weights of C, and we mutate their values, by randomly sampling in $[w_{min}, w_{max}]$, where w_{min} and w_{max} are the minimum and maximum weights values of C; we identify the mutated weights as $\mathcal{W}^{fault} = \{w_1^{fault}, \ldots, w_{10}^{fault}\}$ and the modified AI-enabled CPS as $\mathcal{M}^{C_{fault}}$. If the correctness measure of $\mathcal{M}^{C_{fault}}$ is less than 0.6 (i.e., $CM(\mathcal{M}^{C_{fault}}, \varphi, TS)$), we take $\mathcal{M}^{C_{fault}}$ as the faulty benchmark to repair (i.e., $\mathcal{M}^{C_{orig}}$ in §3); otherwise, we sample other random values for the selected weights. Table 1 reports the obtained values of $CM(\mathcal{M}^{C_{fault}}, \varphi, TS)$.

4.3 Compared approaches

In the experiments, we assess the effectiveness of the proposed repair approaches ContrRep and ContrRep$_{\text{FAST}}$. As explained in §3.1, the approaches require an input of a set of weights SW to repair, usually obtained using some fault localization results.

Following the common practice of assessment of repair approaches in automated program repair [9], for each faulty benchmark $\mathcal{M}^{C_{fault}}$, we built SW in a systematic way. We built three sets of SW, for assessing a repair approach in settings of different complexity:

- $SW_{noNoise}$: we assume that the fault localization approach precisely identified the faulty weights and not others, i.e., $SW_{noNoise} = \mathcal{W}^{fault}$ (see §4.2.1). This setting allows to assess the capabilities of a repair approach, not influenced by possible deficiencies of the fault localization approach;
- SW_2: we assume that the fault localization approach identified, in addition to the faulty weights, two additional weights that are not actually faulty, i.e., $SW_2 = \mathcal{W}^{fault} \cup \{w_1, w_2\}$ with $w_1, w_2 \in \mathcal{W} \setminus \mathcal{W}^{fault}$. This setting allows to assess the capability of a repair approach to handle noise introduced by fault localization;
- SW_4: this setting is similar to SW_2, but four weights in the fault localization results are not faulty, i.e., $SW_4 = \mathcal{W}^{fault} \cup \{w_1, w_2, w_3, w_4\}$ with $w_1, w_2, w_3, w_4 \in \mathcal{W} \setminus \mathcal{W}^{fault}$.

4.3.1 Setting of the search algorithm. ContrRep and ContrRep$_{\text{FAST}}$ use DE as an underlying search algorithm to perform the repair. We set the population size depending on the number of search variables, i.e., the number of suspicious weights in SW; namely, we set the population size to $PopSize = 5 \cdot |SW|$. For all the experiments, we set the maximum number of generations as $NumGens = 50$. For DE, we used the implementation in PlatEMO [35]. We used the default setting because evidence from the literature shows that default settings can provide reasonable results [2].

4.4 Experimental setup

4.4.1 Running of the experiments. An *experiment* consists in the execution of one approach (ContrRep or ContrRep$_{\text{FAST}}$. See §4.3), over a specific faulty AI-enabled CPS (see §4.2.1), using a set of suspicious weights SW (see §4.3). So, in total, there are $2 \times 4 \times 3 = 24$ experiments.

As search algorithms include an element of randomness, by following a guideline on running experiments with randomized algorithms [1], we executed each experiment 10 times.

4.4.2 Evaluation metrics and assessment. To assess the repair effectiveness of a repair approach (RQ1 and RQ2), as an evaluation metric, we consider the value of the correctness measure obtained by the

Table 2: RQ1 and RQ2 – Average *CM* ($\mathcal{M}^{C_{best}}$, φ, *TS*) (%) of ContrRep and ContrRep$_{\text{FAST}}$ across the 10 runs. Values of *CM* for ContrRep$_{\text{FAST}}$ report the exact value and not the approximate value computed by the approach

		ContrRep	ContrRep$_{\text{FAST}}$
ACC#1	$SW_{noNoise}$	80.6	77.3
	SW_2	83.8	77.9
	SW_4	91.5	81.5
ACC#2	$SW_{noNoise}$	80.5	83.3
	SW_2	81.4	79.7
	SW_4	73.2	73.0
AFC#1	$SW_{noNoise}$	71.6	71.3
	SW_2	78.0	77.1
	SW_4	82.6	66.6
AFC#2	$SW_{noNoise}$	72.9	50.7
	SW_2	43.8	43.0
	SW_4	59.6	57.4

best-repaired model, i.e., $CM(\mathcal{M}^{C_{best}}, \varphi, TS)$. For ContrRep$_{\text{FAST}}$, we report the correct value of *CM* (i.e., evaluated over all the tests in *TS*) and not the approximate value assessed by the heuristic method employed by the approach.

To assess the computational cost of a repair approach (RQ2), we consider the total number of test executions along the search *ExecTests*.

When comparing two approaches App1 and App2 over a given metric EM (i.e., *CM* or *ExecTests*), we compare the values obtained across the repetition of the experiments. Specifically, for each faulty benchmark $\mathcal{M}^{C_{fault}}$ and set of suspicious weights *SW*:

- we compare the distributions of 10 values of EM obtained by App1 and by App2, using the Mann-Whitney U test, a non-parametric test that assesses whether there is a significant difference among the distributions. We use $\alpha = 0.05$ as the confidence value of the null hypothesis that there is no significant difference.
- In case of significant difference, we use Vargha and Delaney's $\hat{A}_{12}$ effect size to assess the strength of the significance; if $\hat{A}_{12}$ is greater than 0.5, the results of App1 are significantly higher than those of App2. By following Kitchenham et al.'s classification [15], we identify the following categories of strength: *negligible* when $\hat{A}_{12} \in (0.5, 0.556)$, *small* when $\hat{A}_{12} \in [0.556, 0.638)$, *medium* when $\hat{A}_{12} \in [0.638, 0.714)$, and *large* when $\hat{A}_{12} \geq 0.714$. Similar categories can be identified for $\hat{A}_{12} < 0.5$, i.e., when App1 is significantly lower.

When evaluating *CM*, values of $\hat{A}_{12}$ higher than 0.5 mean that App1 is better. Instead, when evaluating *ExecTests*, values of $\hat{A}_{12}$ lower than 0.5 means that App1 is better.

5 EXPERIMENT RESULTS

Answer to RQ1. In this RQ, we want to assess if ContrRep is actually able to repair the faulty DNN controllers. Table 2 reports, for each approach, the average *CM* across the 10 runs. From the results of Table 2, we observe that we are always able to improve the correctness measure *CM* in $\mathcal{M}^{C_{best}}$ w.r.t. the value of the faulty benchmark $\mathcal{M}^{C_{fault}}$ (see Table 1). We notice that the effectiveness is lower for AFC, where the improvement of *CM* is lower. The physical plant in AFC has indeed complex dynamics that are difficult to control by the DNN controller [11]. For ACC, instead, the control problem is easier. This difference in complexity is witnessed by the difference in *# blocks* of the controlled system $\mathcal{M}$ (see Table 1).

Regarding the performance under different sets of suspicious weights (i.e., $SW_{noNoise}$, SW_2, SW_4), we observe two types of results. In ACC#2 and AFC#2, the values of *CM* decrease when using sets that contain weights that are not faulty (i.e., SW_2 and SW_4), compared to the case in which we use only the faulty weights (i.e., $SW_{noNoise}$). This is somehow expected, as, when using SW_2 and SW_4, the search tries to modify some weights that are not faulty and this is not beneficial for the repair effectiveness: it can break tests passing with the original model, and it wastes search time in trying individuals that are not effective (i.e., those with large changes in the non-faulty weights).

However, in ACC#1 and AFC#1, the value of *CM* increases when using sets that contain weights that are not faulty (i.e., SW_2 and SW_4). Although this is surprising at first, it can be explained by the fact that, as explained in §4.3.1, the population size increases with the number of variables (that corresponds to the size of *SW*) and this allows to perform more exploration of the solution space. In these benchmarks, this higher exploration largely compensates for the noise introduced by trying to modify the non-faulty weights.

Answer to RQ1: ContrRep is able to improve the correctness measure of controllers of AI-enabled CPSs. In some cases, the noise of fault localization results can negatively affect the performance of repair.

Answer to RQ2. In this RQ, we assess to what extent the heuristic applied by ContrRep$_{\text{FAST}}$ can speed up the search and which impacts the repair performance. Table 3 reports the statistical comparison between ContrRep$_{\text{FAST}}$ and ContrRep in terms of executed tests *ExecTests*, and the average number of *ExecTests* across the 10 runs. We observe that ContrRep$_{\text{FAST}}$ is largely significantly better than ContrRep in almost all the benchmarks and fault localization results (except for AFC#1 with SW_4). By looking at the average across the 10 runs of the number of executed tests *ExecTests*, we observe that ContrRep$_{\text{FAST}}$ reduces them by one order of magnitude in 7 out of 12 benchmarks, and, in the other 5 benchmarks, it reduces them to at least half. These results mean that when evaluating the fitness (see Alg. 1), the approach was often able to skip the evaluation of some tests, because some tests in TS^-_{orig} with high robustness was not repaired (so skipping all the other tests in TS^-_{orig} with lower robustness) and/or some test in TS^+_{orig} with low robustness was not broken (so skipping all the other tests in TS^+_{orig} with higher robustness).

Table 4 reports the statistical comparison in terms of *CM* between the approaches. By analyzing the results, we observe that usually ContrRep$_{\text{FAST}}$ is worst in terms of repair performance, confirmed by the results in Table 2. This is expected, as it is the price we need to pay to be faster. However, by comparing the values of *CM* in Table 2 for ContrRep$_{\text{FAST}}$ with the values of *CM* of the original model $\mathcal{M}^{C_{fault}}$ (see Table 1), we observe that ContrRep$_{\text{FAST}}$ is always able to improve the controller.

Table 3: RQ2 – Comparison of ContrRep$_{FAST}$ and ContrRep in terms of number of executed tests *ExecTests*
Legend: (≡: no sign. diff. between the two approaches. ✓: the approach on the left is *better* than the one on top, ✗ means that it is worse; the num. of symbols is the strength: *negligible* (✓, ✗), *small* (✓✓, ✗✗), *medium* (✓✓✓, ✗✗✗), *large* (✓✓✓✓, ✗✗✗✗))

		ContrRep$_{FAST}$ vs. ContrRep	*ExecTests* (avg. 10 runs)	
			ContrRep$_{FAST}$	ContrRep
ACC#1	$SW_{noNoise}$	✓✓✓✓	7.0628e+04	2.5e+05
	SW_2	✓✓✓✓	7.4871e+04	3.0e+05
	SW_4	✓✓✓✓	9.9049e+04	3.5e+05
ACC#2	$SW_{noNoise}$	✓✓✓✓	3.9681e+04	2.5e+05
	SW_2	✓✓✓✓	4.8032e+04	3.0e+05
	SW_4	✓✓✓✓	6.2070e+04	3.5e+05
AFC#1	$SW_{noNoise}$	✓✓✓✓	1.0905e+05	2.5e+05
	SW_2	✓✓✓✓	1.2923e+05	3.0e+05
	SW_4	✓	1.7531e+05	3.07e+05
AFC#2	$SW_{noNoise}$	✓✓✓✓	8.4564e+04	2.5e+05
	SW_2	✓✓✓✓	1.0180e+05	3.0e+05
	SW_4	✓✓✓✓	1.2021e+05	3.5e+05

Table 4: RQ2 – Comparison of ContrRep and ContrRep$_{FAST}$ in terms of *CM* ($\mathcal{M}^{C_{best}}$, φ, *TS*) (legend as in Table 3)

		ContrRep vs. ContrRep$_{FAST}$
ACC#1	$SW_{noNoise}$	✓✓
	SW_2	✓✓✓✓
	SW_4	✓✓✓✓
ACC#2	$SW_{noNoise}$	✗✗
	SW_2	✓✓✓
	SW_4	✓✓
AFC#1	$SW_{noNoise}$	✗
	SW_2	≡
	SW_4	✓✓✓✓
AFC#2	$SW_{noNoise}$	✓✓✓✓
	SW_2	✓✓✓✓
	SW_4	✓✓

From Table 4, we observe an unexpected result: ContrRep$_{FAST}$ is slightly better than ContrRep in ACC#2 for $SW_{noNoise}$ (with strength *small*).[1] We investigated this case and we observed that, in ContrRep, the search finds good individuals that repair failing tests with very low robustness (i.e., the hard ones to repair), but not those with high robustness (i.e., that are close to being satisfied); however, these individuals are not further improved during the search to also repair the other failing tests. In ContrRep$_{FAST}$, the fitness of such individuals is lower than the real *CM*: indeed, since we do not repair the easy tests, we assume that the hard tests are not repairable as well. ContrRep$_{FAST}$, instead, favors the individuals that can repair the easiest tests first. Such individuals turn out to be better in the long term during the search. Thus, it could be that, for some benchmarks, it would be beneficial to prioritize the repair of the easy tests. We leave as future work the investigation of such prioritization during the search.

[1] A similar result occurs for AFC#1 and $SW_{noNoise}$, but with *negligible* strength.

Table 5: RQ3 – Number of repaired and broken tests (average across 10 runs)

		ContrRep				ContrRep$_{FAST}$			
		TS^-_{orig}		TS^+_{orig}		TS^-_{orig}		TS^+_{orig}	
		repaired	not repaired	preserved	broken	repaired	not repaired	preserved	broken
ACC#1	$SW_{noNoise}$	75.89%	24.11%	99.29%	0.71%	71.61%	28.39%	100.00%	0.00%
	SW_2	79.64%	20.36%	100.00%	0.00%	72.32%	27.68%	100.00%	0.00%
	SW_4	89.38%	10.63%	100.00%	0.00%	76.88%	23.13%	100.00%	0.00%
ACC#2	$SW_{noNoise}$	74.34%	25.66%	100.00%	0.00%	78.03%	21.97%	100.00%	0.00%
	SW_2	75.79%	24.21%	100.00%	0.00%	73.29%	26.71%	100.00%	0.00%
	SW_4	65.13%	34.87%	100.00%	0.00%	64.80%	35.20%	100.00%	0.00%
AFC#1	$SW_{noNoise}$	47.02%	52.98%	94.23%	5.77%	43.75%	56.25%	96.70%	3.30%
	SW_2	62.50%	37.50%	92.31%	7.69%	64.84%	35.16%	88.46%	11.54%
	SW_4	67.50%	32.50%	97.69%	2.31%	48.75%	51.25%	79.62%	20.38%
AFC#2	$SW_{noNoise}$	66.84%	33.16%	84.13%	15.87%	29.23%	70.77%	90.48%	9.52%
	SW_2	32.31%	67.69%	65.14%	34.86%	18.62%	81.38%	88.29%	11.71%
	SW_4	36.92%	63.08%	64.00%	36.00%	29.85%	70.15%	85.71%	14.29%

Answer to RQ2: Overall, ContrRep$_{FAST}$ allows us to significantly speed up the search, at the cost of a decrease in repair effectiveness. Still, ContrRep$_{FAST}$ can improve the controllers.

Answer to RQ3. In this RQ, we are interested in assessing how ContrRep and ContrRep$_{FAST}$ improve the correctness measure, i.e., to what extent they repair the failing tests in TS^-_{orig} and preserve those in TS^+_{orig}. Table 5 reports, for all the benchmarks and the two approaches, the percentages of tests in TS^-_{orig} that are *repaired* and *not repaired*, and the percentages of tests in TS^+_{orig} that are *preserved* and *broken*. From the table, we observe that for ACC benchmarks, tests in TS^+_{orig} that were passing with the original AI-enabled CPS $\mathcal{M}^{C_{fault}}$ are almost never broken and still pass with the repaired system $\mathcal{M}^{C_{best}}$. In AFC benchmarks, instead, some tests in TS^+_{orig} are broken, up to 36%.

Depending on the application domain, this could be a problem, as some stakeholders may prefer not to break previously correct behaviors to avoid regression. Note that, in our approaches, we are only interested in maximizing the number of passing tests in $\mathcal{M}^{C_{best}}$, so we do not take any measure to avoid breaking tests in TS^+_{orig}, and we accept it if this allows us to increase the number of repaired tests in TS^-_{orig}. We will consider as future work a version of the approach that aims at repairing, without breaking previously correct tests.

Regarding the number of repaired tests, we can never repair all the tests, with lower performance in AFC#2 that, as seen in RQ1, is the benchmark in which we obtain the worst results.

Answer to RQ3: Depending on the benchmark, the percentage of tests in TS^+_{orig} that are broken varies, with higher break rates for the most difficult benchmarks. Similarly, the percentage of tests in TS^-_{orig} that are repaired depends on the difficulty of the benchmark.

6 THREATS TO VALIDITY

The validity of ContrRep could be affected by different threats [40].

Construct validity. A threat of this type could be that the metrics we use for assessing the compared approaches (i.e., ContrRep and ContrRep$_{\text{FAST}}$) are not suitable. In our context, the goal of repair is to improve the performance of the AI-enabled CPS over the test suite *TS*, which is measured by *CM* (see Eq. 1); therefore, using *CM* as assessment metric is correct. ContrRep$_{\text{FAST}}$ has been introduced to speed up the search by avoiding executing all the tests whose execution is computationally expensive; therefore, using the number of executed tests *ExecTests* to compare ContrRep and ContrRep$_{\text{FAST}}$ is suitable.

Conclusion validity. ContrRep and ContrRep$_{\text{FAST}}$ adopt the search algorithm DE to perform the repair. The randomness of DE can affect the results. Therefore, to account for the nondeterministic nature of the repair approaches, we repeat each experiment 10 times as suggested by guidelines on running experiments with randomized algorithms [1].[2] Moreover, we have also used suitable statistical tests to compare the approach by considering both significant differences and effect size.

Internal validity. A threat of this type is that the obtained results are by chance, for example, due to an *instrumentation* threat [40] (e.g., faulty implementation). To mitigate this threat, we have carefully tested the implementation of ContrRep and ContrRep$_{\text{FAST}}$.

External validity. It could be that the performance of ContrRep does not generalise to other AI-enabled CPS and DNN controllers. To mitigate this threat, we have considered two industrial-scale CPSs that are widely used in competitions on testing CPSs [25] and DNN controllers [22].

7 RELATED WORK

Different works have been proposed to test machine learning systems and systems empowered by machine learning components; please refer to [30, 44] for recent surveys. In this work, we focus on the activities that follow testing, specifically DNN repair.

DNN repair has been inspired by automatic program repair for classic code; we refer the interested reader to recent surveys [9, 16, 26]. Here we review related work on DNN repair.

A simple way of repairing the DNN consists of *retraining* it by using additional inputs. For example, Zhang et al. [43] introduced the approach *Apricot* which retrains the DNN model by using different subsets of the training dataset and then merges them. Fahmy et al. [7] proposed the approach *HUDD*, a repair approach for DNN classifiers; the approach identifies the causes of failures by clustering the heatmaps of the behaviors of each DNN layer, and then uses the images that more likely lead to errors for retraining. Yu et al. [42] introduced *DeepRepair*, which aims to improve a DNN classifier against unknown noise that could be present in the operational environment; it learns unknown failure patterns and introduces them to retraining data via style transfer. Our approach differs from all the previous approaches, as it considers DNN controllers (and not classifiers), it is not based on retraining, and it does not require ground truth for the DNN behaviors.

An issue of retraining is that it modifies all the weights of the network, and so it may not be able to repair the behavior of the failing inputs without introducing other failures. To address this problem, a more popular approach consists of employing search-based approaches to modify a few DNN components, usually identified by some fault localization approach [6, 31]. For example, Sohn et al. [31] introduced *Arachne* that identifies the weights that are responsible for misclassification and then uses differential evolution to do the repair. Sun et al. [34] proposed a similar approach, but using PSO as a search algorithm. Li Calsi et al. [18] proposed *DistrRep* that extends *Arachne* by considering different misclassifications: it repairs each misclassification individually and then merges the different models into a final repaired model. Tokui et al. [36] proposed *NeuRecover* which focuses on the weights that were critical during the training history. Our approach shares with the previous approaches the use of a search-based approach for modifying the weights; however, differently from them, it uses system-level specifications to guide the repair.

A different family of repair approaches [14, 28, 29] considers different types of faults [10] at the architectural level (e.g., wrong choice of the optimizer). These approaches target completely different problems than ours, and so a direct comparison is difficult.

Some works have been proposed for repairing the CPS (i.e., the plant $\mathcal{M}$ of $\mathcal{M}^C$). For example, Valle et al. [39] proposed a search-based approach for repairing the configuration of an elevator system. Jung et al. [13] repair misconfigurations of swarm robots. Ben Abdessalem et al. [3], instead, repair feature interaction bugs of autonomous driving systems. These works are complementary to ours, as they repair the physical plant, while we repair the controller. Future work could consider a combined repair of both.

8 CONCLUSION

AI-enabled CPSs use DNNs as controllers of the physical plant. Despite their advantages, DNN controllers can be faulty, leading to wrong control decisions. We propose ContrRep, a search-based repair approach for DNN controllers, guided by system-level specifications. We further introduce ContrRep$_{\text{FAST}}$, a modification of the approach that tries to speed up the search. Experiments show that ContrRep is indeed able to repair faulty DNN controllers, and that ContrRep$_{\text{FAST}}$ can greatly reduce the computational complexity at a small price in terms of repair effectiveness.

In this work, we assume that the plant is correct and that the DNN controller must be fixed. However, some plants have hyperparameters that can be tuned, which can affect the behavior of the AI-enabled CPS. In future work, we plan to investigate the combined repair of the physical plant and the DNN controller.

ACKNOWLEDGMENTS

P. Arcaini and F. Ishikawa are supported by Engineerable AI Techniques for Practical Applications of High-Quality Machine Learning-based Systems Project (Grant Number JPMJMI20B8), JST-Mirai. Z. Zhang and J. Zhao are supported by JSPS KAKENHI Grant No. JP23H03372. Z. Zhang is also supported by JSPS KAKENHI Grant No. JP23K16865.

[2] Although 30 runs are suggested, it is also recognized in [1] that this may be not possible when using computationally expensive systems as in our approach. In this case, particular attention must be paid to the statistical assessment of the results.

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2011. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1985793.1985795

[2] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18 (2013), 594–623. https://doi.org/10.1007/s10664-013-9249-9

[3] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2020. Automated repair of feature interaction failures in automated driving systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 88–100. https://doi.org/10.1145/3395363.3397386

[4] Jialun Cao, Meiziniu Li, Xiao Chen, Ming Wen, Yongqiang Tian, Bo Wu, and Shing-Chi Cheung. 2022. DeepFD: Automated Fault Diagnosis and Localization for Deep Learning Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 573–585. https://doi.org/10.1145/3510003.3510099

[5] Matias Duran, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2021. What to Blame? On the Granularity of Fault Localization for Deep Neural Networks. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 264–275. https://doi.org/10.1109/ISSRE52982.2021.00037

[6] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, 171–191.

[7] Hazem Fahmy, Fabrizio Pastore, Mojtaba Bagherzadeh, and Lionel Briand. 2021. Supporting Deep Neural Network Safety Analysis and Retraining Through Heatmap-Based Unsupervised Learning. *IEEE Transactions on Reliability* (2021), 1–17. https://doi.org/10.1109/TR.2021.3074750

[8] Georgios E. Fainekos and George J. Pappas. 2009. Robustness of Temporal Logic Specifications for Continuous-Time Signals. *Theor. Comput. Sci.* 410, 42 (Sept. 2009), 4262–4291. https://doi.org/10.1016/j.tcs.2009.06.021

[9] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* 45, 1 (jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[10] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of Real Faults in Deep Learning Systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3377811.3380395

[11] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. 2014. Powertrain Control Verification Benchmark. In *Proceedings of the 17th International Conference on Hybrid Systems: Computation and Control* (Berlin, Germany) *(HSCC '14)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/2562059.2562140

[12] Taylor T Johnson, Diego Manzanas Lopez, Patrick Musau, Hoang-Dung Tran, Elena Botoeva, Francesco Leofante, Amir Maleki, Chelsea Sidrane, Jiameng Fan, and Chao Huang. 2020. ARCH-COMP20 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20) (EPiC Series in Computing, Vol. 74)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 107–139. https://doi.org/10.29007/9xgv

[13] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian Elbaum, and Yonghwi Kwon. 2021. Swarmbug: debugging configuration bugs in swarm robotics. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 868–880. https://doi.org/10.1145/3468264.3468601

[14] Jinhan Kim, Nargiz Humbatova, Gunel Jahangirova, Paolo Tonella, and Shin Yoo. 2023. Repairing DNN Architecture: Are We There Yet?. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 234–245. https://doi.org/10.1109/ICST57152.2023.00030

[15] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Softw. Engg.* 22, 2 (apr 2017), 579–630. https://doi.org/10.1007/s10664-016-9437-5

[16] Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic Program Repair. *IEEE Software* 38, 4 (2021), 22–27. https://doi.org/10.1109/MS.2021.3072577

[17] Davide Li Calsi, Matias Duran, Thomas Laurent, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2023. Adaptive Search-Based Repair of Deep Neural Networks. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Lisbon, Portugal) *(GECCO '23)*. Association for Computing Machinery, New York, NY, USA, 1527–1536. https://doi.org/10.1145/3583131.3590477

[18] Davide Li Calsi, Matias Duran, Xiao-Yi Zhang, Paolo Arcaini, and Fuyuki Ishikawa. 2023. Distributed Repair of Deep Neural Networks. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 83–94. https://doi.org/10.1109/ICST57152.2023.00017

[19] Davide Li Calsi, Thomas Laurent, Paolo Arcaini, and Fuyuki Ishikawa. 2024. Federated Repair of Deep Neural Networks. In *2024 IEEE/ACM International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)*.

[20] Wenliang Liu, Noushin Mehdipour, and Calin Belta. 2022. Recurrent Neural Network Controllers for Signal Temporal Logic Specifications Subject to Safety Constraints. *IEEE Control Systems Letters* 6 (2022), 91–96. https://doi.org/10.1109/LCSYS.2021.3049917

[21] Deyun Lyu, Zhenya Zhang, Paolo Arcaini, Fuyuki Ishikawa, Thomas Laurent, and Jianjun Zhao. 2024. Supplementary Material for the paper "Search-Based Repair of DNN Controllers of AI-Enabled Cyber-Physical Systems Guided by System-Level Specifications". https://github.com/lyudeyun/ContrRep.

[22] Diego Manzanas Lopez, Matthias Althoff, Marcelo Forets, Taylor T Johnson, Tobias Ladner, and Christian Schilling. 2023. ARCH-COMP23 Category Report: Artificial Intelligence and Neural Network Control Systems (AINNCS) for Continuous and Hybrid Systems Plants. In *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23) (EPiC Series in Computing, Vol. 96)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 89–125. https://doi.org/10.29007/x38n

[23] Mathworks. 2023. Adaptive Cruise Control System Using Model Predictive Control. https://www.mathworks.com/help/mpc/ug/adaptive-cruise-control-using-model-predictive-controller.html

[24] Mathworks. 2023. Simulink. https://www.mathworks.com/products/simulink.html

[25] Claudio Menghi, Paolo Arcaini, Walstan Baptista, Gidon Ernst, Georgios Fainekos, Federico Formica, Sauvik Gon, Tanmay Khandait, Atanu Kundu, Giulia Pedrielli, Jarkko Peltomäki, Ivan Porres, Rajarshi Ray, Masaki Waga, and Zhenya Zhang. 2023. ARCH-COMP23 Category Report: Falsification. In *Proceedings of 10th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH23) (EPiC Series in Computing, Vol. 96)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 151–169. https://doi.org/10.29007/6nqs

[26] Martin Monperrus. 2018. Automatic software repair: a bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.

[27] Scott A. Nivison and Pramod P. Khargonekar. 2018. Development of a Robust, Sparsely-Activated, and Deep Recurrent Neural Network Controller for Flight Applications. In *2018 IEEE Conference on Decision and Control (CDC)* (FL, USA). IEEE Press, 384–390. https://doi.org/10.1109/CDC.2018.8619188

[28] Hua Qi, Zhijie Wang, Qing Guo, Jianlang Chen, Felix Juefei-Xu, Fuyuan Zhang, Lei Ma, and Jianjun Zhao. 2023. ArchRepair: Block-Level Architecture-Oriented Repairing for Deep Neural Networks. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 129 (jul 2023), 31 pages. https://doi.org/10.1145/3585005

[29] Xuhong Ren, Jianlang Chen, Felix Juefei-Xu, Wanli Xue, Qing Guo, Lei Ma, Jianjun Zhao, and Shengyong Chen. 2022. DARTSRepair: Core-failure-set guided DARTS for network robustness to common corruptions. *Pattern Recognition* 131 (2022), 108864.

[30] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empir. Softw. Eng.* 25, 6 (2020), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0

[31] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2023. Arachne: Search-Based Repair of Deep Neural Networks. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 85 (may 2023), 26 pages. https://doi.org/10.1145/3563210

[32] Jiayang Song, Deyun Lyu, Zhenya Zhang, Zhijie Wang, Tianyi Zhang, and Lei Ma. 2022. When Cyber-Physical Systems Meet AI: A Benchmark, an Evaluation, and a Way Forward. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 343–352. https://doi.org/10.1145/3510457.3513049

[33] Rainer Storn and Kenneth Price. 1997. Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *J. of Global Optimization* 11, 4 (dec 1997), 341–359. https://doi.org/10.1023/A:1008202821328

[34] Bing Sun, Jun Sun, Long H. Pham, and Jie Shi. 2022. Causality-Based Neural Network Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 338–349. https://doi.org/10.1145/3510003.3510080

[35] Ye Tian, Ran Cheng, Xingyi Zhang, and Yaochu Jin. 2017. PlatEMO: A MATLAB Platform for Evolutionary Multi-Objective Optimization. *Comp. Intell. Mag.* 12, 4 (nov 2017), 73–87. https://doi.org/10.1109/MCI.2017.2742868

[36] Shogo Tokui, Susumu Tokumoto, Akihito Yoshii, Fuyuki Ishikawa, Takao Nakagawa, Kazuki Munakata, and Shinji Kikuchi. 2022. NeuRecover: Regression-Controlled Repair of Deep Neural Networks with Training History. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1111–1121. https://doi.org/10.1109/SANER53432.2022.00128

[37] Hoang-Dung Tran, Feiyang Cai, Manzanas Lopez Diego, Patrick Musau, Taylor T. Johnson, and Xenofon Koutsoukos. 2019. Safety Verification of Cyber-Physical Systems with Reinforcement Learning Control. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 105 (Oct. 2019), 22 pages. https://doi.org/10.1145/3358230

[38] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T Johnson. 2020. NNV: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In *International Conference on Computer Aided Verification*. Springer, 3–17.

[39] Pablo Valle, Aitor Arrieta, and Maite Arratibel. 2023. Automated Misconfiguration Repair of Configurable Cyber-Physical Systems with Search: An Industrial Case Study on Elevator Dispatching Algorithms. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice* (Melbourne, Australia) *(ICSE-SEIP '23)*. IEEE Press, 396–408. https://doi.org/10.1109/ICSE-SEIP58684.2023.00042

[40] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[41] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.* 42, 8 (aug 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[42] Bing Yu, Hua Qi, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, and Jianjun Zhao. 2021. DeepRepair: Style-Guided Repairing for Deep Neural Networks in the Real-World Operational Environment. *IEEE Transactions on Reliability* (2021), 1–16. https://doi.org/10.1109/TR.2021.3096332

[43] Hao Zhang and W. K. Chan. 2019. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 376–387. https://doi.org/10.1109/ASE.2019.00043

[44] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Trans. Softw. Eng.* 48, 1 (jan 2022), 1–36. https://doi.org/10.1109/TSE.2019.2962027

[45] Zhenya Zhang, Deyun Lyu, Paolo Arcaini, Lei Ma, Ichiro Hasuo, and Jianjun Zhao. 2023. FalsifAI: Falsification of AI-Enabled Hybrid Control Systems Guided by Time-Aware Coverage Criteria. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1842–1859. https://doi.org/10.1109/TSE.2022.3194640