FELIX: Model Repair in Numeric Planning

Nir Aharoni
Ben-Gurion University of the Negev, Beer Sheva, Israel

Yarin Benyamin
Ben-Gurion University of the Negev, Beer Sheva, Israel

Shiwali Mohan
Future Concepts Division, SRI International, Menlo Park, CA, USA

Roni Stern
Ben-Gurion University of the Negev, Israel

Abstract

Automated planning algorithms rely on having a sufficiently accurate domain model to solve planning problems, particularly in complex environments with numeric effects. In real-world settings, however, domain models can become misaligned with the environment over time, i.e., no longer accurately reflect the environment. Even a minor drift in numeric effects can lead to faulty plans and execution failures. In this work, we address the problem of repairing numeric planning domain models by proposing FELIX, a learning-based repair mechanism that restores planning effectiveness by identifying and correcting outdated numeric effects. Rather than re-learning the full environment dynamics from scratch, our approach updates only the affected components of the domain model, enabling efficient adaptation with limited data. FELIX restores planning performance across diverse numeric domains, even under complex model drift. It uniquely handles both nonlinear effect shifts and structural signature changes while maintaining polynomial repair overhead under bounded parameter-type assumptions.

2012 ACM Subject Classification Computing methodologies → Knowledge representation and reasoning

Keywords and phrases Automated diagnosis, learning action models, model repair

Digital Object Identifier 10.4230/OASIcs.DX.2026.5

Funding *Roni Stern*: Israel Science Foundation (ISF) #1238/23, Israel Innovation Authority #90011.

1 Introduction

Automated planning is a fundamental challenge in Artificial Intelligence research in which an agent seeks a sequence of actions that transforms an initial state into a goal state. In classical planning, a state 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 numeric values. For example, instead of representing the quantity of a resource using multiple Boolean predicates (e.g., `has-item-1`, `has-item-2`, `has-item-3`), numeric planning can use a single fluent such as `(item-count)` to represent the total number directly. 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 update numeric fluents, typically through parameterized numeric

expressions. This increases modeling difficulty, as it requires defining not only which fluents are affected by each action, but also the exact numeric transformations applied to them. 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 [14, 1, 8] even for numeric action models [23, 27].

However, modeling is rarely a one-time activity. Even models that are initially very accurate may become outdated over time when the environment changes, e.g., when new domain objects are introduced, or action effects are altered. This phenomenon is known as *model drift* in various contexts. Model drift necessitates *model repair* to maintain planning reliability. Most prior work on model repair addresses only classical planning; see [7] for a survey.

To the best of our knowledge, the only prior work on model repair for numeric planning is Hydra [21]. Hydra is an algorithmic framework for a planning-based agent that is designed to detect and adapt to model-environment mismatches. Hydra maintains an internal environment model, and if it detects model inaccuracies, it heuristically searches the space of possible model repairs. Hydra employs a predefined set of model transformation operators, called Model Manipulation Operators (MMOs), each representing a predefined modification to the domain model. For example, an MMO can be to increment or decrement by a fixed amount the value of a specific numeric constant, such as gravity or friction. Hydra repairs numeric models by performing a heuristic search over combinations of these MMOs. Thus, Hydra's runtime grows exponentially with the number of MMOs that need to be applied. In addition, Hydra requires an external agent – a human operator – to manually specify the MMOs.

In this work, we focus on a specific type of model repair problem: changes in the numeric effects. For this problem, we propose FELIX (Find, Estimate, Learn, Improve, eXecute), a workflow for automatically detecting and repairing numeric effects from observed transitions. By collecting execution data across problem instances and fitting regression models to detected discrepancies, FELIX repairs drifted numeric effects without requiring manually engineered operators or combinatorial search.

Moreover, we aim to learn changes in action signatures rather than assuming they are fixed. A notable algorithm that addresses this challenge is called L1 [3]. While our work focuses on numeric changes, L1 focuses on predicates. The authors of L1 prove that inferring missing action parameters in a logical domain is a combinatorial problem at least as hard as graph isomorphism. Consequently, L1 utilizes SAT encodings to "imagine" missing parameters and induce consistent parameter bindings globally - a process with exponential worst-case search time. In contrast, our approach distinguishes between two types of missing parameters. The first type is identified through discrepancies between the learned model and the environment, while the second is discovered via a systematic search over the different numeric parameter types present in the environment. Since each candidate is considered at most once, the discovery process runs in polynomial time.

Our analysis revealed a trade-off between the dimensionality of state representations and repair efficiency: smaller feature sets enable rapid adaptation from limited transition data, whereas richer representations incorporating higher-degree monomials are necessary to capture complex model–environment deviations. To address this trade-off, we employ multiple isolation strategies that vary in expressive power and allow the repair process to progressively increase representation complexity when needed. Empirical results demonstrate that this adaptive repair strategy captures complex forms of model drift while maintaining data efficiency when the drift is simple. Furthermore, we show that the approach automatically learns changes in action signatures, thereby enabling sustained planning performance in the presence of environmental drift.

2 Background

Numeric planning. addresses planning problems involving both discrete (Boolean) and continuous (numeric) variables. A widely used formalism to represent numeric planning problems is Planning Domain Definition Language (PDDL) 2.1 [10]. A numeric planning problem in PDDL2.1 is defined by a *domain* D and a *problem instance* Π . A domain is a tuple $D = \langle F, X, A \rangle$, where F is a set of Boolean variables, X is a set of numeric variables, and A is a set of actions. A state assigns values to all variables in $F \cup X$. Each action $a \in A$ is described by $\langle \text{name}(a), \text{pre}(a), \text{eff}(a) \rangle$. $\text{pre}(a)$ is the action's preconditions, which are Boolean assignments and numeric constraints that must hold in state s for a to be applicable. $\text{eff}(a)$ is the action's effects, which define the state transition that occurs when applying a. This set of preconditions and effects of all actions is referred to as the *action model*. A planning problem instance is specified by $\Pi = \langle I, G \rangle$, where I is the initial state and G is the goal condition. A solution, or *plan*, is a sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ that transforms I into a goal-satisfying state s_G. To improve scalability and abstraction, domains are often represented in a *lifted* form. A lifted action $a(\mathcal{V})$ is a schema where $\mathcal{V} = \{v_1, \ldots, v_k\}$ is a set of parameters. A grounded action is produced by a substitution $\theta : \mathcal{V} \to O$, mapping parameters to specific domain objects O. The same definition applies to a lifted fluent $f(\mathcal{V})$ and its grounded counterpart. Lifted representations are essential for scaling to larger problems, as they allow for generalized planning without requiring the explicit grounding of the entire state space.

Action model learning. Manually creating domain models is difficult, slow, and error-prone [20], limiting scalability in real-world settings. Action model learning algorithms infer action models from observations, which are typically execution *trajectories*. The literature on action model learning is broad, covering approaches that learn symbolic models under limited observability [9, 19, 3], as well as methods designed to handle noise [17], safety constraints [15], conditional effects [25], online learning settings [16], and multi-agent settings [24]. Recent work on action model learning extends to numeric domains [27, 23, 22], integrates with Reinforcement Learning [28, 6], and even allows learning from visual inputs [2, 31, 32]. A less studied challenge of learning an action model is to learn the actions' signature, i.e., their parameters. One of the few algorithms able to do so is L1 [3], which learns classical planning action models from traces without action parameters. [3] demonstrated that inferring these missing parameters is a combinatorial problem at least as hard as graph isomorphism. L1 utilizes SAT-encoding to "imagine" the missing action parameters required to make the planning model logically consistent, effectively outsourcing the combinatorial search and backtracking to the SAT solver.

Planning and Execution Monitoring. In model-based autonomous agents, generating an optimal or satisficing plan is only half the challenge; ensuring its robust delivery in dynamic or partially observable environments requires continuous plan execution monitoring. Early formalizations by [11] established the classic three-tiered monitoring pipeline: discrepancy detection (identifying variances between the model's predicted state transitions and real-world sensor data), state estimation/diagnosis (determining the root cause of the anomaly), and plan evaluation (determining whether the encountered discrepancy actively violates the validity or optimality of the remaining plan suffix). An example of an architecture that implements this classical execution-monitoring pipeline is Hydra [21], which simulates expected trajectories to detect runtime anomalies.

In our solution, we build upon this execution-monitoring paradigm through the FELIX workflow, but fundamentally change how the agent leverages the initial monitoring phase in order to avoid exhaustive model-space search. Rather than relying on human-engineered search operators like Hydra, FELIX employs discrepancy detection as an informative data-filtering mechanism to decide precisely when to repair and how. At runtime, the agent executes planned actions and utilizes a simulator to generate expected next states. The discrepancy detection module triggers a repair iteration when an observed state variable deviates from predictions beyond a numerical tolerance parameter ϵ (i.e., $|\hat{s}'(y_i) - s'(y_i)| > \epsilon$). Crucially, the monitor determines how the repair should proceed by analyzing the exact subset of deviating state variables $\overline{y}$, lifting and filtering them to isolate only the broken components of the domain model for targeted polynomial regression and signature repair.

Model repair. A model repair algorithm accepts an initial, possibly flawed, domain model, and its task is to modify it according to some set of constraints. This modification may involve updating actions' preconditions and effects, adding or removing actions, or changing their signature. A recent survey on model repair [7] categorizes existing approaches to model repair according to the constraints they enforce: (1) *Solvability*: Repair the domain model so that a valid solution can be found. For example, [12] repairs models by adding missing effects but cannot delete effects or alter preconditions; (2) *Plan validity*: Repair the domain model to make specific plans valid or invalid as solutions. [18] addresses this through hitting set computation for minimal changes; (3) *Plan optimality*: Repair the domain model so that a specified plan becomes optimal; for instance, [13] reconciles models interactively with experts to align costs and optimality; and (4) *Solution-space properties*: Enforcing rules or constraints that every valid solution must satisfy. Environment design [33], for example, modifies the world to ensure all feasible plans are safe or distinguishable within physical limits. Our work falls under the category of Solvability. We assume a set of problems is solvable with a correct domain model, but the current domain model is no longer accurate, leading to execution failures.

Most prior work on this type of model repair has focused exclusively on classical planning. The only exception we are aware of is Hydra [21], a model-space learning framework that adapts to model drift by performing a heuristic-guided search over possible domain modifications to realign the agent's internal model with the environment. Hydra explores modifications via a predefined set of model transformation operators called Model Manipulation Operators (MMOs), each representing a predefined modification to the domain model.

3 Problem Setting and Methodology

We assume a planning-based agent operating in an environment that can be sufficiently modeled as a numeric planning domain $\langle F, X, A \rangle$ with an action model M^*. We assume the environment is deterministic. The agent does not have access to the real action model M^*. Instead, it has access to an action model M that may be different from M^*. The task is to repair M such that it sufficiently approximates the true domain M^* to allow solving planning problems in the domain.

Assumptions

The differences between the agent's model M and the real action model M^* are only in the effects alone, or in a combination of both the action signature and the effects. That is, the action preconditions in M and M^* are the same. Action effects in both M and M^* are

limited to polynomial combinations of the numeric state variables. We assume the agent can always observe its current state, including the values of all variables, both before and after each action is executed.

3.1 The FELIX Workflow

To be able to repair M, the agent must have some information about the (modified) environment. In our work, the agent collects such information by performing actions in the environment and observing their outcome. Specifically, we consider an agent that follows a workflow that we call FELIX (Find, Estimate, Learn, Improve, eXecute). FELIX is an iterative model-repair workflow. Throughout its execution, it maintains an incumbent action model M. Every iteration in FELIX begins with a planning problem $\Pi = \langle I, G \rangle$. Then, FELIX calls a domain-independent planner to synthesize a plan for the given problem using the current domain model, denoted by M. Once a plan is generated, the agent performs every planned action a in the real environment, recording the states visited immediately before and after every action. If an action is inapplicable, it is skipped. Note that the first action is always applicable, as we assume preconditions remain unchanged and only consider novelties in the effects. For every executed action a, FELIX also calls a PDDL simulator to generate the expected next state according to the (possibly incorrect) incumbent action model.

The observed next state s' and the expected next state $\hat{s'}$ are compared, and discrepancies between them are recorded. These discrepancies are diagnosed and passed to a model repair module that modifies the incumbent action model to reconcile these discrepancies. Our primary innovation lies in these diagnosis and repair modules. The complete FELIX workflow is illustrated in Figure 1. This workflow repeats across multiple problem instances, with repaired model components and collected transition data retained for subsequent iterations. Over time, these repairs allow the domain model to more accurately reflect the true behavior of the environment. Next, we describe in detail the key components in the FELIX workflow: diagnosis and repair.

3.2 Diagnosing Model Drift

To effectively repair the action model, the first step is to identify which actions yield outcomes that deviate from model predictions (Algorithm 1). This is done using existing methods for detecting discrepancies between the predicted and observed outcomes of actions [21]. Given an observed transition (s, a, s') obtained from executing the plan in the environment, we compare the predicted successor state $\hat{s}'$ with the observed successor state s'. If the deviation exceeds a numerical tolerance parameter ϵ, i.e., if there exists a state variable y_i such that $|\hat{s}'(y_i) - s'(y_i)| > \epsilon$, we mark action a as inconsistent with the current model.

Next, our method identifies the exact subset of state variables for which the environmental transition deviates from the model prediction. Let $\overline{y}$ be the set of state variables that differ between the predicted state $\hat{s}'$ and the observed state s', i.e., $\overline{y} = \{\, y_i \mid |\hat{s}'(y_i) - s'(y_i)| > \epsilon \,\}$, where y_i ranges over all variables in the domain model. We filter the set of deviating state variables $\overline{y}$ to focus only on those that could be affected by the executed action a. Let $\mathit{params}(a)$ denote the set of parameters of action a, and let x_i be a grounded numeric fluent in the state. We define $\overline{x} = \{\, x_i \mid \mathit{params}(x_i) \subseteq \mathit{params}(a) \,\}$, so that $\overline{x}$ contains only those variables whose parameters are consistent with the action's signature and therefore could plausibly be used by a. To keep our dataset consistent, we convert $\langle \overline{y}, \overline{x} \rangle$ to its lifted form by replacing the grounded parameters $\overline{a}$ with the corresponding lifted parameters of the action.

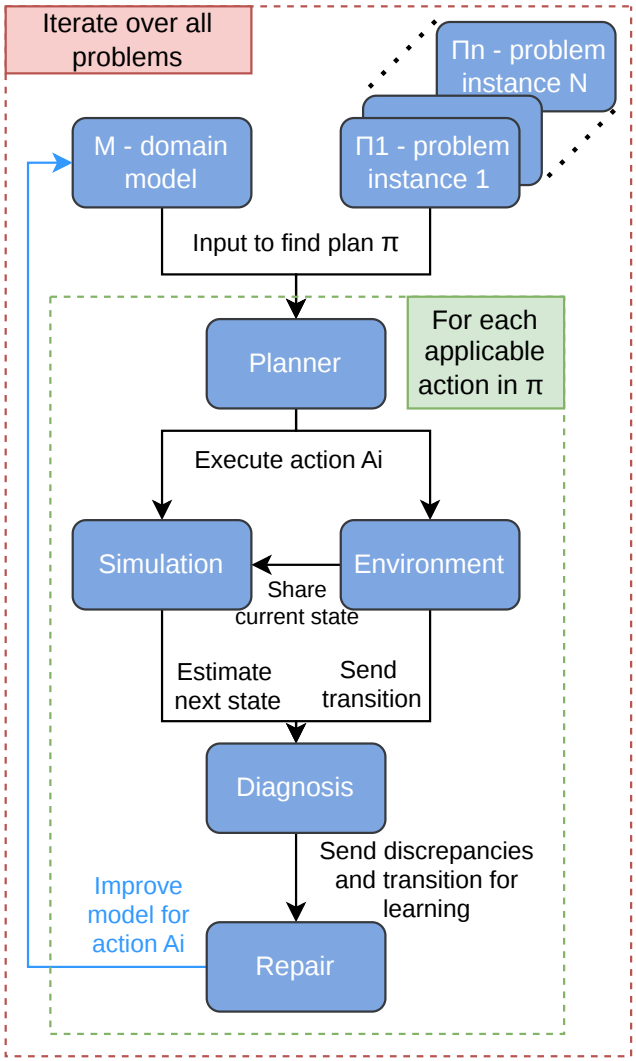

Figure 1 Design of FELIX.

Let $\langle \hat{y}, \hat{x} \rangle$ be the lifted version of $\langle \overline{y}, \overline{x} \rangle$. In each run, we save data pairs $\langle \hat{y}, \hat{x} \rangle$ 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 lifted action $a \in \mathcal{A}$. The variables in $\hat{y}$ indicate the components of the model that produced incorrect predictions and are therefore targeted for relearning, guiding the updates during the model repair process. We illustrate this with an example in the following section.

Variable Isolation Approaches

Model drift may affect variables in ways that range from simple linear adjustments to single parameters to highly nonlinear transformations involving multiple interacting state variables. In such cases, standard linear regression is insufficient to isolate or identify the underlying numeric effects in $\hat{y}$, because the relationships between variables are no longer linear.

Algorithm 1 Resolving Model Drift.

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

To address this, we propose methods to manipulate the state variables. These methods vary in the total number of variables they handle and in the complexity of the model drift they can resolve. To understand the methods, 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: $\{x_1 \leftarrow x_1 + 1\}$. Suppose we introduce a novelty by modifying the semantics of action a to instead be: $\{x_1 \leftarrow x_1 + x_2 + 1\}$.

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 $x_1 = 2$, but we observe $x_1 = 3$. This discrepancy indicates that a novelty has occurred in the semantics of action a.

We then repair the action's effect by using linear regression to predict x_1 from selected variables in the originating state s. In particular, we distinguish between *Changed Variables*, representing the set $\overline{y}$, and *All Variables*, representing the full set of state variables. We denote the changed-variable set for a given effect as $\{x_1\}$ and the complete state-variable set as $\{x_1, x_2, x_3\}$.

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 the model 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. We define *All Monomials of Degree* $\leq k$ as the set of all monomial terms whose total degree does not exceed k. This generalizes the *All Variables* setting (corresponding to $k = 1$), which includes only linear terms, by additionally incorporating higher-order and interaction terms. 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.

3.3 Repairing the Domain Model

3.3.1 Repair Strategy

Given the same scenario as described previously, we consider three repair strategies: *All Monomials*, *Adaptive Selection*, and *Adaptive Selection with Signature Repair*. Each strategy estimates the new numeric effects given a different set of rules.

All Monomials Repair

This strategy expands the state variables to include all monomials over the set of state variables up to degree k. In the running example with variables $\{x_1, x_2, x_3\}$ and $k = 2$, the learned regression model takes the form:

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

Adaptive Selection Repair

This strategy 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, with ties broken in favor of the model using fewer features. Given a degree parameter k, the candidate set will include linear models based on Changed Variables, All Variables, and All Monomials up to degree k. For example, when the true environmental dynamics are linear, if a model using a smaller subset of variables achieves a near-perfect fit, Adaptive Selection prefers this simpler representation over more complex polynomial alternatives, thereby reducing overfitting and improving interpretability.

3.3.2 Repairing the Action Signature

As discussed in 3.2, we filter out grounded fluents that contain parameters that do not appear in the action's current signature, as they cannot be expressed within that signature. If an environment change introduces a new parameter within the effects, the true effect of the action cannot be learned using the existing action signature. Consequently, we employ a repair mechanism to identify and integrate these missing parameters into the action signature.

We distinguish between two types of model drifts that introduce new parameters, each of which requires a different handling strategy. To illustrate these, consider a base action increase(?c - counter) with the effect: $g(?c) \leftarrow g(?c) + 1$, where g is a fluent that receives a counter.

Type 1: Affected

The first type of model drift adds an effect that changes the value of a fluent involving a new parameter. For example, consider the model drift: increase(?c - counter ?a - modifier) $\Rightarrow$ $\{g(?c) \leftarrow g(?c) + h(?a),\ h(?a) \leftarrow h(?a) + 1\}$ In this scenario, the model drift introduces a parameter a, utilizes it within the fluent h (which receives a modifier), and the value of $h(?a)$ is **affected**. Following the execution of this action with grounded constants c and a, where a was selected by the environment without the agent's knowledge, we receive the observation $\overline{y} = \{g(c), h(a)\}$, as defined in Section 3.2. Using $\overline{y}$, we identify whether missing

parameters exist in our signature (see Algorithm 2). Our repair strategy iterates through all parameters in $\overline{y}$ (in this example, $\{a, c\}$). For any parameters not currently present in the action signature, their types are automatically added; the specific type is inferred from the fluent definition (in this example, $?a$ – modifier is inferred from h and is appended to the action signature).

Algorithm 2 Type 1 missing parameter repair.

Input: Parameters of grounded action a_g - $\mathcal{P}_{a_g}$
Input: Different functions - $\overline{y}$ ▷ As described in 3.2
$\mathcal{P}_{used} \leftarrow$ ExtractParameters($\overline{y}$) ▷ Extract parameters from functions
for each $p \in \mathcal{P}_{used}$ **do**
  if $p \notin \mathcal{P}_{a_g}$ **then**
    $\mathcal{P}_a \leftarrow \mathcal{P}_a \cup \{type(p)\}$ ▷ Add new parameter type to the action signature
  end if
end for
return *model*

Type 2: Affects

The second type uses a fluent containing a new parameter to modify the value of an existing fluent. For example, the model drift increase(?c - counter ?a - modifier) $\Rightarrow g(?c) \leftarrow g(?c) + h(?a)$. Here, the parameter ?a is required to evaluate $h(?a)$, which in turn **affects** $g(?c)$. If the current repair model does not yet account for parameters of the type "modifier", it will initially fail to create an accurate model for $g(?c)$.

Solving type 2

Unlike type 1, in type 2, the new parameter is not affected and only **affects** the deviating fluent. As a result, the discrepancy is observed in the affected fluent (in our example, $g(?c)$), while the underlying cause lies in the missing dependency on the fluent with the new parameter (in our example $h(?a)$). How can we repair it (see Algorithm 3):

1. Initial Attempt: We attempt to model $g(?c)$ using the existing parameters. With our most complex repair strategy (all monomials), we require $n + 1$ independent observations from $\mathcal{D}_a$.
2. Evaluation: If the model fails to adapt, defined by an R^2 value below $1 - 10^{-4}$, we conclude that the current action signature is insufficient.
3. The system then augments the action signature by adding new parameter types drawn from the available domain types, bounded by a maximum combination size l. This procedure is repeated, testing previously untried parameter type combinations in sequence, until a model achieving a satisfactory fit is obtained.

3.4 Theoretical Analysis

▶ **Theorem 1** (Expressiveness of polynomial regression for numeric effects). *Let f be any continuous function on a closed interval $[a, b]$. Then a regression model that represents numeric effects as a linear combination of monomials $1, x, x^2, \ldots, x^k$ can approximate f arbitrarily well as $k \to \infty$, given sufficient data.*

Algorithm 3 Type 2 missing parameter repair.

Input: Parameter types of action a - $\mathcal{P}_a$ ▷ according to our model
Input: Available domain parameter types - $\mathcal{P}_{all}$
Input: Tried domain parameter types - $\mathcal{P}_t$
Input: Environment change dataset - $\mathcal{D}_a$
$model \leftarrow$ fit_model($\mathcal{D}_a$)
$r^2 \leftarrow$ evaluate($\mathcal{D}_a, model$)
if $r^2 < 1 - 10^{-4}$ **and** $|\mathcal{D}_a| > |\mathcal{X}|$ **then**
    $\mathcal{P}_a \leftarrow \mathcal{P}_a \setminus \mathcal{P}_t$ ▷ Revert previous attempt
    $\mathcal{P}_{new} \leftarrow$ next($\mathcal{P}_{all} \setminus \mathcal{P}_a$) ▷ Next set
    $\mathcal{P}_a \leftarrow \mathcal{P}_a \cup \mathcal{P}_{new}$ ▷ Add $\mathcal{P}_{new}$ to a's signature
    $\mathcal{P}_t \leftarrow \mathcal{P}_{new}$ ▷ Set current try
    $\mathcal{D}_a \leftarrow \emptyset$ ▷ Reset dataset for new signature
end if
return *model*

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 $x \in [a, b]$ by the Weierstrass Approximation Theorem [30]. 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 to any desired degree of accuracy. ◀

Completeness

By iterating over candidate parameter types and updating the action signature, FELIX ensures that all relevant variables can be included in the effect model. Combined with monomial regression, this guarantees that the model can eventually represent any polynomial numeric effect up to a chosen degree k that depends on the correct set of parameters. The parameter-swap procedure explores candidate parameter combinations in a combinatorial fashion. When the number of candidate parameter types and the action arity are bounded, this search is polynomial in the input size. In the general case, however, the search over parameter combinations can be exponential. Note that while the algorithm achieves completeness for any arbitrary degree k, searching this unrestricted space inherently requires exponential time due to the combinatorial growth of the monomials.

Runtime Complexity

At the agent level, each iteration requires invoking an external planner, a problem class that is PSPACE-hard. However, the computational overhead incurred by the repair mechanism itself is polynomial. When dealing with a missing action parameter, we iterate over candidate domain parameter types and refit a model when necessary. Assuming the total number of available domain parameter types and the action arity are bounded, the number of iterations of the parameter-type swap loop is polynomial. Within each iteration, the algorithm performs model fitting and evaluation over the dataset $\mathcal{D}_a$. Under standard assumptions that model fitting and evaluation are polynomial in the size of the dataset and the number of parameter types, each iteration runs in polynomial time. Consequently, the additional runtime introduced by the parameter type swap and model fitting procedure is polynomial in the input size.

Table 1 Size of Each Domain.

Domain	Lifted Actions	Lifted Fluents	Functions	Object Types
Expedition	4	2	3–4*	2–3*
Minecraft-pogo	6	0	7–8*	0–2*
Sailing	8	1	4	2
Drone	8	1	14–15*	1–2*

*Depends on model drift (e.g Lv.1 and Lv.2 novelties in Table 2).

4 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 [29]. The Minecraft-Pogo domain is a recently introduced numeric planning domain in which an agent operates in the popular Minecraft game and is tasked with crafting a Pogo-Stick [4]. [1] The size of each domain, in terms of lifted actions, lifted fluents, and functions, is listed in Table 1.

For each domain, we developed a custom problem generator to create random instances. These domains were selected to demonstrate the robustness and versatility of FELIX, including planning domains from established numeric planning benchmarks and recently introduced, less-studied planning problems.

Model Drift Complexity

We evaluate FELIX using five sets of model drift. The first three don't introduce new parameters to the action, and are categorized by their difficulty as *Easy*, *Medium*, and *Hard*. These labels reflect the structural complexity of the model drift introduced: *Easy* scenarios involve linear shifts in existing action effects, while *Medium* and *Hard* sets introduce increasing degrees of non-linearity and latent dependencies. The other two sets, *Lv. 1* and *Lv. 2* introduce a new parameter to the action's signature and effect. *Lv. 1* introduces a parameter of type 1 in the action, *Lv. 2* introduces a parameter of type 2 (see 3.3.2). Each set contains three distinct scenarios, for a total of 15 environment changes per domain. Examples of these cases are listed in Table 2, with a complete list provided in the supplementary material.

Baseline and Competing Algorithms

We evaluated All Monomials Repair, Adaptive Selection Repair, and Adaptive Selection with Signature Repair strategies. All methods were run with $k = 2$ (see Section 3.3.1) and with $l = 1$ configured for the signature repair combination limit (see Section 3.3.2). Performance was measured against *No-Repair* agent, which utilizes the original model without any mechanism to detect or address model-environment mismatches.

We did not compare FELIX directly with Hydra [21] because Hydra relies on manually engineered Model Manipulation Operators (MMOs) that define the search space of possible repairs. Constructing MMO sets capable of expressing the nonlinear polynomial drifts and

[1] Specifically, we use the fully numeric version of the environment, referred to as the "Item Counts" modeling of the problem. For more details, see [5].

Table 2 Representative examples of model-environment mismatches in the Sailing, Expedition, and Minecraft-Pogo domains across three levels of drift complexity.

Domain	Sailing	Expedition	Minecraft-Pogo
Action	*go_x* (=direction)	*move_forward*	*get_log*
Original Effect	Moves toward x	Costs 1 supply	Gets 1 log, decreases tree count by 1
Easy	The actions *go_south_west* and *go_north_east* are switched	Costs 2 supplies	Gets 0.5 log, decreases tree count by 0.5
Medium	A person drifts by: $d = d + 0.3 \cdot drift\_factor$	Costs $0.15 \cdot sled\_capacity + 1$ supplies	Gets $0.05 \cdot trees\_in\_map$ logs, decreases same value from tree count
Hard	A person drifts by: $d = d + 0.001 \cdot d \cdot drift\_factor + 0.1$	Costs $2 \cdot factor^2 + 1$ supplies. Here, factor is a fluent	Gets $0.05 \cdot mine\_factor^2$ logs, decreases same value from tree count
Lv. 1	A person drifts. The parameter "person" isn't originally in the action	Costs $3 \cdot factor\_value$ supplies, *factor_value* is increased by 0.02. Uses new parameter "factor"	Gets $1 \cdot axe\_value$, *axe_value* is reduced by 0.01. Uses new parameter "factor"
Lv. 2	The ship travel speed is based on the person's location. The parameter "person" isn't originally in the action	Costs $3 \cdot factor\_value$ supplies. Uses the new parameter "factor"	Gets $1 \cdot axe\_value$. Uses new parameter 'axe'

signature modifications evaluated in our experiments would require substantial domain-specific engineering, and no standard benchmark MMO library currently exists for such settings. Consequently, a controlled empirical comparison across all evaluated drift scenarios is not feasible. We also did not compare against learning-from-scratch approaches because FELIX performs online repair while continuing to solve planning problems, whereas full relearning approaches typically assume offline training traces.

Implementation Details

For planning, we utilized Nyx, a Python-based, domain-independent PDDL+ planner [26]. While our current experiments focus on PDDL2.1 domains, Nyx was selected for its versatility; it provides a robust suite of search algorithms, including Greedy Best-First Search (GBFS) and Breadth-First Search (BFS), which were essential for our methodology. Furthermore, its Python-based implementation allowed for easier integration with the rest of our codebase, and its native support for PDDL+ facilitates a seamless transition to more complex temporal and continuous domains in future work. Notably, the planner's capacity for PDDL+ does not impact the search performance or behavior for the PDDL2.1 domains used in this study. We configured Nyx to solve problems in each domain as follows. For Sailing, Drone, and Expedition, we used a custom, domain-specific heuristic combined with GBFS. For Minecraft-Pogo, we used a simple BFS. This was decided for faster planning, as all domains (except Minecraft-Pogo) were slow in planning. We preferred faster planning, as our focus is not on planning itself; it is on model repair.

A time limit of 60 seconds was imposed on the planner, after which we declared a failure. 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 problem instances. Each domain–drift combination was evaluated across these instances. We excluded problems where model drift rendered the task unsolvable for an oracle planner using the correct domain model.

For each repair algorithm, we iterated sequentially over the problem instances. For each problem, we performed the following procedure:

1. Run the planner using the current domain model to solve the problem.
2. Execute the resulting plan in the real environment and collect observations of any deviations. Notably, the environment maintains the autonomy to handle incomplete action calls; if an action is received with missing parameters, it completes the grounded action using the first available type-compatible parameter.
3. Record the outcome (success or failure; planner timeouts are counted as failures).
4. Apply the evaluated repair algorithm to update the domain model based on the collected observations.
5. Proceed to the next problem using the updated domain model.

4.1 Results

Figures 2 and 3 show our results across all domains and drift complexity levels. Each arranged in a 5x2 grid of plots, Where each column corresponds to a domain and each row corresponds to a specific level of model-environment mismatch (*Easy*, *Medium*, *Hard*, *Lv. 1*, and *Lv. 2*). In the *Easy*, *Medium*, and *Hard* settings, *Adaptive Selection Repair* is subsumed by *Adaptive Selection with Signature Repair*.

In each plot, the x-axis represents the problem index (excluding problems rendered unsolvable by any environment change within that complexity level). For each combination of a problem, a model drift instance, and a repair algorithm, the outcome is recorded as a binary value: 1 for success and 0 for failure. 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 model-environment mismatches within that category. Thus, each plot illustrates the performance of each repair algorithm as the system encounters an increasingly diverse set of problem instances.

In the Easy and Medium sets, both Adaptive variants achieved near-perfect success almost immediately. In contrast, the *All Monomials* strategy lagged in the Sailing and Drone domains. This was particularly evident in the easy-drone graph. There, many actions produced by the model were inapplicable, hindering the data collection required for the All Monomials strategy, which needs more observations to adapt efficiently.

In the Hard set, all three strategies required more time to adjust due to the increased complexity. However, they eventually converged, reaching similar near-maximum success rates over time.

In the Lv. 1 set, the *Adaptive Selection with Signature Repair* managed to adapt after a few problems. In contrast, the *Adaptive Selection* and *All Monomials* strategies did not adapt as effectively, though they still slightly outperformed the no-repair baseline in the Drone, Minecraft-Pogo, and Expedition domains.

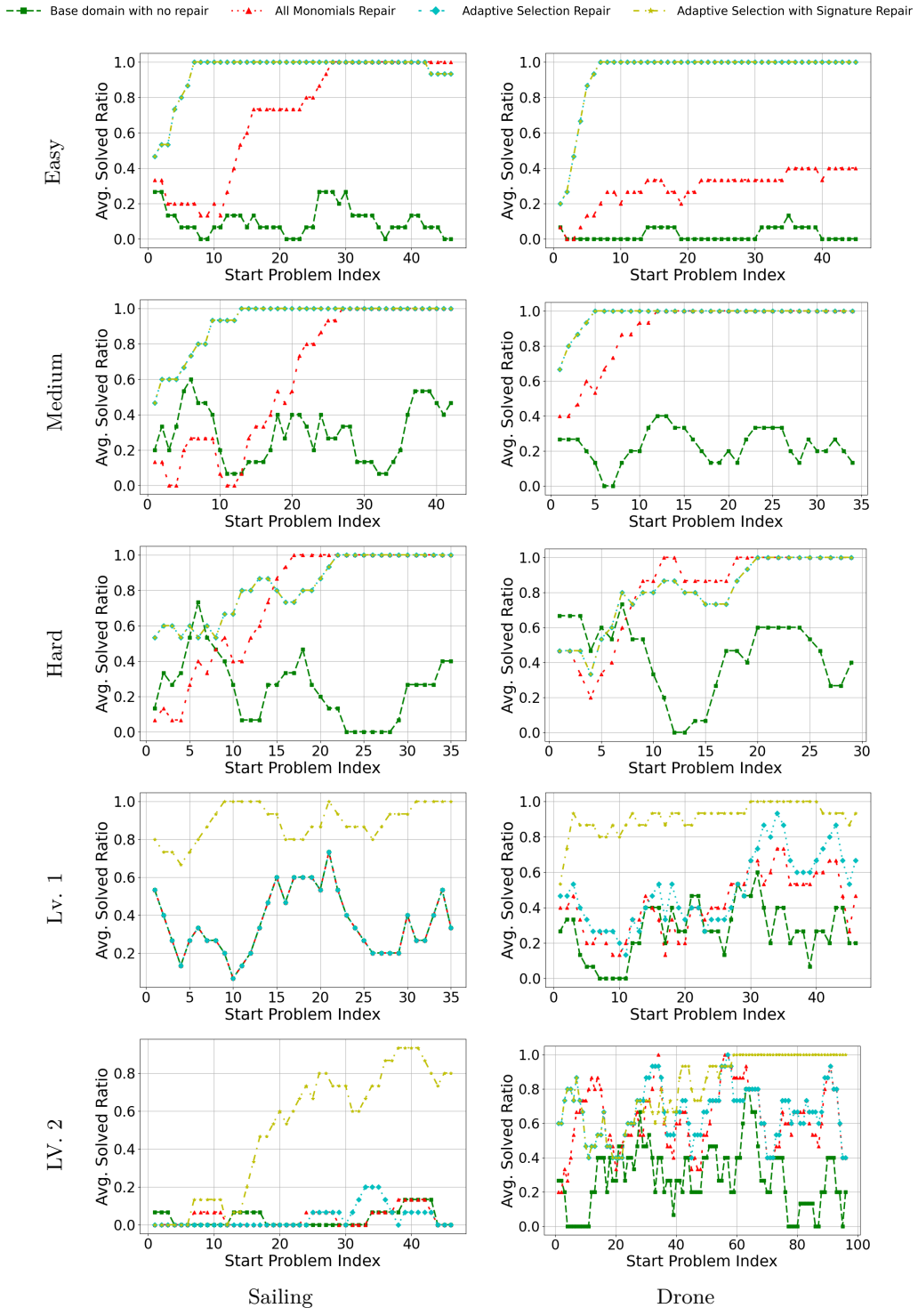

Figure 2 Evaluation results for the Sailing and Drone domains across Easy, Medium, Hard, Lv. 1, and Lv. 2 difficulties.

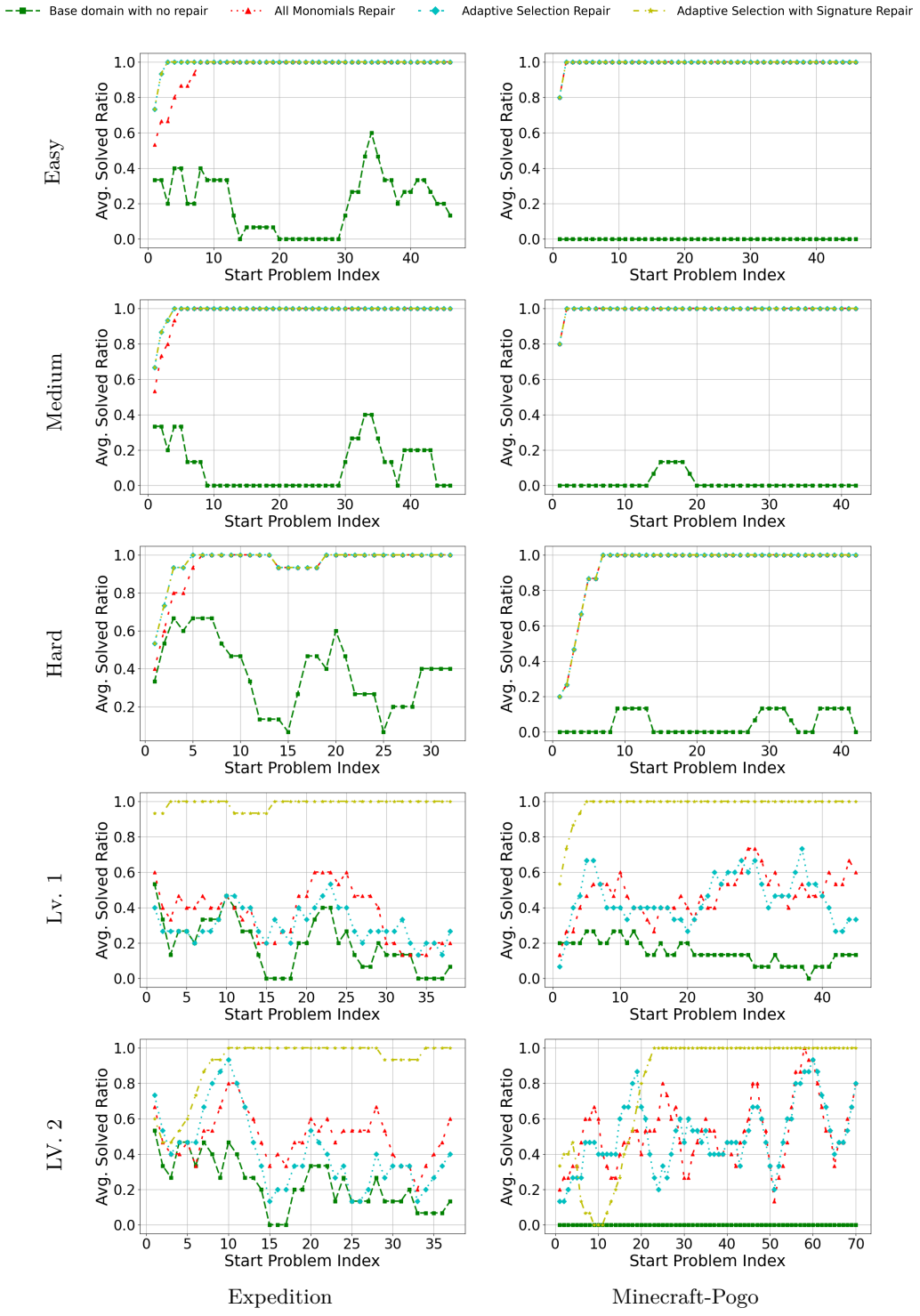

Figure 3 Evaluation results for Expedition and Minecraft-Pogo domains across Easy, Medium, Hard, Lv. 1, and Lv. 2 difficulties.

In the Lv. 2 set, *Adaptive Selection with Signature Repair* required more problems to achieve adaptation than in Lv. 1. This was most noticeable in Minecraft-Pogo, where the system tried an incorrect parameter before finally using the correct one, and in Drone, where adaptation was delayed by slow data collection. Consequently, we evaluated those domains across 100 problems rather than 50, as *Adaptive with signature support* required more data to demonstrate adaptation. Across all domains, the *Adaptive Selection* and *All Monomials* strategies failed to adapt as effectively.

It is important to note that fully repairing the model does not always correlate with consistently solving problems. In many cases, a strategy solves problems before fully repairing the model by acquiring an incomplete numeric-effect function. 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 Selection with Signature Repair* method shows the most consistent adaptation across all model drift sets and domains. It performs comparably to the best repair in each case, combining the strengths of multiple strategies. In some domains and level 2 model drifts, *All Monomials* and *Adaptive Selection* outperform the no-repair baseline. This occurs, for example, in Drone, Expedition, and Minecraft-Pogo. This demonstrates that even without the correct signature, it is possible to approximate real-environment effects using the feature set available under the incorrect signature.

5 Conclusion and Future Work

In this work, we introduced FELIX, a learning-based framework for repairing numeric model drift in planning domains. By detecting discrepancies between predicted and observed transitions at runtime, FELIX selectively updates outdated numeric effects while preserving unaffected components of the domain model. This targeted repair mechanism enables agents to maintain planning performance without requiring full model relearning or explicit identification of the underlying environmental change.

Across multiple domains and levels of drift, FELIX consistently restored planning performance. In particular, the Adaptive Selection with Signature Repair strategy demonstrated rapid recovery under simple drift and steady improvement under more complex drift, highlighting the advantage of combining signature repair with data-driven effect learning.

For future work, we plan to improve FELIX by replacing the current brute-force parameter search with a type-guided heuristic, narrowing the candidate parameters for faster and more efficient repair. We also aim to enhance the adaptive repair strategy by integrating symbolic regression, allowing the framework to discover more complex and non-polynomial functional forms beyond the fixed monomial basis. More broadly, our research sits at the intersection of symbolic model repair, numeric planning, and reliable autonomy. By addressing model discrepancies at run-time, this work contributes to the broader goal of building resilient AI systems.

References

1 Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Etivier, and Sylvie Pesty. A review of learning planning action models. *Knowledge Engineering Review*, 33, 2018. doi:10.1017/S0269888918000188.

2 Masataro Asai, Hiroshi Kajino, Alex Fukunaga, and Christian Muise. Classical planning in deep latent space. *Journal of Artificial Intelligence Research*, 74:1599–1686, 2022. doi:10.1613/JAIR.1.13768.

3 Tomáš Balyo, Martin Suda, Lukáš Chrpa, Dominik Šafránek, Stephan Gocht, Filip Dvořák, Roman Barták, and G Michael Youngblood. Planning domain model acquisition from state traces without action parameters. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 812–822, 2024.

4 Yarin Benyamin, Argaman Mordoch, Shahaf Shperberg, Wiktor Piotrowski, and Roni Stern. Crafting a pogo stick in minecraft with heuristic search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, pages 261–262, 2024.

5 Yarin Benyamin, Argaman Mordoch, Shahaf S Shperberg, and Roni Stern. Model learning to solve minecraft tasks. In *PRL Workshop in ICAPS*, 2023.

6 Yarin Benyamin, Argaman Mordoch, Shahaf S Shperberg, and Roni Stern. Integrating reinforcement learning, action model learning, and numeric planning for tackling complex tasks. *arXiv preprint arXiv:2502.13006*, 2025. doi:10.48550/arXiv.2502.13006.

7 Pascal Bercher, Sarath Sreedharan, and Mauro Vallati. A survey on model repair in ai planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2025.

8 Ethan Callanan, Rebecca De Venezia, Victoria Armstrong, Alison Paredes, Tathagata Chakraborti, and Christian Muise. MACQ: A Holistic View of Model Acquisition Techniques. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, 2022.

9 Stephen N Cresswell, Thomas L McCluskey, and Margaret M West. Acquiring planning domain models using locm. *The Knowledge Engineering Review*, 28(2):195–213, 2013. doi:10.1017/S0269888912000422.

10 Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003. doi:10.1613/JAIR.1129.

11 Christian Fritz. Execution monitoring – a survey. Technical report, University of Toronto, Department of Computer Science, 2005.

12 Alba Gragera, Raquel Fuentetaja, Ángel García-Olaya, and Fernando Fernández. A planning approach to repair domains with incomplete action effects. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 153–161, 2023. doi:10.1609/ICAPS.V33I1.27190.

13 Sachin Grover, Sailik Sengupta, Tathagata Chakraborti, Aditya Prasad Mishra, and Subbarao Kambhampati. Radar: automated task planning for proactive decision support. *Human–Computer Interaction*, 35(5-6):387–412, 2020. doi:10.1080/07370024.2020.1726751.

14 Sergio Jiménez, Tomás De la Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4):433–467, 2012. doi:10.1017/S026988912000001X.

15 Brendan Juba, Hai S. Le, and Roni Stern. Safe learning of lifted action models. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 379–389, 2021. doi:10.24963/KR.2021/36.

16 Leonardo Lamanna, Alessandro Saetti, Luciano Serafini, Alfonso Gerevini, and Paolo Traverso. Online learning of action models for pddl planning. In *IJCAI*, pages 4112–4118, 2021. doi:10.24963/IJCAI.2021/566.

17 Leonardo Lamanna and Luciano Serafini. Action model learning from noisy traces: a probabilistic approach. In *ICAPS*, pages 342–350, 2024. doi:10.1609/ICAPS.V34I1.31493.

18 Songtuan Lin, Alban Grastien, Rahul Shome, and Pascal Bercher. 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, April 2025. doi:10.1609/aaai.v39i25.34861.

19 Alan Lindsay, Jonathon Read, Joao F Ferreira, Thomas Hayton, Julie Porteous, and PJ Gregory. Framer: Planning models from natural language action descriptions. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

20 Thomas Leo McCluskey, Tiago Stegun Vaquero, and Mauro Vallati. Engineering knowledge for automated planning: Towards a notion of quality. In *Proceedings of the Knowledge Capture Conference, K-CAP*, pages 14:1–14:8, 2017. doi:10.1145/3148011.3148012.

21 Shiwali Mohan, Wiktor Piotrowski, Roni Stern, Sachin Grover, Sookyung Kim, Jacob Le, Yoni Sher, and Johan de Kleer. A domain-independent agent architecture for adaptive operation in evolving open worlds. *Artificial Intelligence*, 334:104161, 2024. doi:10.1016/J.ARTINT.2024.104161.

22 Argaman Mordoch, Yarin Benyamin, Shahaf S Shperberg, Brendan Juba, and Roni Stern. Online learning of numeric action models for planning. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems*, pages 1491–1499, 2026.

23 Argaman Mordoch, Brendan Juba, and Roni Stern. Learning safe numeric action models. In *AAAI*, pages 12079–12086. AAAI Press, 2023. doi:10.1609/AAAI.V37I10.26424.

24 Argaman Mordoch, Ori Karat, Lea Shmilovich, Yarin Benyamin, Brendan Juba, and Roni Stern. Safe learning of multi-agent action models from concurrent joint action observations. *Journal of Artificial Intelligence Research*, 86, 2026. doi:10.1613/JAIR.1.20494.

25 Argaman Mordoch, Enrico Scala, Roni Stern, and Brendan Juba. Safe learning of pddl domains with conditional effects. In *International Conference on Automated Planning and Scheduling*, volume 34, pages 387–395, 2024. doi:10.1609/ICAPS.V34I1.31498.

26 Wiktor Piotrowski and Alexandre Perez. Real-world planning with pddl+ and beyond. *arXiv preprint arXiv:2402.11901*, 2024. doi:10.48550/arXiv.2402.11901.

27 José Á Segura-Muros, Raúl Pérez, and Juan Fernández-Olivares. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, pages 1–17, 2021.

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

29 Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, et al. The 2023 international planning competition, 2024.

30 K Weierstrass. Uber die analytische darstellbarkeit sogenannter willkrlicher functionen einer reellen vernderlichen, sitzungsber, akad, 1885.

31 Kai Xi, Stephen Gould, and Sylvie Thiébaux. Neuro-symbolic learning of lifted action models from visual traces. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 34, pages 653–662, 2024. doi:10.1609/ICAPS.V34I1.31528.

32 Kai Xi, Stephen Gould, and Sylvie Thiébaux. Learning lifted action models from unsupervised visual traces. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 36, pages 687–696, 2026.

33 Haoqi Zhang, Yiling Chen, and David Parkes. A general approach to environment design with one agent. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2002–2008, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.