Safety Debugging of Tree-Ensemble Action Policies in AI Planning: From Fault Detection to Fault Fixing

Lorenzo Cascioli[*1], **Chaahat Jain**[*2], **Marcel Steinmetz**[3], **Jesse Davis**[1], **Jörg Hoffmann**[2,4]

[1]Department of Computer Science, KU Leuven, Leuven, Belgium
[2]Saarland University, Saarland Informatics Campus, Germany
[3]LAAS-CNRS, Toulouse, France
[4]German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
{jain, hoffmann}@cs.uni-saarland.de, {lorenzo.cascioli, jesse.davis}@kuleuven.be, marcel.steinmetz@laas.fr

Abstract

We address learned action policies π in discrete planning under initial-state and action-outcome non-determinism, with the objective to reach the goal while avoiding unsafe states. Recent work has shown how to detect *faults* in such π: action decisions $(s, \pi(s))$ one of whose possible outcomes leads from the safe region, where it is possible to be safe with certainty, to the unsafe region where that is not possible. Here we introduce methods to actually fix such faults. We address policies π represented as tree ensembles. Given a set of faults $\{(s_i, \pi(s_i))\}$, we revise π into a minimally different π' that guarantees to take different decisions on the states s_i. Iterating fault-detection with fault-fixing steps, we obtain a policy debugging loop that removes unsafe decisions while modifying the original learned policy as little as possible. In our experiments, consistently across a range of benchmarks, this debugging loop makes the policy safer without paying a price in goal-reaching performance. In some cases, an off-the-shelf method is able to verify the final policy to be safe.

1 Introduction

Learned action policies are gaining traction in AI (e.g., Mnih et al. 2015; Silver et al. 2016, 2018), including in AI Planning (e.g., Issakkimuthu, Fern, and Tadepalli 2018; Groshev et al. 2018; Garg, Bajpai et al. 2019; Toyer et al. 2020; Karia and Srivastava 2021; Ståhlberg, Bonet, and Geffner 2022a,b; Rossetti et al. 2024). However, such policies come without any built-in guarantees. In particular, safety is a major concern. Much work is being done on assessing policy safety in a post-hoc manner: a verification algorithm is applied to a previously learned policy in order to ascertain safety with respect to a symbolic environment model (e.g., Dutta, Chen, and Sankaranarayanan 2019; Akintunde et al. 2019; Ivanov et al. 2021; Bacci and Parker 2022; Vinzent, Sharma, and Hoffmann 2023; Amir et al. 2023; Wang et al. 2024). But this leaves the question: What if the outcome of verification is that the policy is unsafe?

A sizable body of work has addressed this question through **ML model repair**, in the simpler setting of single-step safety where the undesired behavior pertains to individual input/output pairs (Goldberger et al. 2020; Dong et al. 2021; Tan, Zhu, and Guo 2021; Sotoudeh and Thakur 2021; Bauer-Marquart et al. 2022; Boetius, Leue, and Sutter 2023; Zavatteri, Bresolin, and Navarin 2024). ML model repair and verification then together form a **debugging loop**. Here we establish the first such debugging loop for multi-step safety of ML models representing action policies. Figure 1 gives an overview of our loop.

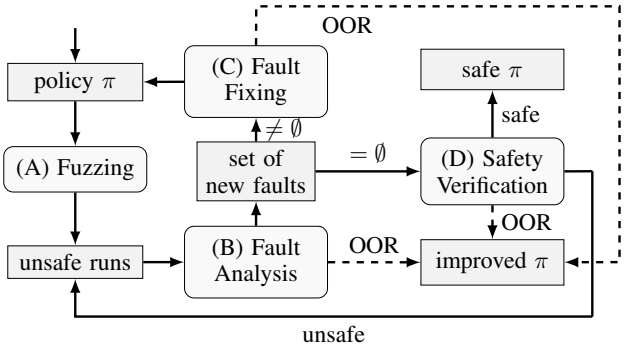


Figure 1: Overview of our safety debugging loop. OOR: Out of resources (timeout/out of memory).

We design and implement this loop in the context of planning for large state spaces with initial-state and action-outcome non-determinism. We address both discrete and continuous state spaces, with finite action choice. We consider policies π whose objective is to reach a state that satisfies a **goal condition** ϕ_*, subject to the hard constraint of never entering a state that satisfies a **failure condition** ϕ_F.

The major components of our debugging loop are: (A) **fuzzing**, which identifies unsafe policy runs, i.e., runs that end in a state satisfying ϕ_F; (B) **fault analysis**, which pinpoints the action decisions – so-called faults, see below – causing unsafety on such runs; (C) **fault fixing**, our repair step, which modifies the policy in a way that guarantees to fix a given set of faults; and (D) **safety verification**, which checks whether or not there exists an unsafe policy run.

With respect to prior work on ML model repair, components (A) and (B) are new. (A) is a much more practical way to find unsafe behavior, as opposed to running a full verifi-

[*]These authors contributed equally.

cation mechanism. (B) is necessary to determine the actual undesired input/output pairs on an unsafe policy run, where many action decisions may be perfectly safe (e.g. the policy may stop at a red light 17 times before deciding to drive too fast). Following Jain et al. (2025), we identify the action decisions $(s, \pi(s))$ causing unsafety on a policy run as **faults**: s itself is safe but one of the possible outcome states s' of applying the action $\pi(s)$ is unsafe (e.g. $\pi(s)$ may decide to drive too fast). Methods for both (A) and (B) in our context were recently established (Jain et al. 2025), and we use these off-the-shelf here. We also use (D) off-the-shelf, experimenting with two recent safety verification methods for tree-ensemble policies (Jain et al. 2024; Anonymous 2025).

Our key contributions are new methods for (C), as well as realizing and evaluating the overall debugging loop. While prior work on ML model repair exclusively addresses neural networks, here we address policies represented as tree ensembles instead. There are two motivations for this. First, recent work has shown that, in our planning context, tree ensembles can learn polices on par with neural ones (Jain et al. 2024). Second, as we show, tree ensembles offer compelling ways to fix undesired input/output pairs. Given a tree-ensemble policy π and a set of faults $\{(s_i, \pi(s_i))\}$, we repair π by finding new leaf values that differ minimally from the previous ones, subject to the constraint that different decisions are taken on the states s_i. This repair step *guarantees* to fix the undesired input/output pairs. It is feasible because we merely need to touch the leaves activated by s_i, i.e., one leaf per tree. Note the contrast to neural networks where any weight may influence the prediction for a given w_i. Our repair step affects other states that activate some of the same leaves. By virtue of that, it may generalize to other similar faults, which indeed our experiments confirm.

We run experiments on the benchmarks established for fault detection by (Jain et al. 2025), which contains instances of 13 domains; 10 of these domains have discrete state spaces, the other 3 have continuous state spaces. Across all these benchmarks, with very few exceptions, we observe:

- Our debugging loop substantially reduces the fraction of initial states in which the policy is unsafe.
- We keep or even increase the fraction of initial states from which the policy reaches the goal.
- Our method outclasses a baseline that does not generalize over the given sets of faults.

The verification methods often time out, but in some cases are able to verify the final policy to be safe.

2 Preliminaries

We consider fully-observable non-deterministic planning (e.g., Cimatti et al. 2003), over discrete or continuous state variables with finite action choice. Specifically, a **state space** is a tuple $\Theta = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{S}_0, \mathcal{S}_* \rangle$. Here, $\mathcal{S}$ is a (possibly infinite) set of **states**. We assume that states are value assignments to a given finite set of **state variables** $\mathcal{V}$, which may be Boolean, integer, or real-valued. $\mathcal{A}$ is a finite set of **action labels**. $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}} \setminus \{\emptyset\}$ is the **transition function**, that maps a given state s and action a to the set of possible successor states s'. $\mathcal{S}_0 \subseteq \mathcal{S}$ is the non-empty set of **initial states** and $\mathcal{S}_* \subseteq \mathcal{S}$ is the set of **goal states**. We denote by $app(s) \subseteq \mathcal{A}$ the set of actions **applicable** in state s, i.e., those for which $\mathcal{T}(s, a)$ is non-empty. We assume w.l.o.g. that $app(s) \neq \emptyset$ for all $s \in \mathcal{S}$. If $s' \in \mathcal{T}(s, a)$ is a possible successor of s under action a, we write $s \to_a s'$.

The fault fixing methods we present here do not require a declarative model of Θ. We merely assume that we have access to $\mathcal{V}$, and that $\mathcal{S}_0$ and $\mathcal{S}_*$ are specified in terms of given formulas over ϕ_0 and ϕ_* over $\mathcal{V}$. That said, in safety debugging as per Figure 1, a declarative model is required for (D) safety verification. Our implementation builds on the prior work by (Jain et al. 2024, 2025), which uses the JANI language (Budde et al. 2017) to model state spaces.

We assume a given **failure condition** ϕ_F in the form of a formula over $\mathcal{V}$. By $\mathcal{S}_F \subset \mathcal{S}$ we denote the set of **failed states** where ϕ_F is true.

A **policy** is a function $\pi : \mathcal{S} \mapsto \mathcal{A}$ such that $\pi(s) \in app(s)$ for all states $s \in \mathcal{S}$. A **run** of π from a state $s_0 \in \mathcal{S}_0$ is an alternating sequence of states and actions $\sigma^{\pi}_{s_0} = s_0, \pi(s_0), s_1, \dots$ such that, for all $i \geq 0$, $s_{i+1} \in \mathcal{T}(s_i, \pi(s_i))$ and if $s_i \in \mathcal{S}_* \cup \mathcal{S}_F$ then the path terminates at s_i while otherwise the path continues. If the run ends in $s_n \in \mathcal{S}_F$, we say that it is **unsafe**. A policy π is **unsafe** if there exists an unsafe run, and is **safe** otherwise.

A state s is **safe** if there exists a policy s.t. no run starting from s is unsafe. We denote by $\mathcal{S}_\top \subset \mathcal{S}$ the set of all safe states, called the **safe region**. All other states form the **unsafe region** $\mathcal{S}_\bot = \mathcal{S} \setminus \mathcal{S}_\top$. Failed states are unsafe, i.e., $\mathcal{S}_F \subseteq \mathcal{S}_\bot$, but in general $\mathcal{S}_\bot \not\subseteq \mathcal{S}_F$ (e.g. in $\mathcal{S}_\bot$ a car may be too fast to break, while in $\mathcal{S}_F$ it has already crashed). We say that the state space Θ is safe if all initial states are safe, $\mathcal{S}_0 \subseteq \mathcal{S}_\top$. Note that this is the case iff a safe policy exists.

We consider learned action policies represented by **tree ensembles** $\boldsymbol{T} = \{T_1, T_2, \dots T_M\}$. A **decision tree** $T_i : \mathbb{R}^m \to \mathbb{R}^n$ maps an input over m features to n values. In our case, $m = |\mathcal{V}|$ and $n = |\mathcal{A}|$. Each internal node of T_i stores a test of the form $X_k < \alpha$, where X_k is an input attribute and $\alpha \in \mathbb{R}$, and T_i contains branches for both possible outcomes of that test. Each *leaf node* l stores a vector of values $\overrightarrow{\nu_l} \in \mathbb{R}^n$. Every input $x \in \mathbb{R}^m$ unambigously identifies a path through T_i. The leaf node reached by this path is denoted by $l_x^{(i)}$. The prediction of T_i is $T_i(x) := \overrightarrow{\nu_{l_x^{(i)}}}$. The tree ensemble additively combines the predictions of the individual decision trees: $\boldsymbol{T}(x) := \sum_{i=1}^{M} T_i(x)$. The **output configuration** (Devos et al. 2023) of x under $\boldsymbol{T}$ is the tuple $OC(x) := \langle l_x^{(1)}, \dots, l_x^{(M)} \rangle$. Note that the output configuration precisely determines the tree ensemble's predictions. The **tree-ensemble policy** associated with $\boldsymbol{T}$ chooses the applicable action with highest value, i.e., $\pi_{\boldsymbol{T}}(s) := \text{argmax}_{a \in app(s)} \boldsymbol{T}(s)[a]$, where $\overrightarrow{\nu}[a]$ is the entry in the value vector corresponding to action a.

3 Faults

As previously outlined, faults are policy decisions one of whose outcomes leads from the safe to the unsafe region:

Definition 1. *(Jain et al. 2025) For a policy π and state s, $(s, \pi(s))$ is a **fault** if 1. s is contained in a run of π, 2. $s \in$*

$\mathcal{S}_\top$, and 3. there exists a transition $s \to_{\pi(s)} s'$ s.t. $s' \in \mathcal{S}_\bot$.

Intuitively, faults identify the causes of unsafe behavior in the given policy π. Every fault lies on an unsafe run, and pinpoints where that run enters the unsafe region. In particular, the following is easy to see:

Proposition 1. *A policy π is safe only if it does not have faults. In safe Θ, the "if" direction also holds.*

The first claim holds simply because a safe policy never enters the unsafe region. For the second claim, observe that, in unsafe Θ, an unsafe policy run may never touch the safe region, and thus does not contain any faults. In safe Θ, this cannot happen, so every unsafe run contains a fault, and thus if there are no faults then there are no unsafe runs.

All that said, in general, unsafe behavior may be needed to reach the goal (e.g., crossing a road while risking to be hit by a meteorite). It then does not necessarily make sense to "fix" a fault. Hence a discussion is in order about the assumptions we make on the application context of our work, motivating the notion of faults and the need to fix them.

The canonical scenario we address is that where the user wants to obtain a policy π for an MDP with initial state uncertainty, a goal, and a failure condition. The user's *ideal objective (O1)* assumes that a safe policy exists and requires finding a safe π that optimizes some objective like goal probability or cost-to-goal among the safe policies. The MDP is too large to obtain an optimal policy w.r.t. (O1) using symbolic methods, hence the user employs ML methods instead. For example, the user may do RL on a *proxy objective (O2)* maximizing expected discounted reward where goal states give positive rewards whereas failed states give negative rewards (e.g. (García and Fernández 2015; Zhao et al. 2023)). Our safety debugging loop then aims to make the learned π safe. This can be understood as addressing the *safety objectve (O3)* which merely requires π to be safe. For this purpose, we can drop the probabilities, resulting in our nondeterministic state space Θ as specified in Section 2. Since (O1) poses safety as a hard constraint, every violation of (O3) also is a violation of (O1). In other words, every fault must be fixed: if $(s, \pi(s))$ is a fault, then any policy optimal w.r.t. (O1) chooses a different action $a \neq \pi(s)$ in s.

The main limiting assumption in this scenario is that a safe policy exists, i.e., that Θ is safe. Out of the 13 benchmark domains in our experiments, 12 fall into this class. The remaining benchmark exemplifies a more general scenario, where the user's (O1) imposes safety *within the safe region, i.e., wherever possible*. Then, still, every fault must be fixed. For example, we cannot guarantee safe driving on icy roads, but nevertheless we do not want to run red lights.

Another interesting generalization is that where (O1) enforces an upper bound $B > 0$ on failure probability. Then a fault is an action decision that loses the ability to satisfy B. While the fault fixing methods we introduce here apply unchanged to this setting, extending fault analysis and verification is challenging and remains a topic for future work.

4 The Policy Safety Debugging Loop

Consider again the debugging loop sketched in Fig. 1. Given an input policy π_0, the loop starts by generating unsafe policy runs, using fuzzing methods (A). Then fault analysis (B) locates a set of faults F_0 on these unsafe runs. If faults were found, $F_0 \neq \emptyset$, fault fixing (C) produces a fix for F_0:

Definition 2 (Fix)**.** *Let $\mathcal{F}$ be a set of faults in a policy π. A policy π' is a **fix** for $\mathcal{F}$ if, for all $(s_i, a_i) \in \mathcal{F}$, $\pi'(s_i) \neq a_i$.*

The fault fixing methods we introduce here (Section 5) aim for the fixed policy π' to be as close as possible to π, so as to not deteriorate goal-reaching performance.

After the fault fixing step, the debugging loop continues with the repaired policy, denoted π_1. Across iterations, we accumulate the faults, $\mathcal{F} := \bigcup_i F_i$, thus ensuring monotonicity, with every intermediate policy π_i guaranteeing to fix all faults found so far. Naturally, in each iteration we fix only the new faults not already contained in $\mathcal{F}$. If no new faults are found, $F_i \setminus \mathcal{F} = \emptyset$, then this does not imply that π_i is safe, as both fuzzing and fault analysis are subject to time and memory limits. We can then choose to stop the debugging loop; or we can continue, investing the effort for formal safety verification (D). If the outcome is that π_i is safe, we can stop. Otherwise, the debugging continues by feeding the identified unsafe policy run back into fault analysis.

Let us briefly analyze the properties of this loop. In what follows, we assume that (a) every policy run has a non-0 probability to be generated by fuzzing; (b) on finite state spaces, fault analysis finds all faults on a given policy run; (c) bug fixing provides a fix for any given pair of policy and fault set; (d) on finite state spaces, safety verification is complete. The off-the-shelf methods we use satisfy (a), (b), and (d) (Jain et al. 2024, 2025; Anonymous 2025); our fault fixing method satisfies (c). Given this, in principle, on finite state spaces our loop will eventually fix every fault:

Theorem 2. *Given assumptions (a) – (c) and unlimited time and memory, if Θ is finite then iterating (A) fuzzing, (B) fault analysis, and (C) fault fixing fixes every fault in the input policy π_0 in finite time.*

Proof. Thanks to (a) and (b), and as every fault lies on some policy run, every fault will eventually be found; thanks to (c), the fault will then be fixed. □

Verification cannot be included here as, on unsafe Θ, it may lead to endless loops: the unsafe policy run σ^π found by verification may never touch the safe region and thus not contain any faults. Fault fixing then does not do anything, verification is called again on the same policy, and may again find the same unsafe run σ^π.[1] For safe Θ, however, our debugging loop is guaranteed to turn π_0 into a safe policy:

Theorem 3. *Given assumptions (a) – (d) and unlimited time and memory, if Θ is finite and safe then our debugging loop terminates in finite time with the outcome "safe π".*

Proof. Verification cannot lead to endless loops on safe Θ: every unsafe policy run σ^π starts in the safe region and hence contains at least one fault; thanks to (b) and (c) that fault will

[1] In our implementation, we stop the loop when this occurs. One could in theory fix the issue by going back into fuzzing instead. In our experiments this did not help though (fuzzing did not find new unsafe runs), so we omit this possibility here to keep things simple.

be found and fixed, so that σ^π is not a run of the new policy. Hence, with (a) – (c) as in Theorem 2, all faults will eventually be fixed. At this point, with Proposition 1 the policy is safe. With (d), the claim follows. □

For infinite state spaces, no theorems of this kind hold as fault analysis and verification may not terminate. In any case, the practical limits are time and memory. (B), (C), and (D) each are worst-case exponential in the number of state variables and/or policy representation size. Hence any of these steps may exhaust resources as indicated in Fig. 1.

We remark that, while in theory verification does not improve the ability to find unsafe policy runs (rather the opposite as it may lead to endless loops), in practice it is a useful complement to fuzzing. It may discover new unsafe policy runs not found by fuzzing, thus further advancing the debugging process. This happens in one of our benchmarks.

Finally, note that verification is needed to certify that all faults have been fixed. This works only on safe Θ, as on unsafe Θ all policies are unsafe. It remains an open question how to effectively prove the absence of faults in this case.

5 Repairing Tree-Ensemble Policies

We introduce fault fixing methods for tree ensemble policies. Given a set of faults $\mathcal{F} = \{(s_i, a_i)\}$ and a policy $\pi_{\boldsymbol{T}}$, our methods alter $\pi_{\boldsymbol{T}}$ into $\pi_{\boldsymbol{T}'}$ minimally while *guaranteeing that $\pi'(s_i) \neq a_i$ for all $(s_i, a_i) \in \mathcal{F}$ (cf. Def. 2).*

A simple idea is to re-train $\boldsymbol{T}'$ using a loss penalizing faulty decisions. Yet this 1. does not give the desired guarantee, nor 2. necessarily encourage minimal changes. Next, we first introduce a straightforward method that fixes 1. We then introduce more fine-grained methods that fix both 1. and 2.

5.1 A Simple Method: Penalty Trees

We can satisfy Def. 2 by adding, for every fault $(s_i, a_i) \in \mathcal{F}$, an additional decision tree P_{s_i,a_i} that heavily penalizes the choice of a_i in s_i. Specifically, a suitable P_{s_i,a_i} is given by the tree that consecutively checks, for each state variable $X_k \in \mathcal{V}$, whether (a) $X_k < s_i[X_k]$ and on the false-branch checks whether (b) $X_k < s_i[X_k] + \epsilon$. Here, $\epsilon = 1$ if X_k is an integer, and $\epsilon > 0$ chosen arbitrarily but small if X_k is real-valued. Let l_P be the leaf node reached by false-branches on (a) and true-branches on (b). We set all leaf values to 0, except the value in l_P associated with a_i. Namely, the latter is set to some value smaller than $\lambda = \nu_{min} - \nu_{max}$ where ν_{max} is the maximal and ν_{min} is the minimal possible sum of leaf values in $\boldsymbol{T}$. Repeating this construction for all faults in $\mathcal{F}$ provides the desired guarantee:

Theorem 4. *Let $\pi_{\boldsymbol{T}}$ be a tree-ensemble policy, and $\mathcal{F}$ be a set of faults. Let $\boldsymbol{T}' := \boldsymbol{T} \cup \{P_{s_i,a_i} \mid (s_i, a_i) \in \mathcal{F}\}$ be the tree ensemble $\boldsymbol{T}$ extended by the penalty trees for all faults. Then $\pi_{\boldsymbol{T}'}$ is a fix for $\mathcal{F}$.*

Proof. Thanks to our construction, each penalty tree P_{s_i,a_i} is designed such that $P_{s_i,a_i}(s)[a] = \lambda$ for $s = s_i, a = a_i$ and 0 everywhere else. Given the choice of λ, the augmented tree ensemble satisfies $\boldsymbol{T}'(s_i)[a_i] = \boldsymbol{T}(s_i)[a_i] + \lambda < \nu_{min} \leq \boldsymbol{T}(s_i)[a] = \boldsymbol{T}'(s_i)[a]$ for all $a \in app(s_i) \setminus \{a_i\}$. Since every state s_i is a fault, and hence in the safe region $\mathcal{S}_\top$, there exists an applicable action a such that, $(s_i, a) \notin \mathcal{F}$. Therewith, $\pi_{\boldsymbol{T}'}(s_i) \neq \pi_{\boldsymbol{T}}(s_i)$, as desired. □

Note that this construction, while giving the desired guarantee, suffers from two major weaknesses. First, its size grows linearly in the number of faults, which is ineffective. Second, the penalty terms apply only to these specific faults, so the fixes do not generalize to other faults. We fix both issues with the methods we present next.

5.2 Modifying Leaf Values using MILP

Inspired by results from prior work (Ren et al. 2015; Devos et al. 2025), instead of changing the structure of the tree ensemble, our fault-fixing method minimally alters the leaf values of the existing trees. We cast the problem of finding suitable leaf values to fix a set of faults as a mixed integer linear program (MILP; this sub-section). In case the MILP is unsolvable, the given faults cannot be repaired by changing leaf values only; we then add suitable additional trees to the ensemble (next sub-section).

Let L denote the total number of leaf nodes in the ensemble $\boldsymbol{T}$. For a leaf node l and action a, we refer by $\nu_{l,a}$ to the value of a in the leaf's value vector $\overrightarrow{\nu_l}$. With some abuse of notation, we denote with $\overrightarrow{\nu} \in \mathbb{R}^{L \cdot |\mathcal{A}|}$ the concatenation of the value vectors of all leaves in the ensemble $\boldsymbol{T}$. Eq. (1) shows the general optimization-problem formulation, which contains one variable $\nu'_{l,a}$ for each leaf node and action:

$$\begin{aligned} &\min_{\overrightarrow{\nu}'} \|\overrightarrow{\nu}' - \overrightarrow{\nu}\|_1 \\ \text{s.t.} \quad &\sum_{l \in OC(s_i)} \nu'_{l,a_i} < \max_{a \in app(s_i) \setminus \{a_i\}} \sum_{l \in OC(s_i)} \nu'_{l,a} \qquad (1) \\ &\qquad\qquad \text{for all } (s_i, a_i) \in \mathcal{F} \end{aligned}$$

where $\|\vec{x}\|_1$ denotes the L^1-norm of $\vec{x}$.

Let $\boldsymbol{T}'$ be the tree ensemble constructed from a solution to Eq. (1). Observe that the sum $\sum_{l \in OC(s)} \nu'_{l,a}$ represents exactly the tree ensemble's output $\boldsymbol{T}'(s)[a]$ for state s and action a, as per definition of the output configuration $OC(s)$. In this manner, the constraints in Eq. (1) enforce that the tree ensemble $\boldsymbol{T}'$ does not chose a_i on s_i, fixing the faults. The optimization objective in Eq. (1) requires the leaf-value changes to be minimal.

The maximization in Eq. (1) can be represented in a MILP via the big-M method (Vielma 2015): we introduce binary integer variables $b_{s_i,a_i,a}$ for every fault $(s_i, a_i) \in \mathcal{F}$ and applicable action $a \in app(s_i) \setminus \{a_i\}$. The constraints in Eq. (1) are then represented by the MILP constraints

$$\sum_{l \in OC(s_i)} \nu'_{l,a_i} < \sum_{l \in OC(s_i)} \nu'_{l,a} + M(1 - b_{s_i,a_i,a})$$

for all $a \in app(s_i) \setminus \{a_i\}$, where M is a sufficiently large constant, combined with the constraint $\sum_{a \in app(s_i) \setminus \{a_i\}} b_{s_i,a_i,a} \geq 1$.

Theorem 5. *Let $\pi_{\boldsymbol{T}}$ be a tree-ensemble policy, and $\mathcal{F}$ be a set of faults. Let $\boldsymbol{T}'$ be the tree ensemble with the same trees as $\boldsymbol{T}$, but with leaf values replaced according to some*

solution to Eq. (1)*. Then, $\pi_{\boldsymbol{T}'}$ is a fix for $\mathcal{F}$. If Eq.* (1) *is infeasible, then there is no $\boldsymbol{T}'$ over the same trees as $\boldsymbol{T}$ such that $\pi_{\boldsymbol{T}'}$ fixes $\mathcal{F}$.*

Proof. The constraints in Eq. (1) ensure that at least one applicable action will have a higher predicted value than a_i. Thus, Eq. (1) enforces that $\boldsymbol{T}'$ satisfies $\text{argmax}_{a \in app(s_i)} \boldsymbol{T}'(s_i)[a] \neq a_i$, for all faults $(s_i, a_i) \in \mathcal{F}$, hence $\pi_{\boldsymbol{T}'}$ is guaranteed to be a fix for $\mathcal{F}$. We can prove the second claim by contradiction. If there is an ensemble $\boldsymbol{T}'$ that fixes $\mathcal{F}$, then there is a leaf assignment to Eq. (1) which guarantees $\sum_{l \in OC(s_i)} \nu'_{l,a_i} < \max_{a \in app(s_i) \setminus \{a_i\}} \sum_{l \in OC(s_i)} \nu'_{l,a}$ for every fault $(s_i, a_i) \in \mathcal{F}$. Hence Eq. (1) must be feasible. □

In our implementation, we leverage the following observation to reduce the size of the MILP. The only leaf values that may need to be changed are those that belong to the *OC* of at least one fault. Any other leaf cannot affect the prediction for the fault states, so can be dropped from the MILP while preserving the optimal solutions.

5.3 Adding New Trees to Resolve Infeasibility

As an example where Eq. (1) is infeasible, say that $\mathcal{F} = \{(s_1, a_1), (s_2, a_2)\}$, both states share the same applicable actions $app(s_1) = app(s_2) = \{a_1, a_2\}$, and the same output configurations $OC(s_1) = OC(s_2)$. Clearly, it is impossible to alter the leaf values in a way that $\pi_{\boldsymbol{T}'}(s_1) \neq \pi_{\boldsymbol{T}'}(s_2)$, as the trees do not distinguish between s_1 and s_2.

This kind of problem can be fixed by adding additional trees that allow for more distinctions between the faulty states. We do this in a manner that guarantees making the MILP feasible eventually. To generate the trees, we make use of *irreducible infeasible subsystems (IIS)* (Gleeson and Ryan 1990). An IIS Φ is a set-minimal subset of constraints that are unsatisfiable in conjunction. Common MILP solvers allow to extract such Φ if infeasibility is detected.

Given an IIS Φ, we identify the faults $\mathcal{F}_\Phi \subseteq \mathcal{F}$ responsible for the constraints, and learn a single decision tree T_Φ that maps every state in $\mathcal{F}_\Phi$ to a different leaf via a standard learning method (Pedregosa et al. 2011). The leaf values in T_Φ are initialized to 0 universally. We set $\boldsymbol{T}' := \boldsymbol{T} \cup \{T_\Phi\}$, and re-try solving the MILP for $\boldsymbol{T}'$. This process is repeated until the MILP becomes feasible:

Theorem 6. *Let $\mathcal{F}$ be a set of faults and $\boldsymbol{T}$ be a tree ensemble for which Eq.* (1) *is infeasible. The tree-generation process described above will terminate eventually with a tree ensemble $\boldsymbol{T}'$ for which Eq.* (1) *becomes solvable.*

Proof. By the construction of T_Φ, the conjunction of constraints Φ is solvable for the augmented tree ensemble. Hence the same Φ can never re-occur. There are only finitely many constraint subsets, showing the claim. □

By initializing the new leaf values to 0, adding the trees a priori does not change any prediction of the ensemble. The solution to the MILP diverges from this initialization only as much as necessary, thus minimally changing the predictions.

To avoid infeasible MILP calls as much as possible, we conduct a simple pre-check. We check for sets of states S in $\mathcal{F}$ that have the same output configuration and where every commonly applicable action is ruled out by $\mathcal{F}$ for some $s \in S$. In that case, we learn a decision tree distinguishing the states in S, and add it to $\boldsymbol{T}$ prior to calling the MILP solver.

6 Experiments

We evaluate our two contributions, the debugging loop and our MILP-based fault fixing methods. We address the following main questions:

- **Q1.** To which extent does our debugging loop improve policy safety?
- **Q2.** Do improvements in safety come at a price in goal-reaching performance?
- **Q3.** Do our fault fixes generalize to other, unseen, faults?

We furthermore analyze the ensemble-size increase caused by our fault fixes (number of added trees), the runtime effort for debugging, the utility of verification in finding unsafe policy runs, as well as policy quality and MILP runtime as a function of the number of faults.

We next describe our benchmark set, our implementation, our methods to measure policy safety and goal-reaching performance, and our results. All our code and data will be made public if accepted.

6.1 Benchmarks

We use Jain et al. (2025)'s benchmarks. These consist of non-deterministic variants of the planning benchmarks *Blocksworld* and *Transport*; the *Beluga* domain which is an abstract version of a logistics-type problem at Airbus; control benchmarks with real-valued state variables (*Bouncing Ball*, *Inverted Pendulum*, and *Follow Car*); as well as 6 variants of a transportation domain called *1 way line* and *2 way line*, where a truck moves along a line of locations (uni-directionally or bi-directionally), with non-determinism due to icy roads. All these benchmarks have a safe state space Θ, except *1 way line Icy* where the only way to be safe is to reach the goal, but one may slip past the packages on icy roads. The control benchmarks have infinite state spaces; in all the benchmarks, the reachable state space is too large to be explicitly enumerated, even when fixing a policy, due to initial-state and/or action-outcome non-determinism.

Jain et al. (2025) considered neural policies for these benchmarks. Here we use these as teachers for imitation learning of tree-ensemble policies. The training data is generated by collecting the states and teacher predictions from 1000 teacher policy runs on random initial states. The tree ensembles are trained using XGBoost (Chen and Guestrin 2016) with fixed hyperparameter settings of max depth = 5, number of trees = 20, and learning rate = 0.4.

6.2 Implementation and Setup

We use steps (A), (B), and (D) of the debugging loop off-the-shelf from prior work as previously outlined. As the MILP solver for fixing faults, we use Gurobi 11.0 (Gurobi Optimization, LLC 2024), which offers the computation of IIS

Table 1: Result summary. %Unsafe: fraction of unsafe evaluation states $\mathcal{S}_0^E$. %Goal: fraction of sampled policy runs (100 on each $s_0 \in \mathcal{S}_0^E$) reaching the goal. Input: input policy π_0. MILP (NoVer.): output of debugging loop not using verification. MILP: output of default debugging loop, with our fault fixing method. Penalty: output of debugging loop with penalty tree fault fixing. The remaining columns pertain to the default loop. #Faults: $|\mathcal{F}|$ at end of loop. #Added Trees: number of trees added to policy. #It. Verif.: number of iterations where verification was invoked. Outcome: of default loop, specific explanations in Section 6.4. [$\mathcal{S}_0^D = \mathcal{S}_0^E$] at benchmark name: < 20000 initial states, debugging states same as evaluation states (see Sections 6.2 and 6.3).

	%Unsafe				%Goal						Runtime					
Benchmark	Input	MILP (NoVer.)	MILP	Penalty	Input	MILP (NoVer.)	MILP	Penalty	# Faults	# Added Trees	# It. Loop	# It. Verif.	Fault Ana.	Gurobi	Verif.	Outcome
Discrete state space																
8Blocks CI [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	1.1	0.0	0.0	0.0	92.0	77.7	77.7	92.8	1	0	2	1	6m	<1s	8m	Safe (IC3)
8Blocks CA [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	1.3	0.0	0.0	0.9	91.8	37.4	37.3	92.7	15	0	13	6	47m	<1s	TO	Improved
Transport	2.3	0.0	0.0	2.3	0.0	0.0	0.0	0.0	26	0	16	8	11m	<1s	TO	Improved
Beluga (4 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	0.0	0.0	79.1	85.0	100.0	100.0	94.3	203	3	10	1	10m	25s	4m	Safe (IC3)
Beluga (5 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	0.0	0.0	100.0	69.4	100.0	100.0	88.3	2574	19	22	1	49m	45h	5h	Safe (IC3)
Beluga (6 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	98.7	-	-	-	86.1	-	-	-	6196	4	3	0	22m	TO	-	Gurobi TO
1 way line (17, 10)	1.1	0.0	0.0	1.0	98.9	100.0	100.0	99.0	30	0	2	1	35s	<1s	TO	Improved
1 way line (30, 20)	100.0	1.2	1.2	95.3	0.0	98.8	98.8	4.7	11349	0	4	1	1h	3s	TO	Improved
1 way line (15, 10) Icy + Park	99.6	0.0	0.0	28.7	91.6	91.6	91.6	91.6	27	0	2	1	49s	<1s	2m	Safe (PPA)
1 way line (30, 20) Icy + Park	99.9	21.1	21.1	99.7	46.2	46.2	46.2	46.2	2431	0	3	1	59m	1s	TO	Improved
1 way line (15, 10) Icy	100.0	100.0	100.0	100.0	88.2	90.9	90.9	90.4	23	0	2	1	4m	<1s	3m	Stopped
1 way line (30, 20) Icy	100.0	-	-	-	49.7	-	-	-	0	0	1	0	OOM	-	-	FA OOM
2 way line (17, 10)	1.1	0.0	0.0	0.5	98.9	98.9	98.9	98.9	13	0	2	1	54s	<1s	TO	Improved
2 way line (30, 20)	80.9	0.3	0.3	80.9	0.0	0.0	0.0	0.0	14185	0	4	1	1m	2s	TO	Improved
2 way line (15, 10) Icy + Park	100.0	0.0	0.0	99.0	17.9	22.9	22.9	17.8	1259	0	3	1	2m	1s	1h	Safe (PPA)
2 way line (30, 20) Icy + Park	70.4	12.3	12.3	70.4	0.0	0.0	0.0	0.0	5250	0	6	1	2m	1s	TO	Improved
2 way line (15, 10) Icy	100.0	29.3	0.0	99.2	17.2	20.4	13.8	17.2	989	0	6	2	3m	<1s	6h	Safe (PPA)
2 way line (30, 20) Icy	11.3	0.0	0.0	6.0	4.5	4.5	4.5	4.5	3	0	2	1	2m	<1s	TO	Improved
Continuous state space (control benchmarks)																
Bouncing Ball ($\tau = 0.01$)	4.9	0.0	0.0	4.9	95.1	100.0	100.0	95.1	531	0	2	1	1m	<1s	OOM	Improved
Inverted Pendulum ($\tau = 0.1$)	2.9	0.0	0.0	2.9	97.1	100.0	100.0	97.1	357	0	4	1	54m	<1s	OOM	Improved
Follow Car ($\tau = 0.4$)	22.4	0.0	0.0	22.4	77.6	100.0	100.0	77.6	2224	0	2	1	2m	<1s	OOM	Improved

out of the box. Our default verifier is policy predicate abstraction (**PPA**) (Jain et al. 2024). We run the alternate verifier **IC3** (Anonymous 2025) after loop termination to check whether we can get additional safety results.

All experiments were run on an Intel i7-12700 CPU with 64GB of memory. We impose a global timeout of 72 hours on the debugging loop. We use the following limits in each iteration: 12 hours for fault identification (A)+(B); 3 hours for fixing the faults (C); 12 hours for safety verification (D).

We set the MILP-based fault fixing (Sections 5.2 and 5.3) as our default method. We compare to penalty trees (Section 5.1) only for the purpose of assessing the capacity of MILP-based fault fixing to generalize beyond the set $\mathcal{F}$ of faults (penalty trees do by construction not have that capacity). To make that comparison precise, we use the exact same set $\mathcal{F}$ for both fixing methods. Namely, we collect the faults $\mathcal{F}$ when running the debugging loop with MILP-based fault fixing. Upon termination, we compare the outcome policy to the input policy π_0 augmented with penalty trees for $\mathcal{F}$.

We run the debugging loop on a fixed set $\mathcal{S}_0^D$ of 10000 **debugging states**, sampled uniformly from $\mathcal{S}_0$. Fuzzing then samples policy runs from $\mathcal{S}_0^D$ (biasing action outcomes towards ϕ_F), and hence all faults identified and fixed by the loop pertain to runs from $\mathcal{S}_0^D$. In 5 of our benchmark instances, $|\mathcal{S}_0| < 10000$. We set $\mathcal{S}_0^D = \mathcal{S}_0$ in these cases, using all initial states for debugging.

6.3 Safety and Goal-Reaching Performance Measurements

To ensure that the effects of debugging do not only pertain to policy runs starting from $\mathcal{S}_0^D$, we evaluate policy safety and goal-reaching performance on a separate set $\mathcal{S}_0^E$ of 10000 **evaluation states**. These are also sampled uniformly from $\mathcal{S}_0$, but with a duplicate check ensuring that $\mathcal{S}_0^E \cap \mathcal{S}_0^D = \emptyset$. The exception are benchmarks where $|\mathcal{S}_0| < 20000$. These are the same 5 benchmarks where $|\mathcal{S}_0| < 10000$, so we set $\mathcal{S}_0^E = \mathcal{S}_0^D = \mathcal{S}_0$ in these cases (the 5 benchmarks are indicated in Table 1). Note here that, when $\mathcal{S}_0$ is small enough to be enumerated, it does not matter in practice whether or not the debugging effects transfer across initial states.

We measure policy goal-reaching performance, denoted **%Goal**, by sampling 100 policy runs from every evaluation state $s_0 \in \mathcal{S}_0^E$ and showing the fraction of these runs that reached the goal. We measure policy safety, denoted **%Unsafe**, by the fraction of $s_0 \in \mathcal{S}_0^E$ on which the policy is unsafe. We check safety from each individual $s_0 \in \mathcal{S}_0^E$ by exhaustive enumeration of the states reachable from s_0 under the given policy. If that enumeration is not feasible, we resort to sampling 100 policy runs from s_0.

Note that it is possible for a policy to be unsafe and still

have %Unsafe $= 0$, as the evaluation is based on $\mathcal{S}_0^E$, while $\mathcal{S}_0$ is exponentially large in general.

6.4 Results

Table 1 summarizes our results. For space reasons, we omit some benchmark instances; the conclusions remain the same. The complete table is in the supplementary material.

Policy safety improvement (Q1) Consider the %Unsafe part of the table. First, observe that, *consistently across all benchmarks, this measure decreases from the input policy π_0 ("Input") to the output policy of our debugging loop "MILP")*. The single exception is *1 way line Icy* where such a decrease is impossible, as the state space is unsafe and hence %Unsafe $=$ 100 for every policy. For benchmarks where π_0 already is fairly safe (low %Unsafe, e.g. *Blocksworld*), the measure decreases further; for the other benchmarks, the decreases are drastic, $> 70\%$ in 8 cases.

In 6 of our benchmarks, the outcome policy is verified to be safe (see the "Outcome" column at the right-hand side of Table 1). In the "Improved" outcomes, verification ran out of time or memory, but %Unsafe decreased. Gurobi TO means Gurobi ran out of time, FA OOM means fault analysis ran out of memory. The "Stopped" outcome pertains to *1 way line Icy*, the only benchmark where no safe policy exists. Here, verification found an unsafe run without faults, and as previously mentioned we stop the debugging process to avoid running into an endless loop.

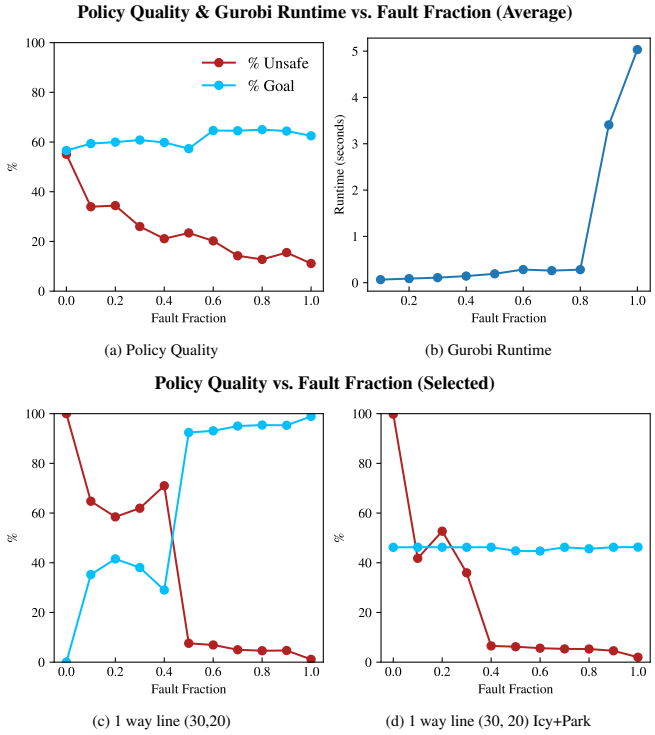
The "NoVer" column shows data for debugging not using verification, iterating fuzzing, fault analysis, and fault fixing until either no more faults are found or time/memory limits are reached. This variant has the same %Unsafe except in a single case (*2 way line (15, 10) Icy*), so fuzzing is typically enough to detect unsafety. This is of high practical relevance as fuzzing is much cheaper than formal verification; prior works on model repair did not include fuzzing methods.

Goal-reaching performance (Q2) The safety improvements come at *no price in goal-reaching performance*. Rather the opposite, in many cases %Goal increases thanks to fault fixing. This makes sense as unsafety is one reason for not reaching the goal. The single exception to this pattern is *Blocksworld*, where fault fixing led to cyclic behavior.

Generalization to unseen faults (Q3) Comparing the "MILP" and "Penalty" columns, observe that %Unsafe is consistently smaller – often much smaller – for MILP. This is true in all benchmarks except *8Blocks CI* (and *1 way line Icy* where %Unsafe $=$ 100 by definition). This data impressively attests to the generalization capacity of our fault-fixing method, reducing unsafe behavior much beyond the specific faults $\mathcal{F}$ used for debugging.

Tree-Ensemble size increase The safety improvements are achieved with hardly any increase in tree-ensemble size. In fact, *in all but Beluga, no additional trees need to be added at all!* The entire fault fixing is accomplished by modifying the leaf values. In *Beluga*, the size increase is partly substantial (remember that π_0 has 20 trees); but in return, in 2 of the 3 instances we go from a 100%Unsafe input policy to a verified-safe output policy.

Computational effort for debugging The most significant portion of debugging runtime is spent on verification, with many instances timing out after 12h. Fault analysis can take substantial time, but much less than verification. Fault fixing takes negligible time except in *Beluga*. Without verification, the overall debugging loop runtime never exceeds 1 hour in our experiments.



(a) Policy Quality (b) Gurobi Runtime

(c) 1 way line (30,20) (d) 1 way line (30, 20) Icy+Park

Figure 2: Policy quality (%Unsafe, %Goal) and Gurobi runtime vs. fraction of $\mathcal{F}$ used.

Policy quality and MILP runtime as a function of the number of faults Consider Figure 2. In (a) we show average %Unsafe and %Goal as a function of fractions of $\mathcal{F}$. The clear tendency is for %Unsafe to decrease. So, as one would expect, fixing more faults is better. The average trend for %Goal is not as clear, as the behavior differs across benchmarks. Fig. 2c and Fig. 2d show representative examples.

Regarding runtime (Fig. 2b) we also get the expected increase as a function of the number of faults, though the larger runtimes here are dominated by *Beluga* as discussed above.

7 Conclusion

We introduce the first safety debugging loop for learned action policies. Focusing on tree ensembles, we introduce repair methods that guarantee to fix a given set of undesired input/output pairs. Combining this with methods from prior work into our debugging loop, we obtain reliable and substantial safety improvements, mostly at no cost in goal-reaching performance, across a range of benchmarks.

These results are highly encouraging for further research on action policy debugging. There is a universe of possibil-

ities, spanned by the dimensions of which kind of environment model, which kind of ML model, and which debugging objectives are considered. As a direct extension of our specific setting here, a scenario of interest is that where the user wishes to impose an upper bound $B > 0$ on failure probability. To this end, faults should be defined relative to a larger "safe region" where that bound B is satisfied.

Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" - project number 471607914, the Research Foundation Flanders (FWO, LC: 11I8125N), and the Flemish Government under the "Onderzoeksprogramma Artifciële Intelligentie (AI) Vlaanderen" program (JD).

References

Akintunde, M. E.; Kevorchian, A.; Lomuscio, A.; and Pirovano, E. 2019. Verification of RNN-Based Neural Agent-Environment Systems. In *33rd AAAI Conference on Artificial Intelligence (AAAI'19)*, 6006–6013.

Amir, G.; Corsi, D.; Yerushalmi, R.; Marzari, L.; Harel, D.; Farinelli, A.; and Katz, G. 2023. Verifying Learning-Based Robotic Navigation Systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, 607–627. Springer.

Anonymous. 2025. IC3 for Safety Verification of Tree-ensemble Policies. Manuscript in preparation for submission.

Bacci, E.; Giacobbe, M.; and Parker, D. 2021. Verifying Reinforcement Learning up to Infinity. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 2154–2160.

Bacci, E.; and Parker, D. 2022. Verified Probabilistic Policies for Deep Reinforcement Learning. In *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *LNCS*, 193–212. Springer.

Bauer-Marquart, F.; Boetius, D.; Leue, S.; and Schilling, C. 2022. SpecRepair: Counter-Example Guided Safety Repair of Deep Neural Networks. In *Model Checking Software: 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings*, 79–96.

Boetius, D.; Leue, S.; and Sutter, T. 2023. A Robust Optimisation Perspective on Counterexample-Guided Repair of Neural Networks. In *Proceedings of the 40th International Conference on Machine Learning: ICML 2023*.

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'17)*, 151–168.

Chen, T.; and Guestrin, C. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. 147(1–2): 35–84.

Devos, L.; Martens, T.; Oruc, D. C.; Meert, W.; Blockeel, H.; and Davis, J. 2025. Compressing tree ensembles through Level-wise Optimization and Pruning. In *Forty-second International Conference on Machine Learning*.

Devos, L.; Perini, L.; Meert, W.; and Davis, J. 2023. Detecting Evasion Attacks in Deployed Tree Ensembles. In *Proceeding of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 120–136.

Dong, G.; Sun, J.; Wang, X.; Wang, X.; and Dai, T. 2021. Towards Repairing Neural Networks Correctly. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 714–725.

Dutta, S.; Chen, X.; and Sankaranarayanan, S. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd International Conference on Hybrid Systems: Computation and Control (HSCC'19)*, 157–168.

García, J.; and Fernández, F. 2015. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16: 1437–1480.

Garg, S.; Bajpai, A.; et al. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*, 631–636.

Gleeson, J.; and Ryan, J. 1990. Identifying Minimally Infeasible Subsystems of Inequalities. *INFORMS J. Comput.*, 2(1): 61–63.

Goldberger, B.; Katz, G.; Adi, Y.; and Keshet, J. 2020. Minimal Modifications of Deep Neural Networks using Verification. In *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, 260–278.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies using Deep Neural Networks. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 408–416.

Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual.

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'18)*, 422–430.

Ivanov, R.; Carpenter, T. J.; Weimer, J.; Alur, R.; Pappas, G. J.; and Lee, I. 2021. Verifying the Safety of Autonomous Systems with Neural Network Controllers. *ACM Transactions on Embedded Computing Systems*, 20(1): 7:1–7:26.

Jain, C.; Cascioli, L.; Devos, L.; Vinzent, M.; Steinmetz, M.; Davis, J.; and Hoffmann, J. 2024. Safety Verification of Tree-Ensemble Policies via Predicate Abstraction. In *Proceedings of the 27th European Conference on Artificial Intelligence (ECAI'24)*.

Jain, C.; Sherbakov, D.; Vinzent, M.; Steinmetz, M.; Davis, J.; and Hoffmann, J. 2025. Policy Safety Testing in Non-Deterministic Planning: Fuzzing, Test Oracles, Fault Analysis. In *Proceedings of the 28th European Conference on Artificial Intelligence (ECAI'25)*. Available at http://fai.cs.uni-saarland.de/jain/papers/ecai25.pdf.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*, 8064–8073.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M. A.; Fidjeland, A.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830.

Ren, S.; Cao, X.; Wei, Y.; and Sun, J. 2015. Global refinement of random forest. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 723–730.

Rossetti, N.; Tummolo, M.; Gerevini, A. E.; Putelli, L.; Serina, I.; Chiari, M.; and Olivato, M. 2024. Learning General Policies for Planning through GPT Models. *Proceedings of the International Conference on Automated Planning and Scheduling*, 34(1): 500–508.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529: 484–503.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Sotoudeh, M.; and Thakur, A. V. 2021. Provable repair of deep neural networks. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 588–603.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'22)*, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning Generalized Policies without Supervision Using GNNs. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning (KR'22)*.

Tan, C.; Zhu, Y.; and Guo, C. 2021. Building verified neural networks with specifications for systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '21, 42–47.

Toyer, S.; Thiébaux, S.; Trevizan, F. W.; and Xie, L. 2020. ASNets: Deep Learning for Generalised Planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Vielma, J. P. 2015. Mixed Integer Linear Programming Formulation Techniques. *SIAM Review*.

Vinzent, M.; Sharma, S.; and Hoffmann, J. 2023. Neural Policy Safety Verification via Predicate Abstraction: CEGAR. In *AAAI*, 15188–15196.

Vinzent, M.; Steinmetz, M.; and Hoffmann, J. 2022. Neural Network Action Policy Verification via Predicate Abstraction. In *Proceedings of the 29th International Conference on Automated Planning and Scheduling (ICAPS'22)*.

Wang, Y.; Zhou, W.; Fan, J.; Wang, Z.; Li, J.; Chen, X.; Huang, C.; Li, W.; and Zhu, Q. 2024. POLAR-Express: Efficient and Precise Formal Reachability Analysis of Neural-Network Controlled Systems. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 43(3): 994–1007.

Zavatteri, M.; Bresolin, D.; and Navarin, N. 2024. Automated Synthesis of Certified Neural Networks. In *Proceedings of the 27th European Conference on Artificial Intelligence*.

Zhao, W.; He, T.; Chen, R.; Wei, T.; and Liu, C. 2023. State-wise Safe Reinforcement Learning: A Survey. In Elkind, E., ed., *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI-23*, 6814–6822. International Joint Conferences on Artificial Intelligence Organization. Survey Track.

A. Benchmark description

A benchmark instance is a pair of non-deterministic planning task described in the JANI language (Budde et al. 2017) and a tree ensemble policy solving that problem. Our benchmark set is taken from prior work on fault analysis for action policies (Jain et al. 2025). It is composed of two parts:

Discrete state space benchmarks These are benchmarks (*Beluga*, two variants of *Blockworld* and *Transport*) from prior work on action-policy safety verification (Vinzent, Steinmetz, and Hoffmann 2022; Jain et al. 2024) as well as *1 way line* and *2 way line*.

In *Beluga*, inspired from a logistics application at Airbus, product parts arriving on a Beluga transport plane must be sent to a factory in a particular order, potentially different from the order in which they are arriving. There is a number of racks in which arriving parts can be stored temporarily. The start condition permits arbitrary orderings of the arriving parts. The goal requires having sent all parts to the factory, under non-determinism which part will be requested next at each step. A state is unsafe if parts are loaded and unloaded on racks too often (modelling the practical guideline to only deliver parts when requested).

In *Blocksworld*, actions moving a block b may non-deterministically fail, and when this happens the cost of moving block b is incremented. The start condition imposes a partial order on the blocks in the initial stacks. A state is unsafe if the number of blocks on the table exceeds a fixed limit. We consider instances with 8 blocks.

In *Transport*, a truck must deliver packages on a straight line road to the other side of a bridge, and an unsafe state occurs if the truck has too many packages while crossing the bridge. The start condition distributes the packages on all locations on the "non-goal" side of the bridge.

In *1 way line* and *2 way line*, a truck moves along a discrete line in one respectively both directions. The truck can accelerate and decelerate by one speed unit at a time, and pick up and drop packages if its velocity is 0. Non-determinism might cause the truck to drop packages while moving. Acceleration and deceleration might fail. We additionally consider variants with an additional parking action and one disabling non-determinism. The safety constraint requires the policy to not drive past either end of the line.

Continuous state space benchmarks (control benchmarks) These are JANI benchmarks modeling the deterministic variants of control problems *Bouncing Ball*, *Follow Car* and *Inverted Pendulum* often considered in reinforcement learning (Bacci, Giacobbe, and Parker 2021; Bacci and Parker 2022). All control benchmarks are parameterized by a timestep τ, which determines how frequently the system is updated. A smaller τ enables more fine-grained control, while a larger τ results in less frequent but more substantial updates to the system dynamics.

In *Bouncing Ball*, the agent controls a ball choosing to hit it with a paddle or do nothing. The ball loses 10% of its energy upon hitting the ground and eventually stops bouncing. Safety requires continuously bouncing.

In *Follow Car*, there are two vehicles *lead* and *ego*. The lead vehicle is moving at a constant speed whilst the agent controls the acceleration of the ego vehicle. The safety constraint is the two vehicles never crash.

In *Inverted Pendulum* (henceforth referred to as *Inv Pendulum*), an agent applies left or right rotational force to a pole pivoting around one of its ends, with the aim of balancing the pole in the upright position. Safety constitutes remaining in a range of positions and velocities such that the upright position can be recovered.

B. Computing penalty values for Penalty Trees

In Section 5.1 we define the penalty value λ used for the penalty trees as $\lambda = \nu_{min} - \nu_{max}$ where ν_{max} is the maximal and ν_{min} is the minimal possible sum of leaf values in $\boldsymbol{T}$. Exactly computing this value is computationally challenging, as one would only need to take into account combinations of leaves in the ensemble belonging to a valid *OC* (e.g., an *OC* where one leaf enforces $X_k < 1$ and another leaf imposes $X_k > 2$ is invalid and should not be taken into account). However in practice, any overestimation of λ is also perfectly valid as a penalty. We compute one such overestimation as follows. For each tree T in the tree ensemble, we recursively traverse the tree and collect, for each action label a_i, the minimum and maximum values assigned to a_i at any of the leaves. This results in a set of per-action bounds:

$$\left\{ \left(\min_{l \in \mathcal{L}_T} \nu_{l,a_i}, \max_{l \in \mathcal{L}_T} \nu_{\ell,a_i} \right) \right\}_{a_i \in \mathcal{A}}$$

where $\mathcal{L}_T$ denotes the set of leaf nodes in tree T, and ν_{l,a_i} is the value assigned to class a_i at leaf l.

From this set of bounds, we extract the overall minimum and maximum values across all actions for tree T:

$$\nu^T_{\min} = \min_{a_i \in \mathcal{A}} \min_{l \in \mathcal{L}_T} \nu_{l,a_i}, \quad \nu^T_{\max} = \max_{a_i \in \mathcal{A}} \max_{l \in \mathcal{L}_T} \nu_{l,a_i}.$$

The penalty component related to tree T is then defined as:

$$\lambda_T = \nu^T_{\min} - \nu^T_{\max}$$

which is guaranteed to be non-positive.

Finally, the global penalty λ is the sum of these values across all trees:

$$\lambda = \sum_{T \in \boldsymbol{T}} \lambda_T$$

This formulation ensures that the employed penalty value is strictly smaller than any valid prediction produced by the ensemble.

C. Extended Experimental Results

Table 2 reports the experiment from Table 1 on our entire set of benchmarks.

Extended Ablation Study

Fig. 3 and Fig. 4 extend Fig. 2 by reporting specific quality and runtime plots for each of the benchmarks we used to generate the average performances shown in Fig. 2 (upper).

Table 2: Result summary on our entire set of benchmarks. %Unsafe: fraction of unsafe evaluation states $\mathcal{S}_0^E$. %Goal: fraction of sampled policy runs (100 on each $s_0 \in \mathcal{S}_0^E$) reaching the goal. Input: input policy π_0. MILP (NoVer.): output of debugging loop not using verification. MILP: output of default debugging loop, with our fault fixing method. Penalty: output of debugging loop with penalty tree fault fixing. The remaining columns pertain to the default loop. #Faults: $|\mathcal{F}|$ at end of loop. #Added Trees: number of trees added to policy. #It. Verif.: number of iterations where verification was invoked. Outcome: of default loop, specific explanations in Section 6.4. [$\mathcal{S}_0^D = \mathcal{S}_0^E$] at benchmark name: < 20000 initial states, debugging states same as evaluation states (see Sections 6.2 and 6.3).

	%Unsafe				%Goal								Runtime			
Benchmark	Input	MILP (NoVer.)	MILP	Penalty	Input	MILP (NoVer.)	MILP	Penalty	# Faults	# Added Trees	# It. Loop	# It. Verif.	Fault Ana.	Gurobi	Verif.	Outcome
Discrete state space																
8Blocks CI [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	1.1	0.0	0.0	0.0	92.0	77.7	77.7	92.8	1	0	2	1	6m	<1s	8m	Safe (IC3)
8Blocks CA [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	1.3	0.0	0.0	0.9	91.8	37.4	37.3	92.7	15	0	13	6	47m	<1s	TO	Improved
Transport	2.3	0.0	0.0	2.3	0.0	0.0	0.0	0.0	26	0	16	8	11m	<1s	TO	Improved
Beluga (3 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	0.0	0.0	83.6	82.3	100.0	100.0	83.3	33	0	6	1	4m	<1s	19s	Safe (PPA)
Beluga (4 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	0.0	0.0	79.1	85.0	100.0	100.0	94.3	203	3	10	1	10m	25s	4m	Safe (IC3)
Beluga (5 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	0.0	0.0	100.0	69.4	100.0	100.0	88.3	2574	19	22	1	49m	45h	5h	Safe (IC3)
Beluga (6 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	98.7	-	-	-	86.1	-	-	-	6196	4	3	0	22m	TO	-	Gurobi TO
Beluga (8 Jigs) [$\mathcal{S}_0^D = \mathcal{S}_0^E$]	100.0	-	-	-	70.7	-	-	-	20420	5	2	0	1h	TO	-	Gurobi TO
1 way line (15, 10)	100.0	0.1	0.1	98.6	0.0	99.9	99.9	1.4	8053	0	3	1	1m	<1s	TO	Improved
1 way line (17, 10)	1.1	0.0	0.0	1.0	98.9	100.0	100.0	99.0	30	0	2	1	35s	<1s	TO	Improved
1 way line (20, 10)	100.0	0.2	0.2	30.0	0.0	99.8	99.8	70.0	3797	0	3	1	1m	<1s	TO	Improved
1 way line (30, 20)	100.0	1.2	1.2	95.3	0.0	98.8	98.8	4.7	11349	0	4	1	1h	3s	TO	Improved
1 way line (40, 25)	-	-	-	-	0.0	-	-	-	0	0	1	0	OOM	-	-	FA OOM
1 way line (15, 10) Icy + Park	99.6	0.0	0.0	28.7	91.6	91.6	91.6	91.6	27	0	2	1	49s	<1s	2m	Safe (PPA)
1 way line (17, 10) Icy + Park	99.5	0.0	0.0	29.9	90.2	90.2	90.2	90.2	46	0	2	1	52s	<1s	1h	Safe (PPA)
1 way line (20, 10) Icy + Park	99.5	1.3	1.3	28.2	91.5	91.5	91.5	91.5	54	0	4	2	1m	<1s	TO	Improved
1 way line (30, 20) Icy + Park	99.9	21.1	21.1	99.7	46.2	46.2	46.2	46.2	2431	0	3	1	59m	1s	TO	Improved
1 way line (40, 25) Icy + Park	54.4	44.8	44.8	54.4	44.5	44.5	44.5	44.5	209	0	3	1	13m	<1s	TO	Improved
1 way line (15, 10) Icy	100.0	100.0	100.0	100.0	88.2	90.9	90.9	90.4	23	0	2	1	4m	<1s	3m	Stopped
1 way line (17, 10) Icy	100.0	100.0	100.0	100.0	87.2	91.6	91.6	91.6	8	0	2	1	6m	<1s	TO	Stopped
1 way line (20, 10) Icy	100.0	100.0	100.0	100.0	90.8	91.6	91.6	91.3	19	0	2	1	10m	<1s	TO	Improved
1 way line (30, 20) Icy	100.0	-	-	-	49.7	-	-	-	0	0	1	0	OOM	-	-	FA OOM
1 way line (40, 25) Icy	100.0	-	-	-	54.0	-	-	-	0	0	1	0	OOM	-	-	FA OOM
2 way line (15, 10)	100.0	0.0	0.0	99.3	0.0	0.0	0.0	0.0	9511	0	3	1	1m	<1s	TO	Improved
2 way line (17, 10)	1.1	0.0	0.0	0.5	98.9	98.9	98.9	98.9	13	0	2	1	54s	<1s	TO	Improved
2 way line (20, 10)	100.0	0.1	0.1	33.3	0.0	0.0	0.0	0.0	7304	0	7	1	2m	3s	TO	Improved
2 way line (30, 20)	80.9	0.3	0.3	80.9	0.0	0.0	0.0	0.0	14185	0	4	1	1m	2s	TO	Improved
2 way line (40, 25)	100.0	0.1	0.1	99.2	0.0	0.0	0.0	0.0	9967	0	6	1	3m	1s	TO	Improved
2 way line (15, 10) Icy + Park	100.0	0.0	0.0	99.0	17.9	22.9	22.9	17.8	1259	0	3	1	2m	1s	1h	Safe (PPA)
2 way line (17, 10) Icy + Park	100.0	0.0	0.0	99.2	18.2	21.3	21.3	18.2	911	0	2	1	1m	<1s	TO	Improved
2 way line (20, 10) Icy + Park	100.0	1.4	1.4	99.3	17.9	17.8	17.9	17.9	1483	0	4	2	2m	<1s	TO	Improved
2 way line (30, 20) Icy + Park	70.4	12.3	12.3	70.4	0.0	0.0	0.0	0.0	5250	0	6	1	2m	1s	TO	Improved
2 way line (40, 25) Icy + Park	65.6	11.3	11.3	65.0	0.7	0.8	0.8	0.7	573	0	2	1	27m	<1s	TO	Improved
2 way line (15, 10) Icy	100.0	29.3	0.0	99.2	17.2	20.4	13.8	17.2	989	0	6	2	3m	<1s	6h	Safe (PPA)
2 way line (17, 10) Icy	100.0	45.9	0.0	99.7	16.9	23.9	24.8	17.9	1884	0	8	2	5m	4s	35m	Safe (PPA)
2 way line (20, 10) Icy	100.0	2.7	0.1	99.7	17.7	26.1	26.1	17.7	2264	0	3	1	2m	<1s	TO	Improved
2 way line (30, 20) Icy	11.3	0.0	0.0	6.0	4.5	4.5	4.5	4.5	3	0	2	1	2m	<1s	TO	Improved
2 way line (40, 25) Icy	100.0	1.6	1.6	99.8	0.8	1.2	1.2	0.8	8856	0	4	1	13h	24s	TO	Improved
Continuous state space (control benchmarks)																
Bouncing Ball ($\tau = 0.01$)	4.9	0.0	0.0	4.9	95.1	100.0	100.0	95.1	531	0	2	1	1m	<1s	OOM	Improved
Bouncing Ball ($\tau = 0.05$)	2.8	0.0	0.0	2.8	97.2	100.0	100.0	97.2	292	0	2	1	1m	<1s	OOM	Improved
Bouncing Ball ($\tau = 0.5$)	0.1	0.0	0.0	0.1	99.9	100.0	100.0	99.9	5	0	2	1	1m	<1s	OOM	Improved
Inverted Pendulum ($\tau = 0.01$)	-	-	-	-	0.0	-	-	-	0	0	1	0	OOM	-	-	FA OOM
Inverted Pendulum ($\tau = 0.05$)	0.5	-	-	-	99.5	-	-	-	0	0	1	0	OOM	-	-	FA OOM
Inverted Pendulum ($\tau = 0.1$)	2.9	0.0	0.0	2.9	97.1	100.0	100.0	97.1	357	0	4	1	54m	<1s	OOM	Improved
Follow Car ($\tau = 0.01$)	-	-	-	-	0.0	-	-	-	0	0	1	0	OOM	-	-	FA OOM
Follow Car ($\tau = 0.05$)	-	-	-	-	19.3	-	-	-	0	0	1	0	OOM	-	-	FA OOM
Follow Car ($\tau = 0.4$)	22.4	0.0	0.0	22.4	77.6	100.0	100.0	77.6	2224	0	2	1	2m	<1s	OOM	Improved

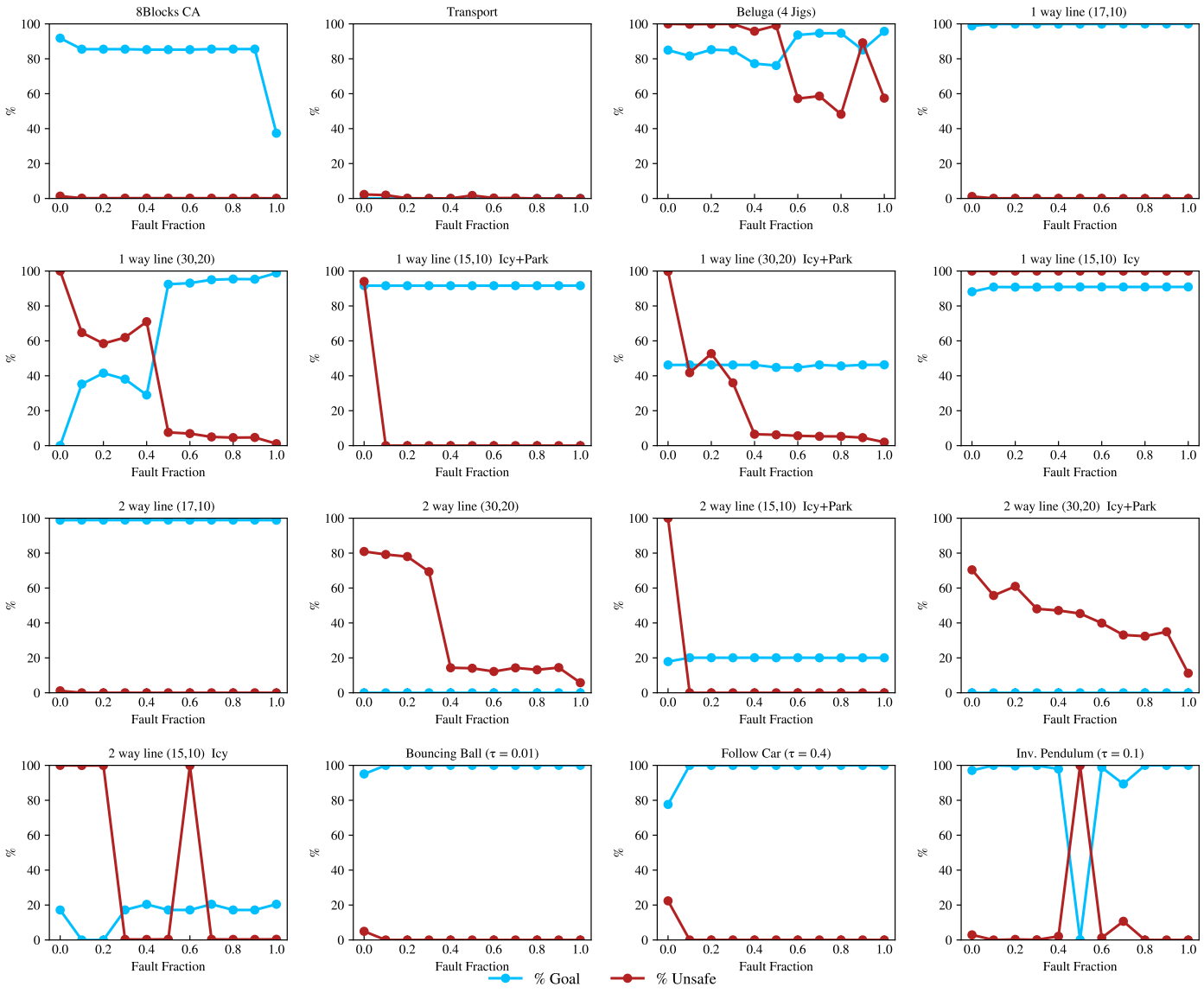


Figure 3: Policy quality (%Goal, %Unsafe) vs. fraction of $\mathcal{F}$ used.

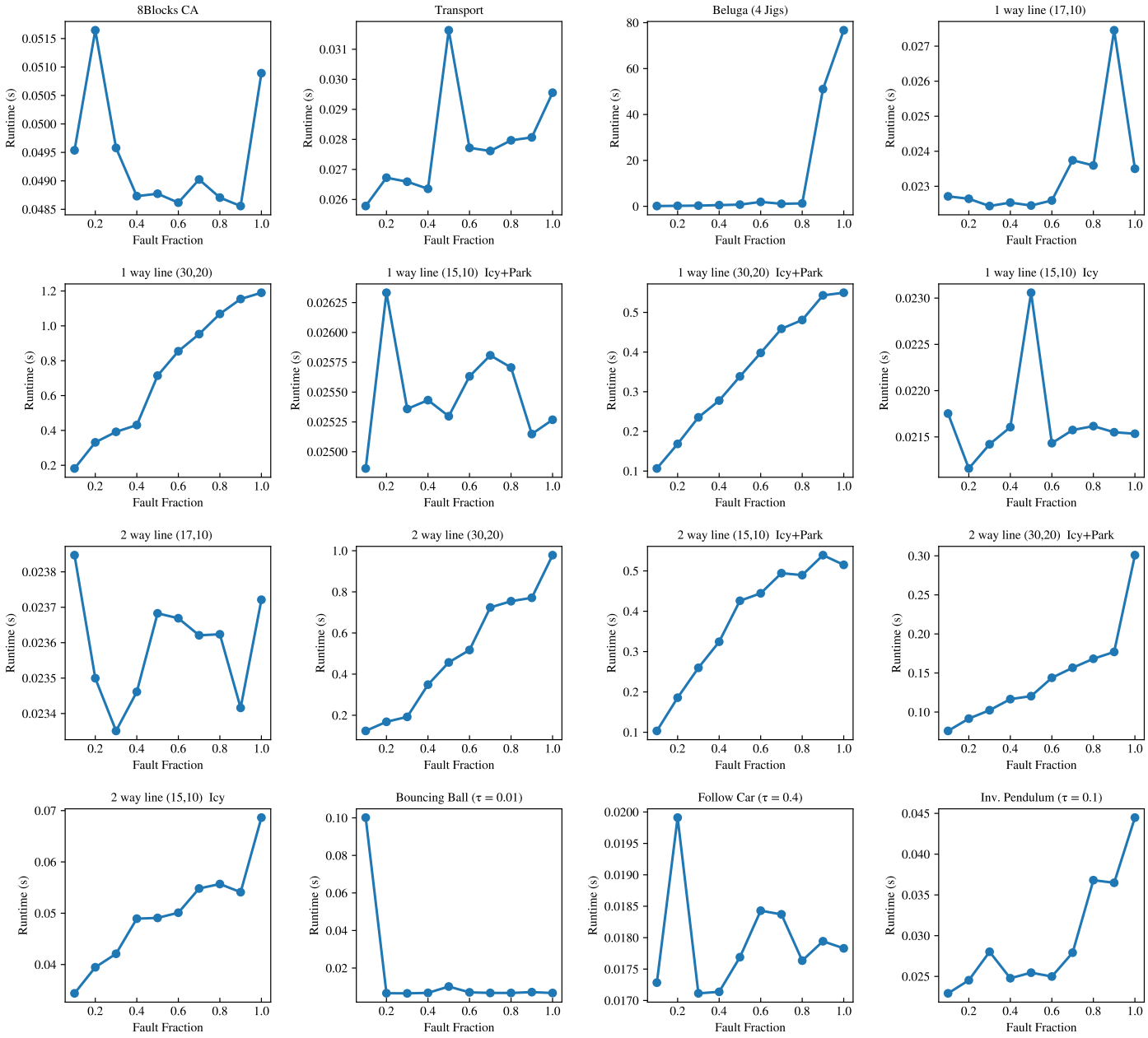


Figure 4: Gurobi runtime vs. fraction of $\mathcal{F}$ used.