Adapting to Novelties in Numeric Planning

Nir Aharoni[1], **Yarin Benyamin**[1], **Shiwali Mohan**[2], **Roni Stern**[1]

[1]Ben-Gurion University of the Negev
[2]Future Concepts Division, SRI International
{nirahar,bnyamin}@post.bgu.ac.il, shiwali.mohan@gmail.com, roni.stern@gmail.com

Abstract

Planning agents rely on accurate PDDL domain descriptions to solve planning problems, particularly in complex environments involving numeric conditions and effects. However, real-world environments often change over time, introducing novelties that invalidate parts of the domain model. Even small changes, such as altered numeric preconditions or updated effect values, can cause the planner to generate faulty plans, ultimately leading to failure in achieving goals. In this work, we focus on the problem of adapting to such novelties in numeric planning domains. We propose a method for detecting run-time failures caused by mismatches between the domain model and the environment, and introduce a learning-based repair mechanism that adjusts the numeric components of the domain to restore some planning effectiveness. Even without fully learning the exact environmental changes, our approach improves planning accuracy and helps maintain effectiveness. Experimental results on benchmark numeric planning domains demonstrate the effectiveness of our repair strategy in restoring agent performance under various types of environmental change.

Introduction

Classical planning is a form of automated planning where an agent seeks a sequence of actions that transforms an initial state into a goal state. The world is represented using Boolean predicates, and actions have deterministic effects defined by preconditions and outcomes that add or remove these predicates. Numeric planning is an extension of classical planning. While classical planning involves reasoning over Boolean predicates that represent the presence or absence of certain facts in the world, numeric planning allows reasoning over numeric fluents—state variables that take numerical values. These fluents can be modified by actions through numeric effects such as incrementing, decrementing, or assigning values. This enables planning in domains involving resources, numeric locations, or measurable quantities. To generate valid and effective plans, both classical and numeric planning rely on an accurate domain model that describes the actions available to the agent, including their preconditions and effects. In numeric planning, the domain model must also specify how actions modify numeric fluents, often through complex and context-dependent functions. This makes model specification even more difficult, as it requires not only identifying which fluents are affected by each action but also defining the correct numeric transformations. Without a reliable model, the planner may generate plans that are invalid or infeasible when executed in the real world. As a result, there has been increasing interest in approaches that reduce manual modeling effort. However, reducing manual effort alone is not sufficient, since even initially correct models may become outdated as the environment evolves over time. Such changes are referred to as novelties, which can introduce or alter domain objects and actions. The term 'novelty' has also been used in a different context in the planning literature (B. Corrêa and Seipp 2022). They referred to novelty as how different a state is from another and used that to guide the search to more diverse paths in the search space. In this work, novelty refers to a change in the environment, following the terminology of (Boult et al. 2019).

A prior survey on model repair (Bercher, Sreedharan, and Vallati 2025) categorizes model-repair approaches into four constraint types: solvability of planning problems, validity of specific plans, optimality of given plans, and enforcement of desired properties in the solution space. For example, Lin et al. (2025) uses hitting-set computation to identify minimal model changes based on known plan feasibility or infeasibility, which falls under the validity category.

Our work focuses on the solvability constraint category. One algorithm that inspired our approach is Hydra (Mohan et al. 2024), a model-space learning framework that adapts to novelties by heuristically searching over possible modifications to the domain model. Hydra employs a predefined set of Model Manipulation Operators (MMOs) to update domain parameters, such as physical constants (e.g., gravity) or other environment-specific values, that directly influence action dynamics while preserving the overall action structure. These parameter updates can, in turn, reveal new action structures, making Hydra a powerful foundation for novelty adaptation.

Instead of heuristic-based search, our work focuses on learning directly from data. Specifically, by collecting transitions across problem instances, we apply linear regression to infer the new effects of novel actions. This enables our model to recover solvability in domains impacted by such

novelties. However, we encountered a trade-off when selecting which state variables to use for learning the new effects: choosing fewer variables requires fewer transitions to achieve accurate adaptation, while using more variables, or even adding monomials, can help capture more complex novelties but increase the data requirement. To address this, we propose four strategies for selecting the features used in the repair process: (1) **Relevant Variables**, which includes only the state variables affected by the novelty; (2) **All Variables**, which includes the full set of state variables; (3) **All Monomials**, which includes polynomial combinations of state variables; and (4) **Dynamic Selection**, a flexible approach that chooses the best-fitting option among the three based on the observed data. Results show that using the dynamic selection allows the system to learn complex novelties without increasing the amount of data required when adapting to simpler ones.

Background

Numeric planning addresses planning problems involving both discrete (Boolean) and continuous (numeric) variables. A widely used formalism is the Planning Domain Definition Language (PDDL) (Aeronautiques et al. 1998), specifically version 2.1 (Fox and Long 2003), which extends earlier versions to support temporal and numeric fluents. PDDL2.1 allows specification of complex planning models with numeric preconditions and effects, enabling richer representations of real-world domains. The numeric planning domain is typically defined as $D = \langle F, X, A \rangle$, where F is a set of Boolean variables, X numeric variables, and A actions. A state assigns values to all variables in $F \cup X$. Each action $a \in A$ is described by $\langle \text{name}(a), \text{pre}(a), \text{eff}(a) \rangle$, where $\text{pre}(a)$ specifies applicability conditions and $\text{eff}(a)$ the resulting state changes. The set of actions defines the *action model*. A planning problem is then specified by $\Pi = (I, G)$, where I is the initial state and G the goal condition, including both Boolean assignments and numeric constraints. A solution, or *plan*, is a sequence of actions that transforms I into a goal-satisfying state s_G. Problems may also be described in a *lifted* form, using parameterized predicates, functions, and actions for abstract and general representations. Lifted representations are especially useful for scaling to larger problems, since they avoid grounding every possible state.

Action model learning. Manually creating domain models is difficult, slow, and error-prone (McCluskey, Vaquero, and Vallati 2017), limiting scalability in real-world settings. Model learning automatically infers action models from observations, typically execution trajectories, reconstructing preconditions and effects so planning can proceed when dynamics are unknown. It spans symbolic models learned under limited observability (Cresswell and Gregory 2011; Lindsay et al. 2017), to noisy, safe, conditional, or online learning (Lamanna and Serafini 2024; Juba, Le, and Stern 2021; Mordoch et al. 2024; Lamanna et al. 2021). Recent work extends to numeric domains (Segura-Muros, Pérez, and Fernández-Olivares 2021; Mordoch, Juba, and Stern 2023), some integrating reinforcement learning (Sreedharan and Katz 2023; Benyamin et al. 2025), and others even learning from visual inputs (Asai et al. 2022; Xi, Gould, and Thiébaux 2024). A central challenge is balancing generality and accuracy: overly general models risk inconsistency, while overly specific ones may fail to transfer across domains. In addition, scalability remains a persistent issue, as the complexity of learned models grows quickly with domain size. These challenges highlight why repair and adaptation remain complementary to learning.

Model repair, by contrast, assumes access to an initial, possibly flawed, model and refines it according to a set of constraints. Unlike model learning, which infers models from scratch, model repair focuses on correcting errors in an existing domain model to resolve issues preventing desired plans. These modifications typically involve updating actions' preconditions and effects, modifying domain objects, or adding new actions. Model repair can be subject to different constraints (Bercher, Sreedharan, and Vallati 2025): (1) *Solvability*: ensuring a set of problems is solvable. For example, (Gragera et al. 2023) repairs models by adding missing effects but cannot delete effects or alter preconditions. (2) *Plan validity*: requiring that specific plans be solutions or excluded. (Lin et al. 2025) address this with hitting set computation for minimal changes. (3) *Plan optimality*: refining the model so a given plan becomes optimal, e.g., (Grover et al. 2020) reconcile models interactively with experts. (4) *Solution-space properties*: requiring all valid plans satisfy desired characteristics, e.g., environment design (Zhang, Chen, and Parkes 2009) modifies the world to ensure feasible, safe, or distinguishable plans within physical limits.

Our approach falls under (1). We assume a set of problems solvable with a correct model, but where the current model is faulty and yields incorrect plans. We use observed transitions and discrepancies between predicted and actual outcomes to identify faults and apply corrections. This approach is data-driven yet remains compatible with symbolic representations, making it suitable for integration with classical planners.

The Hydra algorithm (Mohan et al. 2024), which motivated our work, is a model-space learning framework that adapts to novelties by performing heuristic-guided search over possible domain modifications. In automated planning, domain knowledge typically consists of *entities* (domain objects) and their relationships, together with *actions* described by preconditions and effects. Hydra explores modifications via Model Manipulation Operators (MMOs) that adjust domain parameters, such as physical constants like gravity, directly influencing action dynamics. While Hydra preserves overall action structure, its parameter updates can yield new structures, making it powerful for some novelties. However, Hydra is limited to novelties affecting domain objects. In contrast, our approach targets novelties directly impacting actions, which also include object changes since these affect actions. Moreover, heuristic methods like Hydra, relying on hand-crafted rules or search-based guesses, may lack precision and generalization under complex or unexpected changes. Our method instead takes a data-driven approach, using observed transitions to adapt models more accurately, reducing inconsistency and improving robustness

in dynamic environments.

Related work

Model learning. NSAM (Mordoch, Juba, and Stern 2023) is the current state-of-the-art algorithm for learning safe numeric action models. Safety in this context refers to the property that ensures any plan generated using the learned model is guaranteed to be executable in the real environment. Integrating our repair method with NSAM could enable selective updates to the domain model, only when such changes are proven to be safe, thereby improving both adaptability and reliability in dynamic settings.

Model drift in Model-Based RL. Model-based reinforcement learning methods can adapt to changing environments by updating their internal models, but effective adaptation depends on mechanisms to detect and manage inconsistencies. One of the few approaches that integrates model-based RL with numeric planning is RAMP (Benyamin et al. 2025), which leverages a planner to guide a deep RL agent's decisions. However, it assumes a stationary environment and doesn't address model drift, which can cause inconsistencies after changes. Our repair mechanism can improve this algorithm by detecting and correcting inaccuracies, keeping the model aligned with the environment, and potentially enhancing the RL agent's performance.

Partially observable environments. In more complex settings, not all state transitions are fully observable. Natan, Stern, and Kalech (2024) address this challenge by interacting with the real environment to identify hidden changes and trace them back to the actions that caused them. Combining their approach with our repair mechanism could allow agents to detect and fix model discrepancies even when parts of the environment are not directly visible.

Problem Definition

The repair model task.[1] Let M denote the agent's current (learned) action model, and let M' represent the true domain, which may contain *novelty* (i.e., changes) in action effects. The agent is given a planning problem $\Pi = (I, G)$, where I is the initial state and G is the goal condition, and it must act within M'.

The task is to iteratively refine M to better approximate M' through interaction. Each algorithm begins by attempting to solve Π using M to generate a plan π. It then executes π in the real domain M' and observes the result. Based on the observed discrepancies, such as action failures or unexpected state transitions, the algorithm updates M using repair strategies, aiming to minimize the differences between M and M' over time.

Assumptions. We assume full observability of the environment: the agent has access to the complete state, including the values of all variables, both before and after each action is executed. The environment is deterministic. Novelties arise only in the numeric effects of actions, which may involve linear or non-linear expressions composed over state variables (e.g., linear combinations or monomial terms).

[1]For a more in-depth theoretical analysis of novelty and its implications on model repair, we recommend referring to the work of Mohan et al. (2024).

Methodology

In the following, we provide an overview of this repair pipeline, as depicted in Figure 1, before detailing the key components of each repair algorithm.

Overview and Pipeline

Our approach repairs the action model through a data-driven pipeline that enables efficient and interpretable updates to the planning model. As illustrated in Figure 1, the pipeline adopts a drill-down approach, starting from the broader behavior of the system and progressively narrowing down to the specific elements causing inconsistencies. The process begins by planning actions to solve the problem and executing these plans in the real environment. After observing the resulting changes, the method identifies discrepancies between expected and actual outcomes. It then isolates the state variables most likely responsible for these deviations and applies regression to update the action model's effects, restoring consistency with observations. This learning process is iterative. As the agent encounters new problems, each observed discrepancy is incorporated into the dataset, and the regression is employed again to continuously refine the action model's effects. In the following sections, we define the fundamental building blocks of each repair algorithm.

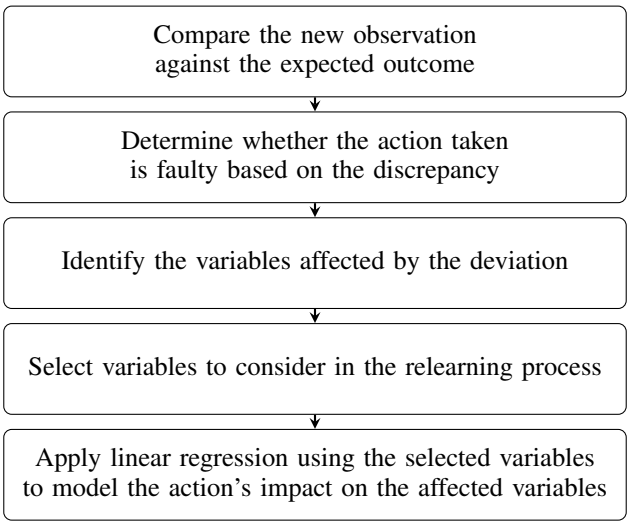


Figure 1: Pipeline for identifying and correcting faulty actions through model repair.

Isolating the novelty

To effectively repair the action model, the first step is to identify which actions yield outcomes that deviate from model predictions (Algorithm 1). This process begins by comparing the observed outcomes in the environment with the predicted outcomes generated by the current model.

Algorithm 1: Novelty Detection

Input: Plan $\pi = [a_1, a_2, \ldots, a_n]$
Input: Current domain model M
Input: Real environment M'
Input: Initial state s
Output: Novelty Dataset $\{\mathcal{D}_a\}_{a \in \mathcal{A}}$
for each $a \in \pi$ **do**
    $s' \leftarrow M'(s, a)$
    $\hat{s}' \leftarrow M(s, a)$
    $\overline{y} \leftarrow \{v \mid v \in \text{Vars}(s'), \hat{s}'[v] \neq s'[v]\}$
    if $\overline{y} \neq \emptyset$ **then**
        $\mathcal{D}_a \leftarrow \mathcal{D}_a \cup \{\langle \overline{y}, s \rangle\}$
    end if
    $s \leftarrow s'$
end for
return $\{\mathcal{D}_a\}_{a \in \mathcal{A}}$

Given our current domain model M, we calculate a plan π that will be executed on the real environment M'. The execution of π yields a list of triplets (s, a, s'), where s is the state before action a, and s' is the resulting state afterward, representing the actual transitions in the environment M'. Using this data, we simulate the expected outcome $\hat{s}'$ of applying a in state s according to model M. If the expected state differs from the observed one, i.e., $\hat{s}' \neq s'$, meaning that at least one state variable has a different value, we conclude that action a produced an unexpected effect.

Let $\overline{y}$ be the set of state variables that differ between the predicted state $\hat{s}'$ and the observed state s', $\overline{y} = \{\, y_i \mid y_i \in \mathcal{Y},\ \hat{s}'(y_i) \neq s'(y_i) \,\}$. In each run, we save data pairs $\langle \overline{y}, s \rangle$ for every action executed in the real environment. These pairs are collected in an active learning manner (as described in this section) and maintained in a list $\mathcal{D}_a$ for each action a. The variables in $\overline{y}$ indicate the components of the model that produced incorrect predictions and are therefore targeted for relearning, guiding the updates during the model repair process.

Repairing the novelty

Using the collected dataset $\mathcal{D}_a$, we fit a repair function F that models how each state variable in $\overline{y}$ should be adjusted based on the observed data. As new data is added to $\mathcal{D}_a$, the function F is reapplied to update the corresponding components of $\overline{y}$, gradually improving their accuracy.

We tested four different repair strategies, each represented by a different choice of F, which varies in how they select the input variables to focus on and how they perform the updates. Specifically, the function F may take as input either variables whose values deviate from the model's predicted effects or all variables appearing in the effect expressions, and apply updates such as linear or polynomial functions. These strategies enable a comparative evaluation of different methods for refining the model and correcting prediction errors.

Different approaches to isolate the variables

Consider an environment $\mathcal{M}'$ with state variables $\{x_1, x_2, x_3\}$. Initially, we have an action a whose effect is defined as:

$$x_1' = x_1 + 1$$

Suppose we introduce a novelty by modifying the semantics of action a to instead be:

$$x_1' = x_1 + x_2 + 1$$

After executing action a in state $s = \{x_1 = 1, x_2 = 1, x_3 = 1\}$, we observe the resulting state $s' = \{x_1 = 3, x_2 = 1, x_3 = 1\}$. According to the original model, we would expect $x_1' = x_1 + 1 = 2$, but we observe $x_1' = 3$. This discrepancy indicates that a novelty has occurred in the semantics of action a.

We denote the relevant variables for the changed effect as $\{x_1\}$ (since it is the variable whose value deviated from the expected), and the full set of variables in the state as $\{x_1, x_2, x_3\}$.

To repair the model of action a, we add the relevant transition, consisting of the variable we aim to learn (x_1') and the original state—to the dataset $\mathcal{D}_a$: $[\{x_1' = 3\}, \{x_1 = 1, x_2 = 1, x_3 = 1\}]$. Using $\mathcal{D}_a$, we attempt to repair the action using four increasingly expressive repair strategies:

Relevant Variables These are the variables whose values are inconsistent with the model's predicted effects of the action; that is, variables whose values differ between the actual and predicted observations following the action. In our example, this will use only x_1 to learn the new effect.

All Variables All variables include *all variables* that appear in the function's expressions, as well as any variables that are not tied to parameters outside those used by the variables in the function. These variables are syntactically present and may influence the outcome of numeric computations, even if their values remain constant. In our example, this allows the model to use the set x_1, x_2, x_3 to learn the new effect.

What about monomials? To better capture nonlinear numeric effects, we consider polynomial feature combinations (monomials) of the variables, such as x_i^2 or $x_i x_j$, rather than restricting ourselves to simple linear terms. The complexity of this representation is controlled by a degree parameter k, which limits the highest degree of monomial considered ($k = 1$ corresponds to the linear case). Later, we fit a linear model over these monomials, allowing the framework to represent nonlinear dependencies while keeping the learning problem linear in the parameters.

Relevant Monomials This concept extends the notion of *Relevant Variables* by fitting different monomial equations, rather than assuming simple linear relationships, to better capture the structure of numeric effects. In our example, allow the model to use the set $\{x_1, x_1^2, \ldots, x_1^k\}$ to learn the new effect.

Domain	Sailing	Expedition	Minecraft-Pogo
Action	*go_x* (=direction)	*move_forward*	*get_log*
Original Effect	Move toward x	Costs 1 supply	Get 1 log, decrease 1 from trees in map
Easy	The actions *go_south_west* and *go_north_east* are switched	Cost 2 supplies	Get 0.5 log, decreases 0.5 from trees in map
Medium	causes a person to drift: $d = d + 0.3 \cdot drift\_factor$	cost $0.15 \cdot$ *sled_capacity* $+1$ supplies	Get $0.05 \cdot$ *trees_in_map* logs, decreases same from trees in map
Hard	causes a person to drift: $d = d + 0.001 \cdot d \cdot drift\_factor + 0.1$	cost $2 \cdot$ *factor*$^2 + 1$ supplies	Get $0.05 \cdot$ *mine_factor*^{2} logs, decreases same from trees in map

Table 1: Sample of our novelties in Sailing, Expedition, and Minecraft-POGO for every novelty type (Easy, Medium, Hard).

All Monomials Similarly, this concept extends the notion of *All Variables* by considering different monomial equations, rather than assuming simple linear relationships. In our example with three variables, we allow the set $\{x_1^{\alpha_1} x_2^{\alpha_2} x_3^{\alpha_3} \mid \alpha_1 + \alpha_2 + \alpha_3 \leq k,\ \alpha_i \geq 0\}$ to represent the space of monomials used to learn the new effect.

Repair Strategy

Given the same scenario as before, we denote four repair strategies—*Relevant Variables*, *All Variables*, *All Monomials*, and *Dynamic Selection*—each of which operates as follows and is illustrated on the repair of the effect for the variable x_1 and $k = 2$:

Relevant Variables Repair Fits a simple linear model using only the affected variable:

$$x_1' = m_1 x_1 + c$$

All Variables Repair Extends the fit to include every state variable appearing in the effect:

$$x_1' = m_1 x_1 + m_2 x_2 + m_3 x_3 + c$$

All Monomials Repair Further extends the feature space to all monomials in $\{x_1, x_2, x_3\}$ up to degree $k = 2$:

$$\begin{aligned} x_1' = {} & m_1 x_1 + m_2 x_2 + m_3 x_3 \\ & + m_{12}\, x_1 x_2 + m_{13}\, x_1 x_3 + m_{23}\, x_2 x_3 \\ & + m_{11}\, x_1^2 + m_{22}\, x_2^2 + m_{33}\, x_3^2 + c \end{aligned}$$

Dynamic Selection Repair Collects three candidate models (Relevant Variables, All Variables, All Monomials up to degree k) and evaluates each on the data set $\mathcal{D}_a$. The variant with the highest R^2 is chosen, with ties broken in favor of the model using fewer features.

The proposed algorithms enable the agent to detect novelties and iteratively fit an updated numeric-effect function for action a using online data collected during interaction with the environment.

Dynamic Selection

This approach dynamically selects the best model variant from a given set of candidates by maximizing the fit metric R^2, which measures how well the model explains the observed data, with 1.0 indicating a perfect fit. At the same time, it aims to minimize complexity by preferring models with fewer variables when multiple candidates achieve similar fit.

Given a degree parameter k, the candidate set will include linear models based on Relevant Variables, All Variables, or All Monomials up to degree k. For example, when dealing with simple novelties that only involve linear effects, if a model using fewer variables achieves a perfect or near-perfect fit, Dynamic Selection will prefer this simpler linear model over more complex polynomial alternatives.

Expressiveness and Completeness As the degree parameter k increases, the model's expressive power grows, enabling it to approximate polynomial numeric effects with increasing accuracy. By the Weierstrass Approximation Theorem (Weierstrass 1885), as $k \to \infty$, the model can approximate any continuous function on a closed interval arbitrarily well given sufficient data.[2]

Proof: Assume our model includes all monomials $1, x, x^2, \ldots, x^k$. Any polynomial of degree $m \leq k$ can then be written exactly as $p(x) = a_0 + a_1 x + \cdots + a_m x^m$ by choosing appropriate coefficients. Moreover, for any continuous function f on $[a, b]$ and any $\varepsilon > 0$, there exists a polynomial p such that $|f(x) - p(x)| < \varepsilon$ for all $x \in [a, b]$. Therefore, as $k \to \infty$ and with sufficient data to estimate coefficients, the model can exactly represent any polynomial effect and approximate any continuous effect arbitrarily well.

Experimental Results

We conducted experiments on four domains: **Expedition**, **Sailing**, **Drone**, and **Minecraft-Pogo**. The first three domains are from the International Planning Competition (IPC) 2023 (Taitler et al. 2024). The last domain (**Minecraft-Pogo**) is a recently introduced numeric planning in which an agent operates in the popular Minecraft game, tasked with crafting a Wooden Pogo stick Benyamin et al. (2024).[3] For each domain, we developed a custom problem generator to create random instances of each domain.

[2]This theorem was later generalized by Stone (1937) to a wider class of topological spaces and function algebras

[3]Specifically, we used the fully numeric version of the environment, referred to as the "Counting" modeling of the problem. For more details, see Benyamin et al. (2023).

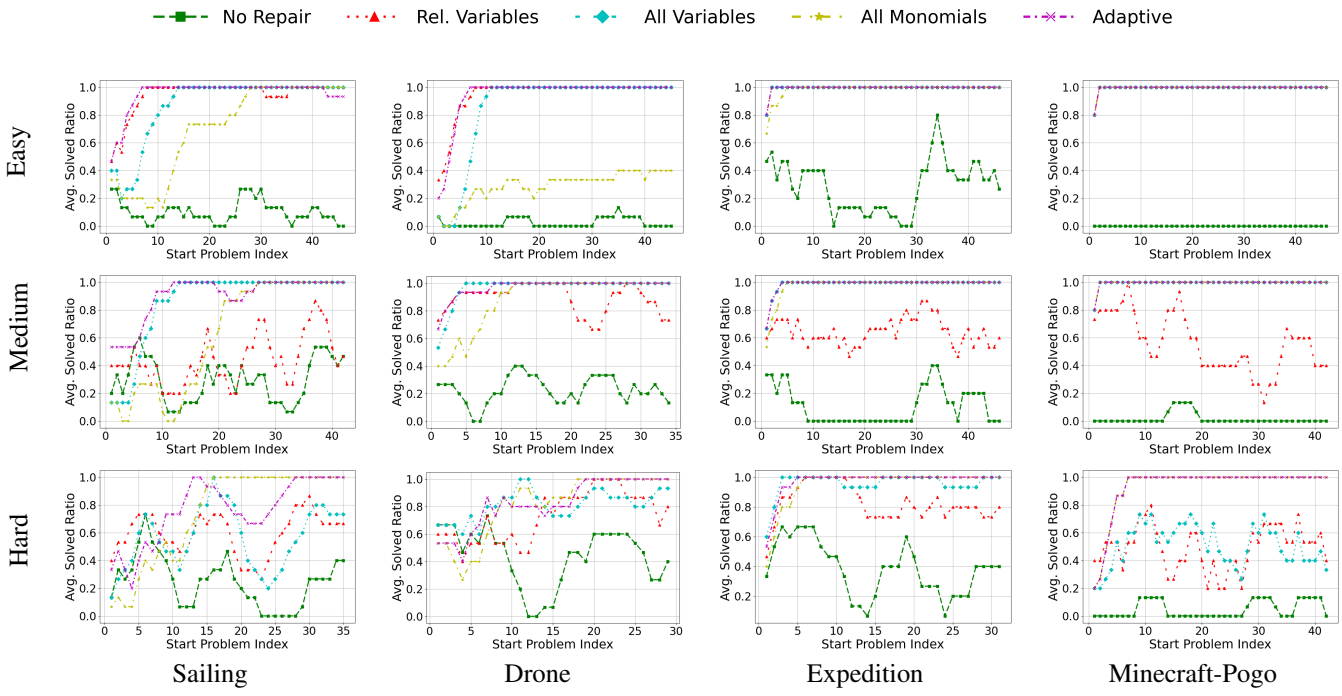


Figure 2: Results for Easy, Medium, and Hard novelty sets across all domains.

These domains were selected to demonstrate the robustness and versatility of our approach, including planning domains from established numeric planning benchmarks and recently introduced, less-studied planning problems.

Novelties Each domain includes three sets of novelties, labeled *Easy*, *Medium*, and *Hard*. These sets differ by the magnitude of environment change the novelty introduces. Each set contains three novelties, for a total of nine novelties per domain. Examples of these novelties are listed in Table 1, and a complete list is given in the supplementary material.

Baseline and Competing Algorithms We evaluated all the repair algorithms described above, namely Relevant Variables, All Variables, All Monomials, and Dynamic Selection. We also evaluated using the original model without any mechanism to detect or respond to novelty. This is referred to as *No-Repair*.

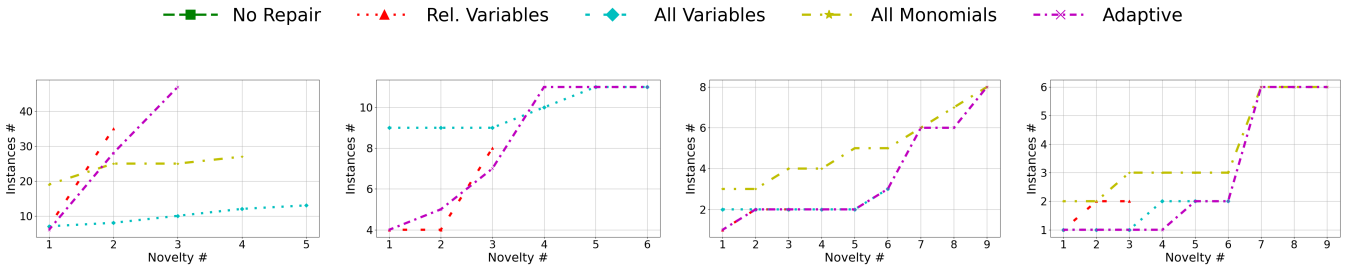
Implementation Details For planning, we used Nyx, a Python-based domain-independent PDDL+ planner (Piotrowski and Perez 2024). We configured Nyx to solve problems in each domain as follows. For Sailing, Drone, and Expedition, we used a custom, domain-specific heuristic and Greedy Best-First Search (GBFS). For Minecraft-Pogo, we used a simple Breadth-First Search (BFS). A time limit of 60 seconds was imposed on the planner, after which we declared a failure. The Nyx planner is deterministic, and therefore, we performed a single run per algorithm, domain, novelty, and problem. All experiments were conducted on a Windows 11 machine with Processor: 11th Gen Intel® Core™ i7-1165G7 @ 2.80GHz, 4 cores, 8 logical processors, and 32GB RAM.

Experimental Design For each domain, we generated 50 problems. Each pair of domain and novelty difficulty were tested on those 50 set, though for every pair we excluded from their set all problem indices where one of the novelties from that difficulty rendered the problem unsolvable. Then, for every repair algorithm we iterate over these problems, and for each problem perform the following steps: (1) run our planner with the current domain model to try to solve the current problem, (2) record the outcome — success or failure (planner timeout is included as failure), and (3) apply the evaluated repair algorithm to repair the current domain model based on the collected observations. For every such experiment, we measured which problems were solved successfully and after how many problems the repair algorithm was able to correctly repair the domain model, if it was able to do so.

Results

Figure 2 shows our results across all domains and novelty types, arranged in a 3x4 grid of plots. Each column corresponds to a domain, and each row corresponds to a novelty type.

In each plot, the x-axis represents the problem index, ranging from 1 to 50 (excluding problems rendered unsolvable by any novelty within that type). For each combination of a problem, a novelty, and a repair algorithm, the outcome is recorded as a binary value: 1 for success and 0 for fail-



(a) Sailing (b) Drone (c) Expedition (d) Minecraft-Pogo

Figure 3: Number of problem instances required to fully learn the novelty for each repair strategy.

ure. The y-axis displays the success rate based on these outcomes, calculated as a moving average with a window size of 5 and averaged over the three specific novelties within that type. Thus, each plot illustrates the performance of each repair algorithm as the system encounters an increasingly diverse set of problem instances.

On the easy problem set , most repair methods resumed solving effectively after a single failed instance. The *Adaptive* and *Rel. Variables* strategies maintained a consistent trajectory close to perfect, across all domains. In the Sailing and Drone domains , the *All Monomials* and *All Variables* strategies required more problems to reach full success. In the Drone domain specifically , the presence of novelties rendered many actions inapplicable, leading to reduced data collection and slower adaptation for several methods. For the medium novelty set , the *Rel. Variables* strategy shows limited adaptation across domains, however has more average success than no repair. The *All Monomials* strategy progresses in success rate slowly in the Drone and Sailing domains. The *Adaptive* and *Rel. Variables* strategies maintain a high performance across all domains, remaining close to a perfect success rate. In the hard novelty set , the *All Monomials* and *Adaptive* strategies exhibit the highest performance across all domains, converging to an average success rate of 1 after enough problem instances. The *All Variables* and *Rel. Variables* methods show unstable average success rates. In the early part of the problem index range, *All Variables* and *Rel. Variables* slightly outperform *Adaptive* in the Sailing, Drone, and Expedition domains. However, this advantage diminishes over time as the number of solved problems increases.

In the second evaluation (Figure 3), we present a cactus plot for each domain, where the horizontal axis shows the number of novelty instances each repair strategy learned, and the vertical axis shows the number of problems required to learn each novelty. Better performance is indicated by points farther to the right (more novelties) and lower (fewer problems).

We observe that the *Adaptive* strategy learned the novelty in 27 of 36 instances across all domains, typically requiring no more than 11 problems per instance. However, in Sailing, the *All Variables* and *All Monomials* strategies adapted faster, learning more novelties with fewer problems than *Adaptive*. Nevertheless, the other strategies were less effective: *All Monomials* learned 22 instances (none in Drone), *All Variables* 23, and *Rel. Variables* only 11.

It is important to note that learning a novelty does not always correlate with consistently solving problems. In many cases, a strategy solved problems before fully learning the novelty by acquiring an incomplete repair. In some instances, a strategy may have avoided an action due to an ineffective repair that made it less favorable during search, or simply because the action was not required.

Overall, the *Adaptive* method shows the most consistent adaptation across all novelty sets and domains. It performs comparably to the best repair in each case, combining the strengths of individual strategies.

The *Adaptive* method adapts quickly to simple novelties with relatively few observations and scales effectively to more complex changes. Over longer horizons, the *Adaptive* converges to a perfect success rate, even in domains where others plateau. In terms of novelty learning, the *Adaptive* demonstrates superior performance, learning 27 of 36 novelties while requiring relatively few problems. As shown in Figure 3, it achieves broader generalization and faster adaptation compared to other strategies, which either learned fewer novelties or needed more problems.

Conclusion and Future Work

In this work, we address the challenge of adapting to environmental novelties in numeric planning domains by detecting run-time failures from model–environment mismatches and applying a learning-based repair mechanism to update numeric components. This enables agents to maintain effectiveness without fully identifying the underlying changes. Across multiple domains and novelty levels, the *Adaptive* method consistently outperformed competing strategies—recovering almost immediately for easy novelties and steadily improving for harder ones—demonstrating the value of combining multiple repair strategies for rapid recovery and sustained adaptation.

References

Aeronautiques, C.; Howe, A.; Knoblock, C.; McDermott, I. D.; Ram, A.; Veloso, M.; Weld, D.; SRI, D. W.; Barrett, A.; Christianson, D.; et al. 1998. PDDL— The Planning Domain Definition Language. *Technical Report, Tech. Rep.*

Asai, M.; Kajino, H.; Fukunaga, A.; and Muise, C. 2022. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74: 1599–1686.

B. Corrêa, A.; and Seipp, J. 2022. Best-First Width Search for Lifted Classical Planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 32(1): 11–15.

Benyamin, Y.; Mordoch, A.; Shperberg, S.; Piotrowski, W.; and Stern, R. 2024. Crafting a pogo stick in minecraft with heuristic search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 261–262.

Benyamin, Y.; Mordoch, A.; Shperberg, S. S.; and Stern, R. 2023. Model Learning to Solve Minecraft Tasks. In *PRL Workshop Series {\textendash} Bridging the Gap Between AI Planning and Reinforcement Learning*.

Benyamin, Y.; Mordoch, A.; Shperberg, S. S.; and Stern, R. 2025. Integrating Reinforcement Learning, Action Model Learning, and Numeric Planning for Tackling Complex Tasks. *arXiv preprint arXiv:2502.13006*.

Bercher, P.; Sreedharan, S.; and Vallati, M. 2025. A Survey on Model Repair in AI Planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Boult, T. E.; Cruz, S.; Dhamija, A.; Gunther, M.; Henrydoss, J.; and Scheirer, W. 2019. Learning and the Unknown: Surveying Steps toward Open World Recognition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01): 9801–9807.

Cresswell, S.; and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 42–49.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Gragera, A.; Fuentetaja, R.; García-Olaya, Á.; and Fernández, F. 2023. A planning approach to repair domains with incomplete action effects. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, 153–161.

Grover, S.; Sengupta, S.; Chakraborti, T.; Mishra, A. P.; and Kambhampati, S. 2020. RADAR: automated task planning for proactive decision support. *Human–Computer Interaction*, 35(5-6): 387–412.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 379–389.

Lamanna, L.; Saetti, A.; Serafini, L.; Gerevini, A.; and Traverso, P. 2021. Online Learning of Action Models for PDDL Planning. In *IJCAI*, 4112–4118.

Lamanna, L.; and Serafini, L. 2024. Action Model Learning from Noisy Traces: a Probabilistic Approach. In *ICAPS*, 342–350.

Lin, S.; Grastien, A.; Shome, R.; and Bercher, P. 2025. Told You That Will Not Work: Optimal Corrections to Planning Domains Using Counter-Example Plans. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(25): 26596–26604.

Lindsay, A.; Read, J.; Ferreira, J. F.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning Models from Natural Language Action Descriptions. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In *Proceedings of the Knowledge Capture Conference, K-CAP*, 14:1–14:8.

Mohan, S.; Piotrowski, W.; Stern, R.; Grover, S.; Kim, S.; Le, J.; Sher, Y.; and de Kleer, J. 2024. A domain-independent agent architecture for adaptive operation in evolving open worlds. *Artificial Intelligence*, 334: 104161.

Mordoch, A.; Juba, B.; and Stern, R. 2023. Learning Safe Numeric Action Models. In *AAAI*, 12079–12086. AAAI Press.

Mordoch, A.; Scala, E.; Stern, R.; and Juba, B. 2024. Safe learning of pddl domains with conditional effects. In *International Conference on Automated Planning and Scheduling*, volume 34, 387–395.

Natan, A.; Stern, R.; and Kalech, M. 2024. Diagnosing Non-Intermittent Anomalies in Reinforcement Learning Policy Executions (Short Paper). In *35th International Conference on Principles of Diagnosis and Resilient Systems (DX 2024)*, 23–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Piotrowski, W.; and Perez, A. 2024. Real-World Planning with PDDL+ and Beyond. *arXiv preprint arXiv:2402.11901*.

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2021. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 1–17.

Sreedharan, S.; and Katz, M. 2023. Optimistic exploration in reinforcement learning using symbolic model estimates. *Advances in Neural Information Processing Systems*, 36: 34519–34535.

Stone, M. H. 1937. Applications of the theory of Boolean rings to general topology. *Transactions of the American Mathematical Society*, 41(3): 375–481.

Taitler, A.; Alford, R.; Espasa, J.; Behnke, G.; Fišer, D.; Gimelfarb, M.; Pommerening, F.; Sanner, S.; Scala, E.; Schreiber, D.; et al. 2024. The 2023 International Planning Competition.

Weierstrass, K. 1885. Uber die Analytische Darstellbarkeit Sogenannter Willkrlicher Functionen Einer Reellen Vernderlichen, sitzungsber, Akad.

Xi, K.; Gould, S.; and Thiébaux, S. 2024. Neuro-Symbolic Learning of Lifted Action Models from Visual Traces. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, 653–662.

Zhang, H.; Chen, Y.; and Parkes, D. 2009. A general approach to environment design with one agent. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2002–2008. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.