Causal Repair of Learning-Enabled Cyber-Physical Systems

Pengyuan Lu
Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
pelu@seas.upenn.edu

Ivan Ruchkin
Electrical and Computer Engineering
University of Florida
Gainesville, FL, USA
iruchkin@ece.ufl.edu

Matthew Cleaveland
Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
mcleav@seas.upenn.edu

Oleg Sokolsky
Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
sokolsky@cis.upenn.edu

Insup Lee
Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
lee@cis.upenn.edu

2023 IEEE International Conference on Assured Autonomy (ICAA) | 979-8-3503-2601-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICAA58325.2023.00009

***Abstract*—Models of actual causality leverage domain knowledge to generate convincing diagnoses of events that caused an outcome. It is promising to apply these models to diagnose and repair run-time property violations in cyber-physical systems (CPS) with learning-enabled components (LEC). However, given the high diversity and complexity of LECs, it is challenging to encode domain knowledge (e.g., the CPS dynamics) in a scalable actual causality model that could generate useful repair suggestions. In this paper, we focus causal diagnosis on the input/output behaviors of LECs. Specifically, we aim to identify which subset of I/O behaviors of the LEC is an actual cause for a property violation. An important by-product is a counterfactual version of the LEC that repairs the run-time property by fixing the identified problematic behaviors. Based on this insights, we design a two-step diagnostic pipeline: (1) construct and Halpern-Pearl causality model that reflects the dependency of property outcome on the component's I/O behaviors, and (2) perform a search for an actual cause and corresponding repair on the model. We prove that our pipeline has the following guarantee: if an actual cause is found, the system is guaranteed to be repaired; otherwise, we have high probabilistic confidence that the LEC under analysis did not cause the property violation. We demonstrate that our approach successfully repairs learned controllers on a standard OpenAI Gym benchmark.**

***Index Terms*—actual causality, control policy repair, cyber-physical system.**

I. Introduction

When a person's leg hurts, they seek detailed diagnosis from a physician regarding the pain's cause, which would lead to an effective intervention as a "repair". Similarly, when a closed-loop cyber-physical system (CPS) violates a desirable property at run time, the violation needs diagnosis — a procedure that identifies the cause for the violation, and by fixing the identified cause, the CPS can be repaired. Traditionally, researchers conduct this kind of analysis from statistical inference on observations [1], [2] . However, statistical diagnosis is prone to mistaking correlation for causation: when a student always wears a green jacket and fails several exams, such algorithms are likely to conclude it is the green jacket's fault due to the perfect correlation. Therefore, in this paper, we focus on stronger causal reasoning and repair on CPS failures.

Researchers have explored the concept of *actual causality* that leverages domain knowledge to produce well-defined and convincing causal explanations. Informally, an actual cause for an outcome is a minimal set of variable assignments, which represent an event, that changes the outcome if assigned some counterfactual values. Finding an actual cause requires the construction of an actual causality model, such as a Halpern-Pearl model, which rigorously defines actual causes for events and precisely assigns responsibility and blame [3]–[5]. With these models, we can encode common knowledge that, for instance, the student's bad grade can be either due to not studying hard enough or misunderstanding some concepts in class.

Although actual causality is promising for analyzing and fixing CPS failures, causal analysis and repair have been complicated by the growing popularity of learning-enabled components (LECs) [6]–[8]. First, LECs usually consist of numerous internal continuous parameters, such as weights and biases in deep neural networks. Second, LECs take a large diversity of forms, from basic statistical models such as linear regressors and support vector machines to complex deep architectures, lacking a shared struture of a parameter template. Furthermore, sometimes the internal structures of LECs are black-boxes protected as intellectual properties, and are invisible to testing engineers. Under these scenarios, it is hard to build a causal model based on the internal information flow of these components. Related to diagnosis and repair are the efforts in explainable AI [9], [10] and formal methods [11], [12], where researchers build frameworks to explain behaviors of learned agents. However, to our knowledge, these strands of work have yet to connect to actual causality.

Since the internal parameters of LECs are hard to analyze, we take a step back and analyze the LEC I/O behavior instead. Our intention in this paper is to leverage actual causality to efficiently identify the granular I/O behaviors of

979-8-3503-2601-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICAA58325.2023.00009

a suspected LEC for a run-time property violation. We intend to identify these behaviors in the form of fine-grained input-output mappings, so that a precise repair can be made for CPS to satisfy the desired property.

To achieve this goal, we first construct an Halpern-Pearl (HP) causality model to encode the dependency of property outcome on the suspected LEC's behaviors. Then, we design an algorithm to search for an actual cause and corresponding repair on this HP model. This algorithm either concludes that an actual cause does not exist within the LEC's behaviors with high confidence , or outputs the set of behaviors that are causing the violation, along with a counterfactual repair. Experiments on the mountain car, a standard OpenAI Gym benchmark [13], demonstrates the capability of our method.

To summarize, our contributions are:

1) A novel use of HP causality model to identify problematic LEC I/O behaviors that cause a run-time property violation, and produce repair suggestions, without touching the component's internal information flow, and
2) Experiments on an OpenAI Gym benchmark that show our solution's utility for learning-enabled CPS repair.

II. Background and Related Work

A. Learning-enabled Components (LEC)

Learning-enabled components (LECs) are functional components of larger systems that are learned from data, either offline or online. One example is the perception unit, with neural networks able to complete difficult vision tasks that generally cannot be accomplished by traditional first-principles algorithms, such as tracking moving objects at 100 fps [6]. Moreover, deep reinforcement learning has provided useful control policies on various tasks as a substitute for conventional controllers [14], [15]. Due to LECs' performance, the state-of-the-art in CPS has a growing enthusiasm of these statistically generated agents, with automated design tools built for CPS with LECs [7], as well as analysis on their assurance on run-time properties such as safety [8]. Consequently, we need to consider the presence of LECs when diagnosing faults in CPS.

B. Repair

A repair is a procedure to change or replace parts of a system to achieve a desirable performance, of which the system previously fell short. This term has been widely used in traditional embedded systems. For example, researchers have studied repair on hardware components such as DRAM [16], [17] or noisy sensors [18]. Also, repair has been performed on software; for example, assembly program can be transformed to decrease resource consumption below a threshold [19].

In modern CPS with LEC, the topic of repair interests many researchers. Learned agents, such as deep neural networks, may lead the system to unsafe states [20] or unplanned paths [21]. Therefore, repair is necessary to recover desirable behaviors. Repair of neural networks is an active area of research [22], [23]. For instance, network parameters can be repaired by search algorithms [24] or constraint solvers [25]. More recently, researchers have studied provable repairs on deep neural networks, with guaranteed satisfaction of a given property after the repair [26]–[28].

Compared to the existing work on repair, we focus on the causal relationships between the repairable elements and the execution outcome. That is, the neural network parameters to be repaired should be the ones that caused the system's failure.

C. Actual Causality and Halpern-Pearl Models

Below we rephrase Halpern and Pearl's definition of actual causality [3], [4]. An actual causality model, or a Halpern-Pearl (HP) model is a recursive structure, i.e., a directed acyclic graph, such that every node represents either an *exogenous variable*, whose value is determined by factors outside this model, or an *endogenous variable*, whose value is determined by other variables in this model. The edges represent dependencies: an exogenous node has only outgoing edges but no incoming edges, while an endogenous node may have both. Every endogenous node is equipped with a function, which defines how the node's value is computed from other nodes. In other words, this function defines the incoming edges to the node. Formally,

1) An *HP model* is a tuple $\mathcal{M} = (\mathcal{U}_{\text{endo}}, \mathcal{U}_{\text{exo}}, \mathcal{V}, \mathcal{E})$, where $\mathcal{U}_{\text{endo}}$ and $\mathcal{U}_{\text{exo}}$ are finite sets of endogenous and exogenous variables, respectively, and for each variable $u \in \mathcal{U}_{\text{endo}} \cup \mathcal{U}_{\text{exo}}$, $\mathcal{V}(u)$ defines a non-empty and potentially infinite set of values that u can take.
2) $\mathcal{E}$ is a set of edges, associated with dependency equations, that defines how the value of each endogenous node u is computed. I.e., for each $e_u \in \mathcal{E}$,

$$e_u : \prod_{u' \in \mathcal{U}_{\text{endo}} \setminus \{u\}} \mathcal{V}(u') \times \prod_{u'' \in \mathcal{U}_{\text{exo}}} \mathcal{V}(u'') \mapsto \mathcal{V}(u).$$

To define an actual cause in an HP model $\mathcal{M}$, we introduce the following notation:

1) An *assignment* of a variable $u \in \mathcal{U}_{\text{endo}} \cup \mathcal{U}_{\text{exo}}$ is denoted as $u := v$, for some $v \in \mathcal{V}(u)$. Assignments of multiple variables are denoted in vector form $\mathbf{u} := \mathbf{v}$, with $\mathbf{u} = [u_1, u_2, \dots]$, $\mathbf{v} = [v_1, v_2, \dots]$. This assignment means a conjunction $(u_1 := v_1) \wedge (u_2 := v_2) \wedge \dots$.
2) A *property* ψ is a Boolean function of endogenous variable assignments, e.g., $\psi = (u_1 := v_1) \wedge (u_2 := v_2) \vee (u_3 := v_3)$. The satisfaction relation $(\mathcal{M}, \mathbf{v}_{\text{cxt}}) \models \psi$ denotes that a property holds on $\mathcal{M}$ given exogenous nodes assigned with $\mathbf{v}_{\text{cxt}}$.
3) A *counterfactual* $\mathbf{v}'$ (with respect to the "factual" $\mathbf{v}$) is an alternative value assignment on some endogenous variables. The values of dependent nodes in $\mathbf{v}'$ may be different from $\mathbf{v}$ in accordance with to $\mathcal{E}$. A property ψ holds on a counterfactual that replaces the factual values $\mathbf{v}$ with $\mathbf{v}'$ on variables $\mathbf{u}$ is denoted as $(\mathcal{M}, \mathbf{v}_{\text{cxt}}) \models [\mathbf{u} \leftarrow \mathbf{v}']\psi$, or simply $[\mathbf{u} \leftarrow \mathbf{v}']\psi$.

Then, in the above terms, the Halpern and Pearl definition of an actual cause is phrased as follows.

Definition 1 (Actual Cause)**.** *On an HP model $\mathcal{M}$ with exogenous node values $\mathbf{v}_{cxt}$, the assignment on a set of endogenous*

variables $\mathbf{u} := \mathbf{v}$ *is an actual cause of* ψ *iff the following conditions hold:*

1) *AC1:* $(\mathcal{M}, \mathbf{v}_{cxt}) \models (\mathbf{u} := \mathbf{v}) \wedge \psi$
2) *AC2:* $\exists$ *partition* $\mathcal{U}_{endo} = \mathbf{u} \cup \mathbf{u}_1 \cup \mathbf{u}_2$. *Denote the factual values of* $\mathbf{u}_1$ *and* $\mathbf{u}_2$ *as* $\mathbf{v}_1$ *and* $\mathbf{v}_2$, *respectively. Then,* $\exists \mathbf{v}', \mathbf{v}_1'$, *such that*
   (a) $[\mathbf{u} \leftarrow \mathbf{v}', \mathbf{u}_1 \leftarrow \mathbf{v}_1']\neg\psi$
   (b) $[\mathbf{u} \leftarrow \mathbf{v}, \mathbf{u}_1 \leftarrow \mathbf{v}_1', \mathbf{u}_2^* \leftarrow \mathbf{v}_2^*]\psi$, *for any* $\mathbf{u}_2^* \subseteq \mathbf{u}_2$ *and* $\mathbf{v}_2^*$ *is the original factual value of* $\mathbf{u}_2^*$ *(a subvector of* $\mathbf{v}_2$*).*
3) *AC3:* $\nexists \mathbf{u}' \subset \mathbf{u}$ *that satisfies AC1 and AC2.*

Condition AC1 ensures the suspected actual cause and outcome are factual. Then, AC2(a) ensures a sufficient counterfactual, that switching the suspect $\mathbf{u}$ and a circumstance $\mathbf{u}_1$ to that counterfactual assignment guarantees a flipped outcome $\neg\psi$, and AC2(b) states that switching the circumstance alone does not change the outcome - as long as the suspect remains the factual value assignments. Finally, AC3 guarantees minimality of the actual cause. Detailed explanation for this definition can be found in the original publications [3], [4].

Researchers have applied actual causality and HP model to CPS diagnosis. For instance, Ibrahim et al. have designed a SAT solver to practically compute actual causes to explain undesirable CPS behaviors [29], [30]. However, the solver is restricted to finite $\mathcal{V}(u)$ for variables and their HP model design only captures discrete events like "there exists a Byzantine fault" or "the system is on autopilot mode". Unfortunately, this design does not extend to diagnosis of LECs, which generally have continuous value spaces for I/O and internal variables, and their internal information flows are not interpretable.

III. Problem Formulation

A. System Setting

We formalize the system setting as follows. We have an exact dynamical system model $\mathcal{S}$, which describes how the components of the agent interact with each other, as well as how the agent interacts with the environment. This closed-loop model is assumed to be deterministic. In other words, the agent's trajectory only depends on the initial state, the environment dynamics, and the component designs. These assumptions are often satisfied in model-based CPS engineering and can be relaxed in future research.

The system aims to satisfy a given specification/property φ that takes Boolean values (true/satisfied/1, false/violated/0), e.g., the linear temporal logic (LTL) [31] or signal temporal logic (STL) [32] formula. In this paper, we focus on STL.

We observe a trajectory on a given initial state s_0, where the system failed to satisfy φ and we suspect one of its components $\mathcal{C}$, such as a controller, is at fault for the violation. The component $\mathcal{C}$'s internal design is invisible to us, but we can observe its input/output (I/O) behavior, which we denote by function $f : \mathcal{X} \mapsto \mathcal{Y}$, with its domain and codomain being *continuous and bounded* metric spaces $(\mathcal{X}, d_{\mathcal{X}})$ and $(\mathcal{Y}, d_{\mathcal{Y}})$, respectively. Therefore, we can replace this behavior by any *counterfactual* $f' : \mathcal{X} \mapsto \mathcal{Y}$. We denote the counterfactual system that uses f' in place of f, with everything else remaining the same, as $\mathcal{S}(f')$. We use $\mathcal{S}(f') \models \varphi$ and $\mathcal{S}(f') \not\models \varphi$ to denote that the property will be satisfied or violated under $\mathcal{S}(f')$. With this notation, the factual outcome is $\mathcal{S}(f) \not\models \varphi$. Next, we formalize the distance between two choices for the behaviors of component $\mathcal{C}$.

Definition 2 (Distance between I/O behaviors)**.** *For any two I/O behaviors* $f_1, f_2 : \mathcal{X} \mapsto \mathcal{Y}$, *we define distance* $||\cdot||_{d_f}$ *as*

$$||f_1 - f_2||_{d_f} = \max_{x \in \mathcal{X}} ||f_1(x) - f_2(x)||_{d_{\mathcal{Y}}} \tag{1}$$

We then make the following assumptions.

Assumption 1. *We can check the outcome of* φ *on the given initial state* s_0 *when substituting different* f' *in place of the component by calling a simulator,*

$$\text{SIMULATE}_{s_0} : (\mathcal{X} \mapsto \mathcal{Y}) \mapsto \{0, 1\}, \tag{2}$$

which encodes the knowledge of $\mathcal{S}$ *and* φ*. For simplicity, we assume a fixed* s_0 *in the remainder of this paper, and the simulator is denoted as simply* SIMULATE*.*

Assumption 2. *The behavior* f *is Lipschitz-continuous, with an unknown Lipschitz constant. This is a common property of many types of learning models, such as neural networks.*

Assumption 3. *The property outcome is robust against small changes from the factual I/O behavior* f*, i.e.,* $\forall f' : \mathcal{X} \mapsto \mathcal{Y}$,

$$||f - f'||_{d_f} \leq \epsilon \implies \text{SIMULATE}(f) = \text{SIMULATE}(f'), \tag{3}$$

for some small $\epsilon > 0$*.*

We expect Assumption 3 to hold in most practical cases. Suppose the distance of the factual f to the decision boundary of the simulator outcome on I/O behaviors is δ. If $\delta > 0$, there exists an arbitrarily small $\epsilon < \delta$ where the assumption holds. The only case that Assumption 3 does not hold is when $\delta = 0$, i.e., the factual behavior is right on the decision boundary, but this event would usually have a probability measure of 0.

B. Problem Statement

In the above setting, we want to identify a subset of I/O behaviors of the suspected component $\mathcal{C}$ — that is, a set of input-output tuples of f — that indeed caused the property violation, as well as the counterfactual outputs on these inputs that can repair the system.

Main Problem. Upon the observation of a property violation on a runtime trace, $\mathcal{S}(f) \not\models \varphi$, how can we use HP causality to identify a subset of the suspected component $\mathcal{C}$'s I/O behaviors (modeled as f) that caused this violation?

1) **Sub-problem 1.** Encode the dependency structure of $\mathcal{C}$'s behaviors on φ using an HP model.
2) **Sub-problem 2.** On the encoded HP model, design a search algorithm for an actual cause, such that, upon success, provide a repair suggestion in form of a counterfactual $f^* : \mathcal{X} \mapsto \mathcal{Y}$ that $\mathcal{S}(f^*) \models \varphi$. Upon failure, quantify the confidence that the property violation is not caused by the I/Os of $\mathcal{C}$.

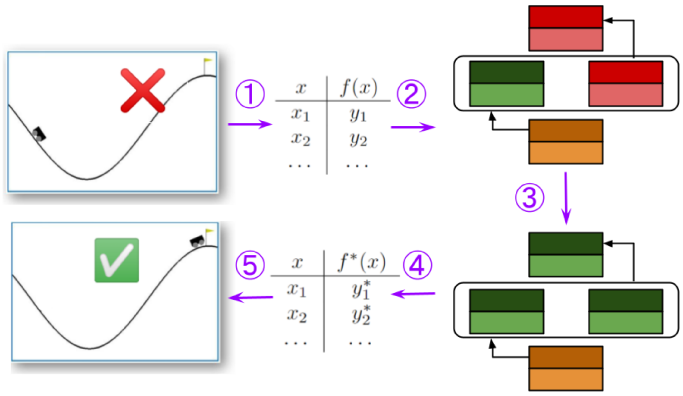

Fig. 1: Workflow of our causal repair approach, which constructs an HP model to encode the I/O behaviors of a suspected LEC and search for a repair by causal analysis.

To solve these problems, we employ the following workflow, which is visualized in Figure 1: (1) Extract the behaviors of $\mathcal{C}$ as an I/O table. (2) Encode the dependency of the property outcome on the I/O behaviors with an HP model (Section IV). (3) Search for a counterfactual model value assignment, revealing an actual cause and a repair (Section V). (4) Decode the found assignment as a counterfactual component behavior. (5) Replace $\mathcal{C}$ with an alternative component that performs this counterfactual behavior to repair the system.

The entire workflow is implemented on an OpenAI Gym example in Section VI.

IV. Halpern-Pearl Model Design

We first describe a naive way of encoding the dependency of φ on component $\mathcal{C}$'s I/O behaviors in Section IV-A. This results in an HP model with an infinite number of nodes. To alleviate this issue, we describe a method for constructing an HP model with a finite number of nodes in Section IV-B.

To fulfill the minimality in AC3 of Definition 1, we will need to distinguish which counterfactuals are closer to the factual behavior. This requires a partial order on I/O behaviors that aligns with some partial order on the HP node values. Unfortunately, the naive finite HP model does not admit fine enough partial orders over its node values. Thus, we will refine our HP model to support suitable partial orders and make it amenable to the counterfactual search.

A. Infinite HP Model

Intuitively, a naive way to build the HP model to model the dependency of property outcome on I/Os of $\mathcal{C}$ is to model every input $x \in \mathcal{X}$ as an endogenous node, and its corresponding output $y \in \mathcal{Y}$ as its value. We illustrate this model in Figure 2 and refer to it as the *infinite HP model*.

This model encodes the component $\mathcal{C}$'s implemented I/O behaviors as an exogenous node (yellow), and the behaviors for each input $x \in \mathcal{X}$, as well as the property outcome as endogenous nodes. Since we assume the full knowledge of the $\mathcal{C}$'s I/Os, we can extract the input-output tuples by tabulating

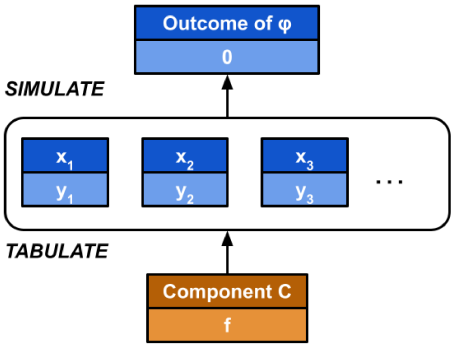

Fig. 2: Infinite HP model.

Algorithm 1: Discretize HP Model

Input : Factual behavior $f : \mathcal{X} \mapsto \mathcal{Y}$, continuous bounded spaces $\mathcal{X}$ and $\mathcal{Y}$, initial cell width Δ_x^{init}, Δ_y^{init}

Output: Partitioned cells $\mathcal{X}_1, \ldots, \mathcal{X}_m$, $\mathcal{Y}_1, \ldots, \mathcal{Y}_n$, a map $g : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ from input cells to output cells

1 $\Delta_x \leftarrow \Delta_x^{\text{init}}$;
2 $\Delta_y \leftarrow \Delta_y^{\text{init}}$;
3 **do**
4 Partition $\mathcal{Y}$ with cell width Δ_y;
5 $\Delta_y \leftarrow \Delta_y/2$;
6 **do**
7 Partition $\mathcal{X}$ with cell width Δ_x;
8 $\Delta_x \leftarrow \Delta_x/2$;
9 **while** $\exists \mathcal{X}_i$ *that cannot be completely mapped into some* $\mathcal{Y}_j$;
10 $g \leftarrow$ the current mapping between cells;
11 $f_r \leftarrow \text{RECON}(g)$;
12 **while** $\text{SIMULATE}(f_r) \neq \text{SIMULATE}(f)$;

them. Then, based on Assumption 1, the outcome can be obtained by calling SIMULATE on the I/Os. This design is illustrated in Figure 2. Each node has its variable name at the top and a value at the bottom.

Recall that the input space $\mathcal{X}$ is continuous. Therefore, an obvious drawback of this infinite HP model is that it has infinitely many nodes for each $x \in \mathcal{X}$, and thus an infinitely large search space for an actual cause given by Definition 1.

B. Discretized HP Model

To create a finite HP model, we partition the input space $\mathcal{X}$ and output space $\mathcal{Y}$ into sufficiently fine-grained, but finitely many cells $\mathcal{X} = \bigcup_{i=1}^{m} \mathcal{X}_i$ and $\mathcal{Y} = \bigcup_{j=1}^{n} \mathcal{Y}_j$, and then consider functions that map all $x \in \mathcal{X}_1$ to the center of one $\mathcal{Y}_j$. The partitioning procedure is detailed in Algorithm 1.

Algorithm 1 splits the input space and output space into hypercube cells. The cell size keeps shrinking until the two conditions, respectively in line 10 and line 13 are met. We

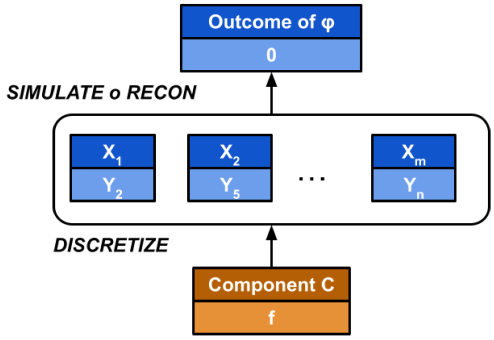

Fig. 3: Discretized HP model. We illustrate this with an example value assignment of nodes.

define the reconstruction method RECON at line 12 as follows.

$$\text{RECON}(g) = f_r, \text{ where } \forall x \in \mathcal{X}_i, f_r(x) = \text{center}(\mathcal{Y}_j), j = g(i)$$

That is, it reconstructs an I/O behavior from the discrete map g, which approximates the factual I/O behaviors as a mapping among cells, by mapping every x in an input cell to the center of the corresponding output cell. This algorithm is guaranteed to terminate by the following theorem.

Theorem 1 (Termination of Discretization)**.** *Algorithm 1 is guaranteed to terminate.*

For the proof, please refer to our full paper with appendix [33]. Proofs of later theorems can also be found there.

Upon the termination of Algorithm 1, we obtain a function $f_r = \text{RECON}(g)$ that represents f by (1) having arbitrarily close outputs on the same inputs and (2) having the same property outcome. Then, our causal analysis is established on the family of functions like f_r, i.e.,

Definition 3 (Representative Component Behavior Space)**.** *Given the cell partition $\mathcal{X} = \bigcup_i^m \mathcal{X}_i$ and $\mathcal{Y} = \bigcup_j^n \mathcal{Y}_j$, we define the* representative component behavior space *as a finite subset of $(\mathcal{X} \mapsto \mathcal{Y})$ as*

$$\mathcal{F}_r = \{f' \in (\mathcal{X} \mapsto \mathcal{Y}) \mid \forall i, \forall x \in \mathcal{X}_i, \exists j, f'(x) = \text{center}(\mathcal{Y}_j)\} \tag{4}$$

Notice that if we wrap up the generation of g from f in Algorithm 1 as a method DISCRETIZE, the two methods DISCRETIZE and RECON are inverse to each other if the behaviors are restricted within $\mathcal{F}_r$.

With this shrunken space of behavior choices, we modify our HP model to the next version, called the *discretized HP model*, as in Figure 3. Here, we have one endogenous node per input cell, and its value is the mapped output cell. This HP model has $m + 2$ nodes and can express every behavior choice in $\mathcal{F}_r$. Notice that in place of TABULATE, we now have DISCRETIZE since the nodes are now representing the discrete map g, and the SIMULATE method requires RECON on the discrete map first.

Generally, we want to pick a counterfactual for $\mathcal{C}$ that repairs the system $\mathcal{S}$, i.e., flipping the outcome from 0 to 1, that is closest to the factual. This is motivated by the minimality of actual causes in AC3 in Definition 1. Therefore, we first define a partial order on I/O behaviors.

Definition 4 (Partial Order on Behaviors)**.** *For a factual behavior choice $f : \mathcal{X} \mapsto \mathcal{Y}$, we define a* partial behavior order $\preccurlyeq_f$ *on $(\mathcal{X} \mapsto \mathcal{Y})$ as*

$$\begin{aligned} f_1 \preccurlyeq_f f_2 \iff \forall x \in \mathcal{X}, \forall j \in \{1, \ldots, \dim(\mathcal{Y})\}, \\ (f(x)[j] \leq f_1(x)[j] \leq f_2(x)[j]) \\ \vee (f(x)[j] \geq f_1(x)[j] \geq f_2(x)[j]) \end{aligned} \tag{5}$$

where $[j]$ denotes the j-th dimension of a vector.

In plain words, $f_1 \preccurlyeq_f f_2$ iff f_1 has closer outputs to the factual f than f_2 does on all dimensions and on all inputs.

However, one drawback of this discrete HP model and the partial order from Definition 4 is that we cannot tell the difference between two counterfactual I/O behaviors in terms of the sets of "disagreeing" nodes compared to the factual $\mathbf{v} = \text{DISCRETIZE}(f_r)$, which is something we need for reasoning about (AC3) in Definition 1. For example, in Figure 3, assume that two counterfactuals f_1 and f_2 change the mapping on $\mathcal{X}_1$ from the factual $\mathcal{Y}_2$ (as in f_r) to $\mathcal{Y}_3$ and $\mathcal{Y}_4$, respectively. With this mapping change, both flip the outcome to 1. In the discretized HP model, the number of "disagreeing" nodes under both assignments is 1, but $\mathcal{Y}_4$ may be further from the factual $\mathcal{Y}_2$ than $\mathcal{Y}_3$. In plain words, we cannot answer "which one is more different from f_r: f_1 or f_2?" by simply comparing the two sets of nodes they modify from the factual value assignment.

C. Propositional HP Model

We now present a partial order, $\preccurlyeq_{\mathbf{v}}$, over node value differences and define how $\preccurlyeq_{\mathbf{v}}$ relates to partial order $\preccurlyeq_f$ from Definition 4. We then present an HP model construction technique, starting from the discretized HP model, that preserves $\preccurlyeq_{\mathbf{v}}$ and $\preccurlyeq_f$.

For the new partial order, if the subset of nodes that differ between assignments $\mathbf{v}_2$ and factual $\mathbf{v}$ contains the subset of nodes that differ between $\mathbf{v_1}$ and $\mathbf{v}$, we say that $\mathbf{v}_1$ is closer to $\mathbf{v}$ than $\mathbf{v}_2$ does. We formulate this as another partial order, on value assignments of a subset of endogenous nodes $\mathcal{U}_{\text{io}} \subseteq \mathcal{U}_{\text{endo}}$. The value space of $\mathcal{U}_{\text{io}}$ is denoted as $\mathcal{V}_{\text{io}}$.

Definition 5 (Partial Order on HP Node Values)**.** *On a set of HP model nodes $\mathcal{U}$ and its value space $\mathcal{V}$, given the factual value assignment $\mathbf{v} \in \mathcal{V}$, we can define a partial order $\preccurlyeq_{\mathbf{v}}$ that*

$$\mathbf{v}_1 \preccurlyeq_{\mathbf{v}} \mathbf{v}_2 \iff \text{diff}_{\mathcal{U}}(\mathbf{v}, \mathbf{v}_1) \subseteq \text{diff}_{\mathcal{U}}(\mathbf{v}, \mathbf{v}_2), \tag{6}$$

where $\text{diff}_{\mathcal{U}}(\cdot, \cdot)$ denotes the subset of nodes in $\mathcal{U}$ that has different values by two assignments. Equality $=_{\mathbf{v}}$ holds iff $\text{diff}_{\mathcal{U}}(\mathbf{v}, \mathbf{v}_1) = \text{diff}_{\mathcal{U}}(\mathbf{v}, \mathbf{v}_2)$.

With this partial order, we can determine if an assignment $\mathbf{v}_1$ differs more from the factual $\mathbf{v}$ than assignment $\mathbf{v}_2$ does.

Consequently, a larger change from the factual I/O behavior needs to be reflected in a larger set of differing nodes, i.e., the

partial order $\preccurlyeq_{\mathbf{v}}$ needs to preserve the partial order $\preccurlyeq_f$. We define partial order preservation as follows:

Definition 6 (Partial Order Preservation)**.** *Consider a representative behavior space $\mathcal{F}_r \subset (\mathcal{X} \mapsto \mathcal{Y})$ with a factual representative component f_r and the induced partial behavior order $\preccurlyeq_{f_r}$. Consider also a subset of endogenous nodes, $\mathcal{U}_{io} \subseteq \mathcal{U}_{endo}$, with the value space of this subset $\mathcal{V}_{io}$. An encoding of a representative I/O behavior onto the nodes $\mathcal{U}_{io}$,* ENCODE $: \mathcal{F}_r \mapsto \mathcal{V}_{io}$, preserves partial order *iff* ENCODE$(f_r) = \mathbf{v}$, *and*

$$\forall f_1, f_2 \in \mathcal{F}_r, f_1 \preccurlyeq_{f_r} f_2 \iff \text{ENCODE}(f_1) \preccurlyeq_{\mathbf{v}} \text{ENCODE}(f_2) \tag{7}$$

Finally, we define a finer-grained HP model that preserves these two partial orders based on Definition 6.

First, we first define an output bin of $\mathcal{Y}$. As illustrated in Figure 4, if the output space is partitioned into 4×4 hypercube cells, we can index the cells from $\mathcal{Y}_{11}$ to $\mathcal{Y}_{44}$. We therefore have 4 bins along dimension 1 and 4 bins along dimension 2. Formally, if we index the $\dim(\mathcal{Y}) = d$-dimensional output cells as $\mathcal{Y}_{k_1 k_2 \ldots k_d}$, we have

Definition 7 (Output Bins)**.** *The k-th output bin along dimension j is $bin_{\mathcal{Y}}(j,k) = \bigcup\{\mathcal{Y}_{k_1 k_2 \ldots k_d} \mid k_j = k\}$*

We denote the total number of bins along the j-th dimension as n_j, and therefore $k_j \in \{1, \ldots, n_j\}$. Next, we can define the lower bound of $bin_{\mathcal{Y}}(j,k)$ in j-th dimension as $lo(bin_{\mathcal{Y}}(j,k))$. Notice that along a fixed dimension j, we have a total order of lower bounds of bins based on k.

We are now ready to construct the final, *propositional HP model*, created by Algorithm 2. In contrast to the discretized HP model, now the endogenous nodes that encode the I/O behaviors ($\mathcal{U}_{\text{io}}$ in the algorithm) take propositional values. Node u_{ijk} encodes whether for input $x \in \mathcal{X}_i$ the output cell for the j-th dimension is in at least the k-th output bin. In the example illustrated in Figure 5, we have 1-dimensional $\mathcal{X}$, with $\mathcal{X}_1 = [0,1]$ and $\mathcal{X}_2 = [1,2]$. We also have 2-dimensional $\mathcal{Y} = ([5,6] \cup [6,7]) \times ([10,11] \cup [11,12])$. The example value shows that the representative function f_r of f maps all $x \in [0,1]$ to the center of $[6,7] \times [10,11]$, and all $x \in [1,2]$ to $[5,6] \times [10,11]$. Under this propositional HP model, the encoding ENCODE $: \mathcal{V}_{\text{io}} \mapsto \mathcal{F}_r$ is simply evaluating the node propositions based on the component behavior, and DECODE is first identifying the discretized mapping between cells, and then calling RECON. Notice that ENCODE and DECODE are inverse to each other if we restrict the behavior choices in $\mathcal{F}_r$.

With m input cells, d output dimensions and n_j bins along a dimension j, this HP model has $m\sum_{j=1}^{d} n_j + 2$ nodes, with $|\mathcal{U}_{\text{io}}| = m\sum_{j=1}^{d} n_j$. However, the total number of value assignments on $\mathcal{U}_{\text{io}}$ is not $2^{|\mathcal{U}_{\text{io}}|}$, because some value assignments are not allowed. For example, we cannot let $f(x) \geq 5$ be false and $f(x) \geq 6$ be true. The total number of valid assignments is $(\prod_{j=1}^{d} n_j)^m = n^m$, the same as the discretized HP model.

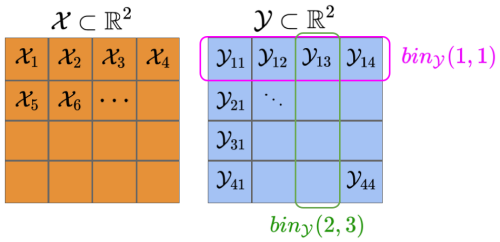

Fig. 4: An example indexing of cells and output bins when both $\mathcal{X}$ and $\mathcal{Y}$ are 2-dimensional.

Algorithm 2: Construct Propositional HP Model

Input : Input space $\mathcal{X}$ and output space $\mathcal{Y}$ of component behaviors and their indexed cells, simulator SIMULATE, encoder ENCODE, decoder DECODE

Output: HP model $\mathcal{M} = (\mathcal{U}_{\text{endo}}, \mathcal{U}_{\text{exo}}, \mathcal{V}, \mathcal{E})$

$\mathcal{U}_{\text{exo}} \leftarrow \{u_{\text{comp}}\}$;
$\mathcal{U}_{\text{endo}} \leftarrow \{u_{\varphi}\}$;
$\mathcal{V}(u_{\text{comp}}) \leftarrow (\mathcal{X} \mapsto \mathcal{Y})$;
$\mathcal{U}_{\text{io}} \leftarrow \emptyset$;
$\mathcal{E} \leftarrow \{\text{SIMULATE} \circ \text{DECODE}, \text{ENCODE}\}$;
for $i = 1, \ldots, m$ **do**
  for $j = 1, \ldots, d$ **do**
    for $k = 1, \ldots, n_j$ **do**
      $u_{ijk} \leftarrow (x \in \mathcal{X}_i \implies f(x)[k] \geq lo(bin_{\mathcal{Y}}(j,k))$;
      $\mathcal{V}(u_{ijk}) \leftarrow \{0,1\}$;
      $\mathcal{U}_{\text{io}} \leftarrow \mathcal{U}_{\text{io}} \cup \{u_{ijk}\}$;
    end
  end
end
$\mathcal{U}_{\text{endo}} \leftarrow \mathcal{U}_{\text{io}} \cup \mathcal{U}_{\text{endo}}$;

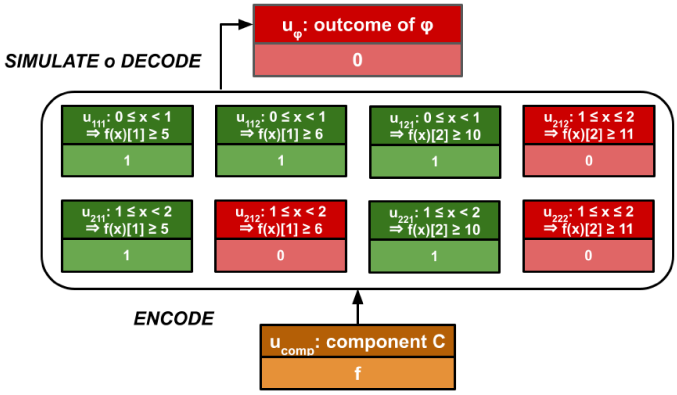

Fig. 5: An example of the propositional HP model. We use red and green colors to show the nodes with 0 and 1 values.

Theorem 2 (Propositional Encoding Preserves Partial Order)**.** *The encoding* ENCODE : $\mathcal{F}_r \mapsto \mathcal{V}_{io}$ *specified by evaluating the proposition of every* $u_{ijk} \in \mathcal{U}_{io}$*, as constructed in Algorithm 2, preserves the partial order as defined in Definition 6.*

V. The Causal Repair Algorithm

A. Satisfactory Counterfactual Search

Based on the constructed propositional HP model, we look for an actual cause. The idea is to first search for a counterfactual $\mathbf{v}'_{\text{io}}$ on $\mathcal{U}_{\text{io}}$ that leads to a satisfactory outcome, i.e., $\mathcal{S}(f'_r) \models \varphi$ and $f'_r = \text{DECODE}(\mathbf{v}'_{\text{io}})$. The subset of nodes with different value assignments between the factual $\mathbf{v}_{\text{io}}$ and counterfactual $\mathbf{v}'_{\text{io}}$ is not necessarily an actual cause yet (it may not be minimal as per condition AC3), and in the next subsection, we will look for a different $\mathbf{v}^*_{\text{io}}$ that fulfills the actual cause conditions from this $\mathbf{v}'_{\text{io}}$. Here, we focus on the search for $\mathbf{v}'_{\text{io}}$.

Since a brute-force search on all n^m possible value assignments takes exponential time, we leverage random sampling as follows in Algorithm 3.

Algorithm 3: Counterfactual Random Sampling

Input : HP model $\mathcal{M}$, probability threshold p, significance level $\alpha \in [0,1]$, simulator SIMULATE, decoder DECODE, probability distributions $\mathcal{D}_1$, $\mathcal{D}_2$

Output: Either a counterfactual value assignment $\mathbf{v}'_{\text{io}}$ on $\mathcal{U}_{\text{io}}$ with $\text{SIMULATE}(\text{DECODE}(\mathbf{v}'_{\text{io}})) = 1$, or a statement $\Pr_{\mathcal{D}_2}[\Pr_{\mathcal{D}_1}[\text{SIMULATE}(\text{DECODE}(\mathbf{v}'_{\text{io}})) = 1] \leq p] \geq 1 - \alpha$

1 $N \leftarrow \lceil (1/p - 1)Q(1-\alpha/2)^2 \rceil$;
2 **for** $1 \ldots N$ **do**
3 $\mathbf{v}'_{\text{io}} \leftarrow$ uniform sampling from all n^m settings;
4 $\varphi \leftarrow \text{SIMULATE}(\text{DECODE}(\mathbf{v}'_{\text{io}}))$;
5 **if** φ **then**
6 Return $\mathbf{v}'_{\text{io}}$;
7 **end**
8 **end**
9 Return $\Pr_{\mathcal{D}_2}[\Pr_{\mathcal{D}_1}[\text{SIMULATE}(\text{DECODE}(\mathbf{v}'_{\text{io}})) = 1] \leq p] \geq 1 - \alpha$;

The distribution $\mathcal{D}_1$ is a distribution on different value assignments in the value space $\mathcal{V}_{\text{io}}$. For example, it can be a uniform distribution on the propositional node values. Distribution $\mathcal{D}_2$ is the induced distribution on a sampled value assignment's success rate, i.e., $p = \Pr_{\mathcal{D}_1}[\text{SIMULATE}(\text{DECODE}(\mathbf{v}'_{\text{io}}) = 1]$, which is treated as another random variable. In line 1, the function $Q(\cdot)$ means the quantile on standard normal distribution. Starting from line 2 to 8, we uniformly sample different value assignments $\mathbf{v}'_{\text{io}}$ until we find one $\mathbf{v}'_{\text{io}}$ that produces satisfactory φ by SIMULATE or we reach a maximum number of samples. If we fail to find a satisfactory value assignment, we report that the probability of finding a satisfactory $\mathbf{v}'_{\text{io}}$ with uniform sampling, i.e., the portion of satisfactory $\mathbf{v}'_{\text{io}}$ in the entire search space, is at most p with confidence $1 - \alpha$ at line 9. We show our probabilistic failure statement at line 9 holds with the following theorem.

Theorem 3 (Probabilistic Guarantee on Search Failure)**.** *Given an HP model constructed by Algorithm 2, a probability threshold* $p \in [0,1]$ *and a confidence* $1 - \alpha \in [0,1]$*, the final statement at line 9 of Algorithm 3 holds.*

Upon N consecutive failures, this search algorithm states that with some confidence a counterfactual assignment that flips the outcome is unlikely to be found. This suggests that the actual cause for the run-time property violation lies elsewhere: possibly the environment, the behaviors of another component, or the suspected component together with another component — but not the I/O behaviors of this component alone. This confidence is based on a substantial number of samples without finding a successful counterfactual. For example, for $p = 0.001$ and $\alpha = 0.05$, one would need to uniformly sample at least $N = 3838$ failed counterfactuals in a row.

B. Node Value Interpolation for Actual Cause

Suppose we have successfully obtained a satisfactory $\mathbf{v}'_{\text{io}}$ on $\mathcal{U}_{\text{io}}$, from Algorithm 3. The final step is to find a counterfactual $\mathbf{v}^*_{\text{io}}$ that can flip the outcome and is as close to the factual $\mathbf{v}_{\text{io}}$ as possible, starting from $\mathbf{v}'_{\text{io}}$. This step is required to satisfy the minimiality condition (AC3) of Definition 1.

We therefore perform a deterministic interpolation between $\mathbf{v}'_{\text{io}}$ and $\mathbf{v}_{\text{io}}$ in Algorithm 4, which starts from the found satisfactory counterfactual assignment $\mathbf{v}'_{\text{io}}$. In this algorithm, the counterfactual value assignment steps towards the factual $\mathbf{v}_{\text{io}}$ by flipping the differing value assignments one-by-one. This procedure continues until there are no nodes it can flip while still satisfying φ. Because the total number of nodes in $\mathcal{U}_{\text{io}}$ is $m\sum_{j=1}^{d} n_j$, Algorithm 4 is guaranteed to output an actual cause with a satisfactory counterfactual within $O(m\sum_{j=1}^{d} n_j)$ time. The complexity is linear in terms of m and n, allowing for efficient computation.

We visualize the process of searching for $\mathbf{v}^*_{\text{io}}$ as Figure 6. Without loss of generality, consider only the top left input cell. The factual behavior $f = \text{DECODE}(\mathbf{v}_{\text{io}})$ maps inputs in this cell to some output cell and this behavior produces a property violation. The sampling by Algorithm 3 finds a satisfactory counterfactual $f' = \text{DECODE}(\mathbf{v}'_{\text{io}})$, which is an alternative mapping. Next, the interpolation Algorithm 4 flips the nodes disagreed by $\mathbf{v}_{\text{io}}$ and $\mathbf{v}'_{\text{io}}$ one-by-one towards $\mathbf{v}_{\text{io}}$. This flipping is equivalent to stepping through the output cells one by one. Eventually, the algorithm reaches a node assignment where any steps towards f result in φ becoming violated, at which point it returns the current node assignment. Depending on the dimensions, we can take different paths in stepping, i.e. we can end up either in cell (1) or cell (2), from interpolating in the vertical or horizontal dimension first, respectively. Mapping the input cell to either of these two cells represents a valid I/O behavior of f^*.

Notice that Algorithm 4 incrementally steps towards the factual $\mathbf{v}_{\text{io}}$ from $\mathbf{v}'_{\text{io}}$ by flipping nodes one-by-one. A more

Algorithm 4: Actual Cause Search by Incremental Interpolation

Input : HP model $\mathcal{M}$, factual $\mathbf{v}_{\text{io}}$, satisfactory counterfactual $\mathbf{v}'_{\text{io}}$, simulate function SIMULATE

Output: Satisfactory counterfactual $\mathbf{v}^*_{\text{io}}$ such that its difference from $\mathbf{v}_{\text{io}}$ is an actual cause as per $\mathcal{M}$

$\mathbf{v}^*_{\text{io}} \leftarrow \mathbf{v}'_{\text{io}}$;
$\text{diff}^*_{\mathcal{U}_{\text{io}}} \leftarrow \text{diff}_{\mathcal{U}_{\text{io}}}(\mathbf{v}_{\text{io}}, \mathbf{v}'_{\text{io}})$;
for $i = 1, \ldots, m$ **do**
  for $j = 1, \ldots, d$ **do**
    for $k = 1, \ldots, n_d$ **do**
      if $u_{ijk} \notin \text{diff}^*_{\mathcal{U}_{io}}$ **then**
        Continue;
      end
      $\mathbf{v}_{\text{temp}} \leftarrow \mathbf{v}^*_{\text{io}}$ with value assignment on u_{ijk} the negation as in $\mathbf{v}^*_{\text{io}}$;
      if SIMULATE(DECODE($\mathbf{v}_{temp}$)) $= 1$ **then**
        $\mathbf{v}^*_{\text{io}} \leftarrow \mathbf{v}_{\text{temp}}$;
        $\text{diff}^*_{\mathcal{U}_{\text{io}}} \leftarrow \text{diff}^*_{\mathcal{U}_{\text{io}}} \setminus \{u_{ijk}\}$
      end
    end
  end
end

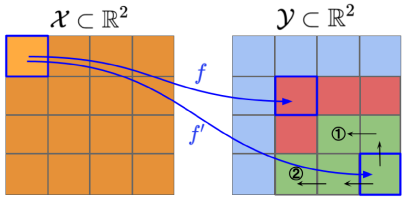

Fig. 6: An example of finding actual cause by interpolation, with two-dimensional $\mathcal{X}$ and $\mathcal{Y}$. Red cells in $\mathcal{Y}$ denote property violation, green denotes satisfaction, and blue denotes cells where satisfaction is not relevant to the example.

efficient variant is to do a *binary search* between these two value assignments. We denote these two approaches as incremental interpolation and binary search interpolation, respectively. Both are evaluated in Section VI.

Next, we show that the output is indeed an actual cause based on HP model $\mathcal{M}$ in Theorem 4.

Theorem 4 (Output is Actual Cause)**.** *Let HP model $\mathcal{M}$ constructed in Algorithm 2 be given with factual node value assignment $\mathbf{v}_{io}$. Let $\mathbf{v}'_{io}$ be a counterfactual node value assignment from Algorithm 3. The node values on $\mathcal{U}_{io}$ where assignments $\mathbf{v}_{io}$ and $\mathbf{v}^*_{io}$ disagree in Algorithm 4 are an* actual cause *of $\mathcal{S}(f) \not\models \varphi$ as per $\mathcal{M}$ constructed in Algorithm 2.*

VI. Experimental Evaluation

A. Setup

We test our diagnosis on the mountain car from the OpenAI Gym [13], which has become a common benchmark for learning-enabled CPS [34], [35]. The car starts in the valley between two mountains with the task is to drive it to the top of the right mountain within a deadline. The system has two one-dimensional state variables: position and velocity, and a one-dimensional control signal, as follows.

$$\begin{aligned} pos(t+1) &= pos(t) + vel(t) \\ vel(t+1) &= vel(t) + 0.0015 ctrl(t) - z\cos(3pos(t)) \\ ctrl(t) &= f(pos(t), vel(t)) \end{aligned} \tag{8}$$

where $z = 0.0025$ is the steepness of the hill. The variables are bounded, such that $pos(t) \in [-1.2, 0.6]$, $vel(t) \in [-0.07, 0.07]$ and $ctrl(t) \in [-1, 1]$. The initial condition is $(pos(0), vel(0)) = (-0.5, 0)$, i.e., staying still at the bottom of the valley. The controller is a learning-based function f, for which we use a pre-trained deep neural network. The run-time property can be specified as an STL [32] formula

$$\varphi = F_{t \le 110}(pos(t) \ge 0.45), \tag{9}$$

i.e., reaching $pos(t) \ge 0.45$ before the deadline of $t = 110$. Due to its limited power, the car cannot reach its goal directly, and the challenge is to first climb the left mountain to gain enough momentum.

As the controller function f, we use a pre-trained rectangular deep neural network with shape 8×16 with sigmoid activations. The run-time property φ is violated on initial state $(pos(0), vel(0)) = (-0.5, 0)$, and we suspect it is caused by this learned controller. Therefore, we run our diagnosis step-by-step to find an actual cause and observe its effect on system repair. All experiments are run on a single core of Intel Xeon Gold 6148 CPU @ 2.40GHz.

B. Results

Based on the setup, the input space and output space of the suspected controller are $\mathcal{X} = [-0.6, 1.2] \times [-0.07, 0.07]$ and $\mathcal{Y} = [-1, 1]$. We first run Algorithm 1 and find that the cell width is of size 0.1×0.01 on $\mathcal{X}$ and 0.1 on $\mathcal{Y}$. In other words, position is split into $(1.2 - (-0.6))/0.1 = 18$ equally sized intervals, velocity into $(0.07 - (-0.07))/0.01 = 14$ intervals and control into $(1 - (-1))/0.1 = 20$ intervals. Therefore, there are $18 \times 14 = 252$ input cells and 20 output cells.

By using Algorithm 2, we constructed an HP model, with every $u_{ik} \in \mathcal{U}_{\text{io}}$ representing a Boolean statement "on this input cell $\mathcal{X}_i$ of (position, velocity), the output control is at least in output bin $bin_{\mathcal{Y}}(1, k)$". We do not have j here because the output space is one-dimensional. Then, using Algorithm 3, we found three counterfactuals $\mathbf{v}^1_{\text{io}}$, $\mathbf{v}^2_{\text{io}}$, and $\mathbf{v}^3_{\text{io}}$ that lead to the satisfaction of φ. We then applied Algorithm 4 to find the minimal modification needed from factual $\mathbf{v}_{\text{io}}$ to these 3 counterfactuals, with the interpolated counterfactuals denoted as $\mathbf{v}^{1*}_{\text{io}}$, $\mathbf{v}^{2*}_{\text{io}}$ and $\mathbf{v}^{3*}_{\text{io}}$, respectively.

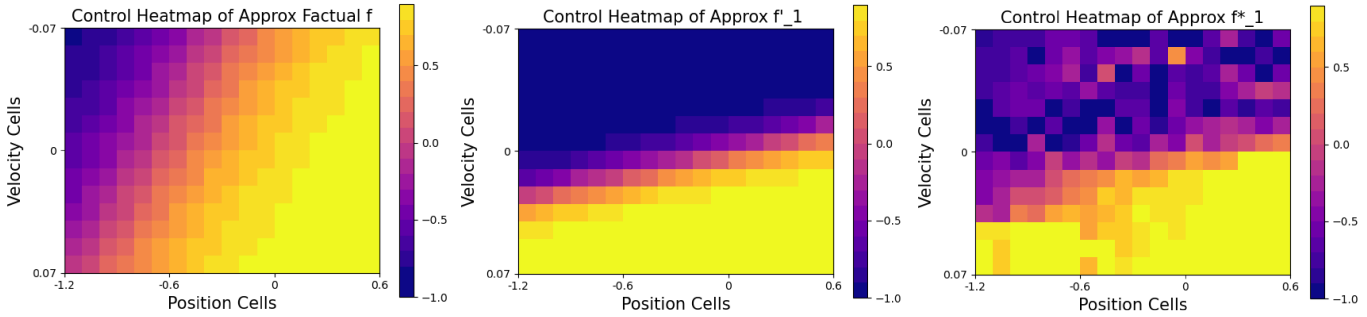

Fig. 7: Control functions, approximated by mapping between cells, visualized as heatmaps on the position-velocity space. The functions from left to right are the factual f, the searched counterfactual f_1' and interpolated counterfactual f_1^* between the first two.

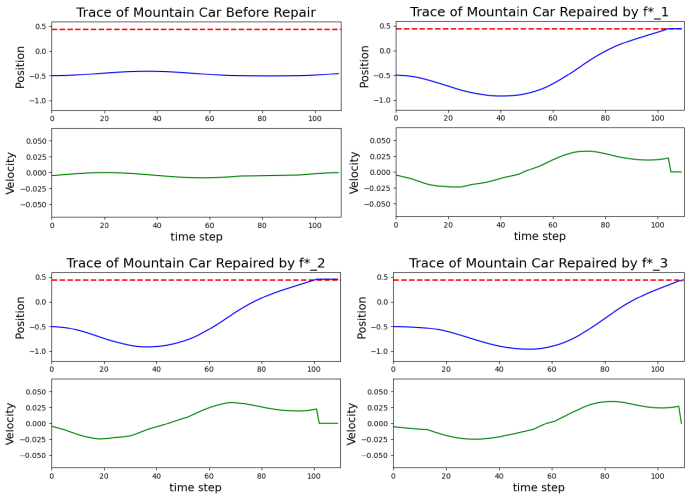

Fig. 8: Mountain car traces: one before repair and three causally repaired. The red dashed line means $pos = 0.45$, i.e., the target. The car stops immediately after target is reached.

Figure 7 shows the control functions f, $f_1' = \text{DECODE}(\mathbf{v}_{\text{io}}^1)$, and $f_1^* = \text{DECODE}(\mathbf{v}_{\text{io}}^{1*})$ as heatmaps. There are 153 (out of 252) input cells that map to a different output cell between f and f_1^*. Consequently, our diagnosis pipeline concludes that the actual cause is *"the control signals on these 153 input cells given by the factual controller f"*, with a corresponding repair *"had these 153 cells been mapped by f_1^* instead of f, the system would have satisfied φ"*.

After we found a counterfactual behavior, we used both the incremental and binary search interpolations. The computation time is listed in Table I. We can see the time overhead is predominantly from running the simulator, and that interpolation with binary search does reduce this overhead.

To validate that these modifications indeed repair the system, we replace the controller with interpolated control functions f_1^*, f_2^* and f_3^*. The re-runs give satisfactory results as shown in Figure 8, showing the utility of our causal diagnosis.

VII. Discussion and Conclusion

Our diagnosis and repair pipeline has several advantages. First, we are able to leverage domain knowledge of the system, such as how to simulate the outcome of the given STL property, and well-defined actual causality models to reason about the causes. Second, the actual cause we identify reflects a minimal subset of the component's I/O behaviors that cause the problem under our defined partial order. Therefore, we avoid changing the complex and possibly unavailable internal structure of components and blaming unrelated, "innocent" behaviors. Finally, as the experiments demonstrate, our approach produces practically useful repair suggestions.

Limitations do exist in our current pipeline, so future research is required. First, the HP model construction and component behavior encoding in Section IV-C is only one way of preserving the partial order on behaviors. There may be more effective ways with fewer HP model nodes and therefore smaller memory and time overhead. Second, the search algorithm in Section V-A draws uniformly random samples from an exponentially large search space, and even if we are confident to report the satisfactory portion is relatively small and hence the search fails, that portion may still be large in an absolute sense. For example, a fraction of 10^{-6} on a space size of 10^{20} is still very large. Therefore, there is potential for making the search algorithm smarter, such as leveraging prior knowledge of the STL property. Moreover, we perform a repair on only one initial condition, which may violate the property on the others.

To summarize, this paper uses a novel construct of the Halpern-Pearl model to search for an actual cause for a runtime CPS property violation and provide a repair, without changing complex internal structures of LECs. Our diagnostic pipeline consists of two steps: (1) construct the propositional HP model and (2) search for different variable assignments on the model until we reach an actual cause. Our causal pipeline provides repair guarantees and can generate useful fixes to deep neural network controllers.

Interpolation	Total time (s)	Simulator time (s)	Stepping time (s)	# of operations
Incremental	9148.77	9148.76	0.003	1231
Binary	4965.03	4965.02	0.001	880

TABLE I: Computation time of interpolation from counterfactual f_1' to factual f. An "operation" is a combination of executing SIMULATE and stepping the counterfactual towards the factual behavior.

ACKNOWLEDGEMENT

This work was supported in part by ARO W911NF-20-1-0080 and AFRL and DARPA FA8750-18-C-0090. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Air Force Research Laboratory (AFRL), the Army Research Office (ARO), the Defense Advanced Research Projects Agency (DARPA), or the Department of Defense, or the United States Government.

REFERENCES

[1] S. Wang, A. Ayoub, B. Kim, G. Gössler, O. Sokolsky, and I. Lee, "A causality analysis framework for component-based real-time systems," in *International Conference on Runtime Verification*. Springer, 2013, pp. 285–303.

[2] J. Schumann, P. Moosbrugger, and K. Y. Rozier, "R2u2: monitoring and diagnosis of security threats for unmanned aerial systems," in *Runtime Verification*. Springer, 2015, pp. 233–249.

[3] J. Y. Halpern and J. Pearl, "Causes and explanations: A structural-model approach. part i: Causes," *The British journal for the philosophy of science*, 2005.

[4] J. Y. Halpern, *Actual causality*. MiT Press, 2016.

[5] H. Chockler and J. Y. Halpern, "Responsibility and blame: A structural-model approach," *Journal of Artificial Intelligence Research*, vol. 22, pp. 93–115, 2004.

[6] D. Held, S. Thrun, and S. Savarese, "Learning to track at 100 fps with deep regression networks," in *European conference on computer vision*. Springer, 2016, pp. 749–765.

[7] C. Hartsell, N. Mahadevan, S. Ramakrishna, A. Dubey, T. Bapty, T. Johnson, X. Koutsoukos, J. Sztipanovits, and G. Karsai, "Model-based design for cps with learning-enabled components," in *Proceedings of the Workshop on Design Automation for CPS and IoT*, 2019, pp. 1–9.

[8] C. E. Tuncali, J. Kapinski, H. Ito, and J. V. Deshmukh, "Reasoning about safety of learning-enabled components in autonomous cyber-physical systems," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[9] T. Chakraborti, S. Sreedharan, Y. Zhang, and S. Kambhampati, "Plan explanations as model reconciliation: Moving beyond explanation as soliloquy," *arXiv preprint arXiv:1701.08317*, 2017.

[10] B. Krarup, M. Cashmore, D. Magazzeni, and T. Miller, "Model-based contrastive explanations for explainable planning," 2019.

[11] V. Raman and H. Kress-Gazit, "Explaining impossible high-level robot behaviors," *IEEE Transactions on Robotics*, vol. 29, no. 1, 2012.

[12] "Towards minimal explanations of unsynthesizability for high-level robot behaviors," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 757–762.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[14] X. Qi, Y. Luo, G. Wu, K. Boriboonsomsin, and M. Barth, "Deep reinforcement learning enabled self-learning control for energy efficient driving," *Transportation Research Part C: Emerging Technologies*, vol. 99, pp. 67–81, 2019.

[15] D. Cao, J. Zhao, W. Hu, N. Yu, F. Ding, Q. Huang, and Z. Chen, "Deep reinforcement learning enabled physical-model-free two-timescale voltage control method for active distribution systems," *IEEE Transactions on Smart Grid*, vol. 13, no. 1, pp. 149–165, 2021.

[16] R. McConnell and R. Rajsuman, "Test and repair of large embedded drams. i," in *Proceedings International Test Conference 2001 (Cat. No. 01CH37260)*. IEEE, 2001, pp. 163–172.

[17] Y. Zorian, "Embedded memory test and repair: Infrastructure ip for soc yield," in *Proceedings. International Test Conference*. IEEE, 2002, pp. 340–349.

[18] D. Cazes and M. Kalech, "Model-based diagnosis with uncertain observations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 03, 2020, pp. 2766–2773.

[19] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 317–328, 2013.

[20] U. S. Cruz, J. Ferlez, and Y. Shoukry, "Safe-by-repair: a convex optimization approach for repairing unsafe two-level lattice neural network controllers," *arXiv preprint arXiv:2104.02788*, 2021.

[21] R. Peddi and N. Bezzo, "Interpretable run-time prediction and planning in co-robotic environments," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 2504–2510.

[22] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1135–1146.

[23] K. Majd, S. Zhou, H. B. Amor, G. Fainekos, and S. Sankaranarayanan, "Local repair of neural networks using optimization," *arXiv preprint arXiv:2109.14041*, 2021.

[24] J. Sohn, S. Kang, and S. Yoo, "Search based repair of deep neural networks," *arXiv preprint arXiv:1912.12463*, 2019.

[25] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu, "Nn repair: constraint-based repair of neural network classifiers," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 2021, pp. 3–25.

[26] X. Lin, H. Zhu, R. Samanta, and S. Jagannathan, "Art: abstraction refinement-guided training for provably correct neural networks," in *# PLACEHOLDER_PARENT_METADATA_VALUE#*, vol. 1. TU Wien Academic Press, 2020, pp. 148–157.

[27] D. Cohen and O. Strichman, "Automated repair of neural networks," *arXiv preprint arXiv:2207.08157*, 2022.

[28] F. Fu, Z. Wang, J. Fan, Y. Wang, C. Huang, X. Chen, Q. Zhu, and W. Li, "Reglo: Provable neural network repair for global robustness properties," in *Workshop on Trustworthy and Socially Responsible Machine Learning, NeurIPS 2022*.

[29] A. Ibrahim, S. Rehwald, and A. Pretschner, "Efficient checking of actual causality with sat solving," *Engineering Secure and Dependable Software Systems*, vol. 53, p. 241, 2019.

[30] A. Ibrahim, S. Kacianka, A. Pretschner, C. Hartsell, and G. Karsai, "Practical causal models for cyber-physical systems," in *NASA Formal Methods Symposium*. Springer, 2019, pp. 211–227.

[31] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[32] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 2004, pp. 152–166.

[33] P. Lu, I. Ruchkin, M. Cleaveland, O. Sokolsky, and I. Lee, "Causal repair of learning-enabled cyber-physical systems," *arXiv preprint arXiv:2304.02813*, 2023.

[34] R. Ivanov, T. Carpenter, J. Weimer, R. Alur, G. Pappas, and I. Lee, "Verisig 2.0: Verification of Neural Network Controllers Using Taylor Model Preconditioning," in *Computer Aided Verification*. Cham: Springer International Publishing, 2021, pp. 249–262.

[35] I. Ruchkin, M. Cleaveland, R. Ivanov, P. Lu, T. Carpenter, O. Sokolsky, and I. Lee, "Confidence composition for monitors of verification assumptions," in *2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS)*, 2022, pp. 1–12.

[36] R. G. Newcombe, "Interval estimation for the difference between independent proportions: comparison of eleven methods," *Statistics in medicine*, vol. 17, no. 8, pp. 873–890, 1998.