# JUS: Extending SIFT to Learn Planning Domains from Justified Plans

**Alexander Lodemann[1], Michael Ventura[1], Gregor Behnke[2], Birte Glimm[1]**

[1]Ulm University
[2]University of Amsterdam
alexander.lodemann@uni-ulm.de, michael.ventura@uni-ulm.de, g.behnke@uva.nl, birte.glimm@uni-ulm.de

**Abstract**

The recently introduced SIFT algorithm learns planning domains from action sequences alone under an unknown state-space. For theoretical guarantees, the formalism relies on a *value-alternation* assumption, requiring every action effect to change the value of the affected variable. We show that this assumption prevents the approach from learning certain classes of predicates, which occur in standard benchmark domains to record achieved subgoals.

We propose a domain learning algorithm, based on SIFT, that relaxes the value-alternation assumption to capture all possible influences a feature can have on an action. This yields a more expressive upper bound domain, which we reduce using a novel predicate *dominance* relation and symmetry. Furthermore, we use *justified* plans as input to extract forbidden action sequences, enabling two intertwined *counterexample-guided abstraction refinement (CEGAR)*-like loops for iterative domain refinement. Finally, we present a SAT encoding to validate plan justification in a given domain, which directly produces a counterexample to an existing justification.

## 1 Introduction

AI planning is a subfield of symbolic AI concerned with automated decision-making. Given a symbolic description of an application domain, the current state, and a goal, a planner determines an action sequence, called a *plan*, which can be executed in the current state to reach a state that fulfills the goal. Actions are described through their preconditions and the effect their application has on the state.

Specifying the symbolic domain model is a time-consuming task for domain experts. Typically, the modeling is done in the *Planning Domain Definition Language (PDDL)* (McDermott et al. 1998), requiring domain expertise and familiarity with PDDL. Still, the process remains difficult and error-prone (McCluskey, Vaquero, and Vallati 2017). Two research areas address this complexity: *domain repair* and *domain learning*. Repair modifies an existing domain, using example action sequences that should apply but do not (Bercher, Sreedharan, and Vallati 2025), while learning is the special case of starting from a largely empty model and filling in missing information.

Recently, SIFT (Gösgens, Jansen, and Geffner 2025) has been proposed to handle unknown *state-spaces*. It builds on LOCM (Cresswell, McCluskey, and West 2009) to infer object types from input traces and derive an upper bound on predicates describing the state-space. Its novelty lies in the theoretical guarantees it provides under its assumptions. While guarantees such as *safe* learning have been studied before (Juba, Le, and Stern 2021), they have not, to our knowledge, been extended to unknown state-spaces until then.

In this paper, we make two contributions: First, we identify a class of predicates that SIFT excludes due to its value-alternation assumption and show that they are required to model standard benchmark domains. Second, we propose an algorithm based on SIFT that relaxes this assumption and uses *justified* plans (Bachor and Behnke 2024). After computing an upper bound on the domain, a novel *dominance* relation and two intertwined CEGAR-like (Clarke et al. 2003) refinement loops are used to obtain a minimal domain justifying the input.

We next introduce basic notions from planning and domain learning. We then discuss related work in Section 3, followed by an introduction of SIFT and its limitations in Section 4. In Sections 5 and 6, we propose and evaluate our algorithm, before concluding in Section 7.

## 2 Background

We start by introducing STRIPS and lifted planning, before defining basic notions relevant to domain learning.

**STRIPS Planning.** We consider STRIPS (Fikes and Nilsson 1971) extended with negative preconditions:

**Definition 1** (STRIPS Planning Domain). *A STRIPS* planning domain *is a pair $D = (F, A)$, where $F$ is a finite set of Boolean* (state) variables *and $A$ is a finite set of* actions. *Each action $a \in A$ is a tuple $a = \big(prec(a), \textit{neg-prec}(a), add(a), del(a)\big)$, where $prec(a), \textit{neg-prec}(a) \subseteq F$ are the* positive *and* negative preconditions*, and $add(a), del(a) \subseteq F$ are the* add *and* delete effects*, respectively. Any subset $s \subseteq F$ is called a* state*. We use $\textit{eff}(a) = add(a) \cup del(a)$.*

Intuitively, variables in a state $s$ are considered true, whereas those not occurring in $s$ are considered false.

**Definition 2** (Applicability and Successor State). *Let $D = (F, A)$ be a planning domain. An action $a \in A$ is called* applicable *in a state $s \subseteq F$ if $prec(a) \subseteq s$ and $\textit{neg-prec}(a) \cap s = \emptyset$. For $a \in A$ and $s \subseteq F$, the* successor state *obtained by applying $a$ in $s$ is $\gamma_D(s, a) = \big(s \setminus del(a)\big) \cup add(a)$ if $a$*

*is applicable in* $s$ *and it is* `undef` *otherwise. Given a state* $s_0$*, a sequence of actions* $as = \langle a_1, \ldots, a_n \rangle$ *induces a sequence of states* $s_1, \ldots, s_n$ *if, for each* $1 \leq i \leq n$*,* $a_i$ *is applicable in* $s_{i-1}$ *and* $s_i = \gamma_D(s_{i-1}, a_i)$*. We call as an* applicable action sequence *and write* $\gamma_D(s, as)$ *instead of* $\gamma_D(\gamma_D(\ldots, a_{n-1}), a_n)$*.*

In the remainder of the paper we often do not state that an action sequence is applicable, but assume this silently.

**Definition 3** (Planning Problem)**.** *A* planning problem *is a triple* $\Pi = (D, s_i, g)$*, where* $s_i \subseteq F$ *is the* initial state *and the* goal $g$ *is a set of literals over* $F$*. A* plan *or* solution *for* $\Pi$ *is an applicable action sequence* $\pi = \langle a_1, \ldots, a_n \rangle$ *s.t.* $\gamma_D(s_i, \pi) \supseteq \{v \mid v \in g\}$ *and* $\gamma_D(s_i, \pi) \cap \{v \mid \neg v \in g\} = \emptyset$*. The set of all plans for* $\Pi$ *is denoted* $\theta(\Pi)$*. We use* $prec(\pi) = \bigcup_{i=1}^{n} prec(a_i)$*,* $neg\text{-}prec(\pi) = \bigcup_{i=1}^{n} neg\text{-}prec(a_i)$*.*

As an abstraction, we model plans for a single planning problem by $(D, \emptyset, \emptyset)$, encoding the initial state and goal as the first (*initial*) and last (*goal*) actions. This technique is common in various planning contexts, like domain learning (Yang, Wu, and Jiang 2007). If we refer to plans for a domain, we assume the empty planning problem as before and denote $\theta(D)$ for all plans for this problem with the first and last actions being all possible combinations of initial and goal actions.

**Lifted Planning (PDDL).** The Planning Domain Definition Language provides a *lifted* representation of STRIPS in which actions and facts are parameterized by (typed) object variables (McDermott et al. 1998).

**Definition 4** (Lifted Domains and Instances)**.** *Let* $T$ *be a set of* types *and let* $V$ *and* $O$ *be a set of typed* variables *and* objects*, respectively. A* term *is either a variable or object. Let* $P$ *be a set of* predicate symbols *such that each* $p \in P$ *has an associated arity* $n$ *and each of its* $n$ *arguments has an associated type. We only consider well-formed predicates* $p(t_1, \ldots, t_n)$*, where the terms* $t_i$ *correspond to the respective type. A predicate is* ground *if it does not contain variables. A* lifted domain *is a triple* $D = (T, P, A)$*, where* $T$ *is a set of types,* $P$ *is a set of predicates, and* $A$ *is a set of* action schemata*. An* action schema $a(x_1, \ldots, x_n)$*, with* name $a$ *and* parameters $x_1, \ldots, x_n$*, is defined in terms of its preconditions and effects, which are sets of literals over* $P$ *such that* $x_1, \ldots, x_n$ *are all the variables that occur in the preconditions and effects. Given a set of objects* $O$*, an action schema can be instantiated into a (ground) action by replacing its parameters with objects from* $O$ *(respecting the types). An* instance $I = (O, s_i, g)$ *for a lifted domain* $D$ *provides a finite set of typed objects* $O$*, an initial state and a goal. Grounding all action schemata over* $O$ *yields a STRIPS domain* $D_O$ *and a planning problem* $\Pi = (D_O, s_i, g)$*.*

Since an action sequence based on a lifted domain contains the *grounded* actions, the actions carry the action schema's name and the concrete object names with which the action schema was grounded.

It can be useful to distinguish the so-called static predicates, which are those not changed by any action:

**Definition 5** (Static Predicates)**.** *Let* $D = (T, P, A)$ *be a lifted domain. A predicate* $p \in P$ *of arity* $n$ *is called* static *(in* $D$*) if there is no action schema* $a \in A$ *such that* $p(x_1, \ldots, x_n)$ *occurs in* $\mathit{eff}(a)$*.*

We use a running example (neglecting types) adapted from the *visit-all* domain, which has a single action schema $move(x_1, x_2) = \big(\{at(x_1), connected(x_1, x_2)\}, \emptyset, \{at(x_2)\}, \{at(x_1)\}\big)$. The *connected* predicate is static: the connections are needed for grounding of the action schema, but they are never changed.

**Domain Learning.** Domain learning is the inverse problem to planning: given action sequences from a *hidden* domain $D$, the goal is to find a domain where they are applicable. To avoid the trivial empty domain, we require that each planning problem in the hidden domain has a corresponding problem in the learned domain with the same set of plans. We define unobserved lifted domain learning as follows:

**Definition 6** (Unobserved Lifted Domain Learning)**.** *Let* $D = (T, P, A)$ *be a lifted domain and* $\delta$ *a set of action sequences, each applicable in some state of some instance for the* hidden *domain* $D$*. The task of* unobserved lifted domain learning *is to infer* $D' = (T', P', A')$ *from* $\delta$ *such that* $\theta(D_O) = \theta(D'_O)$ *for any set of objects* $O$*.*

# 3 Related Work

Domain learning is a well-established area in AI planning, ranging from fully observable traces to unknown state variables and predicates. Approaches vary in their goals: SAM (Juba, Le, and Stern 2021) learns a *safe* domain allowing only valid action sequences, while FAMA (Aineto, Jiménez Celorrio, and Onaindia 2019) tolerates missing states and actions but offers no guarantees. While many methods assume predefined state-spaces (Yang, Wu, and Jiang 2007; Tantakoun, Muise, and Zhu 2025; Mourão et al. 2012), few address fully empty state definitions.

The closest related research to our work are LOCM (Cresswell, McCluskey, and West 2009), LOCM2 (Cresswell and Gregory 2011), SIFT (Gösgens, Jansen, and Geffner 2025) and the theoretical work done by Bachor, Dekker, and Behnke (2025). Bachor, Dekker, and Behnke show that learning for arbitrary domains, without any information about state variables, is in $\mathrm{coNP}$, if additional assumptions are placed on the input plans. Besides the assumption that the input must be "rational", they also only consider STRIPS planning without negative preconditions. In fact, their reasoning relies mainly on their lemma five, which states that deciding whether there exists a domain in which one plan is applicable and another is not is in P. The proof of this lemma uses the monotonicity of state variables (a state $s \supset s'$ allows strictly for the same or more actions to be applicable in $s$ compared to $s'$), which does not hold if negative preconditions are present.

LOCM learns a lifted PDDL representation from action sequences alone by building state machines for object types that are defined by their position in the parameter list of each action. Since LOCM relies heavily on the domain structure of the hidden domain, the authors further proposed the extension LOCM2, for which no full evaluation exists.

Building upon LOCM's inference of types, SIFT uses the same input as LOCM to build a *most restrictive* domain first. This is achieved by introducing action patterns and features as a variable-centered representation of the domain. Each feature comprises multiple patterns and expresses the hypothesis that a predicate may exist. This predicate appears in the actions defined by the action patterns, with arguments that are likewise specified within those patterns. Inference of all possible features is made feasible through the typing LOCM provides, effectively limiting the number of action patterns with the compatible typing.

SIFT relies on the assumption that all affected state variables in an action's effect change value. While the learned domain does not consider negative preconditions, it is based on an underlying domain that is similar except for additional negative preconditions whenever a state variable is added. This induces a binary (0/1) assignment of action patterns of features. Based on this structure, constraints are defined to identify *valid* features, yielding the most restrictive domain. SIFT's implementation further adds a step to reduce the most restrictive domain by minimizing the number of features. After all states along the input are checked for non-applicable actions, an answer-set program determines the final output domain, which contains a minimal set of features that prohibits all the non-applicable actions.

## 4 SIFT

The current state-of-the-art algorithm for domain learning from action sequences alone, which is theoretically sound, is SIFT (Gösgens, Jansen, and Geffner 2025). As our approach builds on SIFT, we briefly review its definitions and algorithm. SIFT assumes that the hidden domain is *well-formed*. Since their assumption entails not just syntactical correctness but imposes major restrictions on the domain, we call it the *value-alternation* assumption:

**Definition 7** (Value-Alternating Domains). *A STRIPS planning domain $D = (F, A)$ is* value-alternating *if, for each state $s \subseteq F$ and action $a = \big(\mathit{prec}(a), \mathit{neg\text{-}prec}(a), \mathit{add}(a), \mathit{del}(a)\big) \in A$ such that $a$ is applicable in $s$, $\mathit{add}(a) \cap s = \emptyset$ and $\mathit{del}(a) \subseteq s$.*

In a value-alternating domain, each action effect changes the value of the affected variable. Hence, adding something that was already true is not allowed. With negative preconditions, one can enforce value-alternation by adding effects of an action in their complement form to the preconditions of the action. Since SIFT does not handle negative preconditions, value-alternation is not enforced in the learned domain, but it is a prerequisite for the hidden domain.

**Action Patterns and Features.** Rather than describing action schemata by their preconditions and effects, SIFT adopts a representation that describes which predicates in the effects are affected by which parameters. The action schema $\mathit{move}(x_1, x_2) = \big(\{\mathit{at}(x_1), \mathit{connected}(x_1, x_2)\}, \emptyset, \{\mathit{at}(x_2)\}, \{\mathit{at}(x_1)\}\big)$ from above affects the predicate *at* and we say that the (negative) effect $\mathit{at}(x_1)$ is affected by an *action pattern* that refers to the index of the first parameter, written $\mathit{move}[1]$, and the (positive) effect $\mathit{at}(x_2)$ is affected by the action pattern $\mathit{move}[2]$. Formally, this is defined as:

**Definition 8** (Action Pattern). *For an action schema $a(x_1, \dots, x_n)$, an* action pattern *has the form $a[i_1, \dots, i_k]$, where each $i_j$, $1 \leq j \leq k$, is a distinct index from $\{1, \dots, n\}$. We call $k$ the* arity *of the action pattern.*

The goal is to learn the predicates of a domain as well as the action patterns that affect them. The predicates that are possible given a set of action patterns, are called features:

**Definition 9** (Feature). *Given a non-empty set $B$ of action patterns all of arity $k$, a* feature *$f$ is a pair $\langle k, B \rangle$, where $B$ is called the* feature support *of $f$, also referred to as $B_f$.*

A feature $f = \langle k, B \rangle$ represents the hypothesis that the hidden domain contains a predicate $f(x_1, \dots, x_k)$ of arity $k$ whose grounded state variables are affected by all and only the actions and argument positions defined in $B_f$. Consider the *at* predicate from our example above. A feature representing *at* would have arity one and the action patterns $\mathit{move}[1]$ and $\mathit{move}[2]$, so $f = \langle 1, \{\mathit{move}[1], \mathit{move}[2]\} \rangle$.

**Pattern Signs and Constraints.** Under the value-alternation assumption, SIFT considers only two roles for an action pattern: add or delete (with corresponding preconditions), encoded by a binary *sign*. It imposes two constraints: (1) *consecutive* patterns, i.e., those affecting the same state variable within a sequence, must have opposite signs; (2) for known identical states (e.g., shared initial states), patterns that first affect the same variable across sequences must share the same sign. Based on these constraints, SIFT determines which features are *valid* for the input sequences.

**From Features to the Domain.** Depending on the sign assigned to each pattern for a feature, the feature is added as a predicate with the corresponding effects. Preconditions are assigned if, under the constructed effects, a state variable is always true before the application of the action. To reduce the number of calculations, SIFT only learns non-static predicates. To handle static predicates, each action schema of arity $n$ is extended with an $n$-ary predicate. The grounded state variables of static predicates are part of the initial state iff the corresponding action appears in the input.

**Type Inference.** A naive enumeration of all features over the observed action names is exponential in the number of action patterns. Like LOCM (Cresswell, McCluskey, and West 2009), SIFT infers *object types* from the traces to make the approach computationally feasible: an initial type $\omega_{a,i}$ is assigned to each argument position $i$ of each action schema $a$. Two types $\omega_{a,i}$ and $\omega_{b,j}$ are merged if there is an object in the traces that appears both as the $i^{\text{th}}$ argument of $a$ and as the $j^{\text{th}}$ argument of $b$. This process is iterated until a fixpoint is reached, partitioning all objects into disjoint types. Features are then generated only for action patterns with compatible type signatures, which reduces the hypothesis space.

**Maximum Domain.** SIFT introduces the notion of a *maximal* domain $D_{\mathit{max}}$ that is equivalent (i.e., can generate the same action sequences, given the same objects) to the hidden domain $D$. Since SIFT considers all valid features, $D_{\mathit{max}}$ can

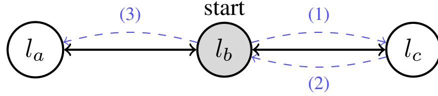


Figure 1: Visit-all domain examples with the agent starting at location $l_b$; the dashed arrows show possible paths: (1) move to $l_c$, (2) return to $l_b$, (3) move to $l_a$; Step (2) adds *visited*$(l_b)$ again, violating SIFT's assumption

be learned given the right input sequences. Gösgens, Jansen, and Geffner (2025, Theorem 20) state that there exists a finite set of sequences that allows learning a domain equivalent to $D$. Deciding whether a given set of sequences has this property is conjectured to be undecidable.

## 4.1 Limitations of SIFT

While SIFT performs well on many benchmark domains, certain structures, which also occur in benchmark domains, cannot be learned due to the value-alternation assumption. Consider again $move(x_1, x_2) = (\{at(x_1), connected(x_1, x_2)\}, \emptyset, \{at(x_2)\}, \{at(x_1)\})$, with an instance with three *fully connected* locations $l_a$, $l_b$, and $l_c$ and an agent starting in $l_a$. Given the applicable action sequence $\langle move(l_a, l_b), move(l_b, l_c), move(l_c, l_a), move(l_a, l_b)\rangle$, SIFT is not able to learn any binary feature: assume that the first action $move(l_a, l_b)$ flips the truth value of the predicate for the feature. Since the value is not flipped again by the next two actions (they use different groundings), the last action (again $move(l_a, l_b)$) would not flip the truth value, violating the value-alternation assumption. An analogous case can be constructed for features of arity one and just one action pattern.

It is also easy to show that SIFT would not have valid features if we allowed multiple *at* variables to be true in the initial state. We focus, however, on a limitation that is more relevant to the standard benchmarks. In the full *visit-all* domain, a unary predicate *visited* records the already visited locations. Since the value-alternation assumption is violated for *visited*, the predicate cannot be learned. We call such kinds of predicates *achievement predicates*:

**Definition 10** (Achievement Predicate). *Let $D = (T, P, A)$ be a planning domain, then an $n$-ary predicate $p \in P$ is an* achievement predicate *if, for each $a \in A$, $p(x_1, \ldots, x_n) \cap prec(a) = \emptyset$ and $p(x_1, \ldots, x_n) \cap del(a) = \emptyset$ and, for some $a \in A$, $p(x_1, \ldots, x_n) \in add(a)$.*

The key intuition is that some domains decompose goals into subgoals that must be achieved once but need not hold at the end. In visit-all, each location defines such a subgoal. Since not all locations must hold in the final state, a *visited* predicate is introduced to refer to intermediate states. While such achievement predicates do not affect which action sequences are applicable, they determine which problems can be modeled. To illustrate why such predicates can be unlearnable for SIFT, consider Figure 1. There are, again, three locations, $l_a$, $l_b$, and $l_c$, and the goal is to visit each at least once. The agent starts at $l_b$, so *visited*$(l_b)$ is initially true. Unlike before, only location $l_b$ is connected to both $l_a$ and $l_c$. Moving to $l_c$ adds *visited*$(l_c)$; reaching $l_a$ then requires returning via $l_b$. Revisiting $l_b$ re-adds *visited*$(l_b)$, although it is already true, violating SIFT's value-alternation assumption. While all hidden-domain action sequences remain executable, the learned domain cannot represent problems where the shortest plan from $l_b$ exceeds one step.

Besides the achievement predicate in visit-all, we were able to identify other domains in which the value-alternation assumption was not met by the original domain model, including but not limited to movie, rovers, and satellite.

To address this, we propose *JUS* (JUstified Sift), which drops the value-alternation assumption of SIFT, yielding a larger but more inclusive version of $D_{max}$ that is iteratively reduced to a usable domain. JUS takes plans as input that reflect practical settings with "reasonable" plans (containing no unnecessary actions), corresponding to the notion of a *perfect justification* as introduced by Fink and Yang (1992) and adapted by Bachor and Behnke (2024). Before introducing justification, we define the *closure* of a plan, as the set of all subplans:

**Definition 11** (Closure of a Plan). *The* closure *of a plan $\pi = \langle a_1, \ldots, a_n\rangle$, written $\bar{\pi}$, is the set of all subsequences of $\pi$ that preserve the relative ordering of actions, i.e., $\bar{\pi} = \{\langle a_{i_1}, \ldots, a_{i_k}\rangle \mid 1 \leq i_1 < \cdots < i_k \leq n, k \leq n\}$.*

With the closure, we can define justification as follows:

**Definition 12** (Well-Justified and Perfectly Justified Plans). *Given a set of plans $\delta = \{\pi_1, \ldots, \pi_n\}$, $\bar{\delta} = \bigcup_{i=1}^{n} \bar{\pi}_i$. A plan $\pi$ of length $n$ for a planning problem $\Pi$ is called* well-justified *if no sequence of length $n-1$ in $\bar{\pi}$ is a plan for $\Pi$; $\pi$ is* perfectly justified *if no element of $\bar{\pi}$ is a plan for $\Pi$.*

Besides applicable sequences, additional information is needed to learn a domain, which can be provided through the implied negative plan examples of justified plans.

# 5 JUS

The value-alternation assumption lets SIFT assign a binary truth value to each pattern of each feature, indicating whether the predicate is added or deleted. Without this assumption, all possible roles a pattern can play with respect to its action must be considered.

**Definition 13** (Influence). *Let $f = \langle k, B\rangle$ be a feature. The* influence *of an action pattern $a[i_1, \ldots, i_k] \in B$ on $f$ is an element of $\mathcal{I} = \{prec, neg\text{-}prec, add, del, prec + del, neg\text{-}prec + add\}$.*

Intuitively, the influence specifies where the predicate implied by $f$ with its pattern grounding appears in the action definition. Only six influences are considered, as others would make $a$ inapplicable ($prec$ + $neg\text{-}prec$), are syntactically ambiguous (simultaneous add/delete), or meaningless (e.g., deleting a required-false fact). Under SIFT's assumptions, only the last two are considered structurally. Thus, JUS strictly generalizes SIFT's binary sign assignment by considering all six influences. We introduce a pattern function to make them explicit.

**Definition 14** (Pattern Function). *A* pattern function *for a feature $f = \langle k, B\rangle$ is a mapping $\phi_f : B \to \mathcal{I}$ that assigns an influence to each action pattern in $B$.*

We extend Definition 9 with a pattern function, yielding an *annotated feature* $f = \langle k, B, \phi_f \rangle$. The translation to a predicate $p_f(x_1, \ldots, x_k)$ is direct: for each $a[t] \in B_f$, $p_f(t) \in \phi(a[t])(a)$, i.e., it appears in the preconditions or effects specified by $\phi$. For simplicity, we still refer to annotated features as features. The domain $D$ with only the predicate implied by $f$ is called $D_f$. Feature enumeration follows SIFT: after type inference, JUS collects all features, then considers all $|\mathcal{I}|^{|B_f|}$ influence combinations, adding the corresponding pattern function to each feature and thus replacing each with multiple annotated features.

To handle the large number of features, JUS first computes an upper bound of $D_{max}$, $D_{res}$, by checking all features for validity, similar to SIFT but with slightly different constraints (discussed below). It then searches for a domain with a minimal number of predicates from $D_{res}$ that preserves all input justifications. We focus on a single planning problem; multiple problems can be handled by intersecting their $D_{res}$.

Validity of a feature $f$ is checked by progressing each input plan in $D_f$. During progression, we collect literals required in the initial state and verify that all preconditions hold. If any precondition fails or a state variable must be both true and false initially, $f$ is invalid. The collected literals form a *partial state*:

**Definition 15** (Partial State)**.** *Let $D = (F, A)$ be a STRIPS domain. A* partial state *$s$ is a set of literals over $F$, containing, for each $v \in F$, at most one of $v$ or $\neg v$.*

In a partial state, a variable's value may be known or unknown. After validating a plan $\pi = \langle a_1, \ldots, a_n \rangle$, the initial state $s_i$ contains all literals that first appear as preconditions and are not affected by earlier actions. For defining this precisely, let, for $v \in \mathit{prec}(\pi)$, $\tau_p(v)$ denote the smallest $i$ such that $v \in \mathit{prec}(a_i)$. We analogously define $\tau_n(v)$ for negative preconditions. With that we define

$$s_i = \{v \mid v \in \mathit{prec}(\pi) \text{ and } v \notin \textstyle\bigcup_{j=1}^{\tau_p(v)-1} \mathit{eff}(a_j)\} \cup \{\neg v \mid v \in \mathit{neg\text{-}prec}(\pi) \text{ and } v \notin \textstyle\bigcup_{j=1}^{\tau_n(v)-1} \mathit{eff}(a_j)\}.$$

For multiple plans, initial states are unioned; if this violates consistency (i.e., includes both $v$ and $\neg v$), the feature is pruned. An upper bound on the goal is obtained by intersecting the states after application of all plans. Literals whose complement does not hold after the application of all plans for a problem build an upper bound on the goal.

Using all valid features yields the domain $D_{res}$, which is an upper bound for $D_{max}$ of SIFT, since all features of $D_{max}$ must be valid for the input and, thus, are part of $D_{res}$. Furthermore, $D_{res}$ relates to the most restrictive domain as introduced by Bachor, Dekker, and Behnke (2025), in that it allows all input plans, not just one.

**Definition 16** (Restrictiveness (Bachor, Dekker, and Behnke 2025))**.** *Let $D$ and $D'$ be grounded domains. We say $D$ is* more restrictive *than $D'$, written as $D \leq D'$ if it holds that $\theta(D) \subseteq \theta(D')$.*

$D_{res}$ is the most restrictive domain: it contains all features and, therefore, predicates that allow the input plans.

**Lemma 1.** *Given an arbitrary set of objects $O$ and two lifted domains $D = (T, P, A)$ and $D' = (T, P', A')$ where $P' \subset P$ and $A'$ contains all actions of $A$ restricted to predicates from $P'$, it holds that $D_O \leq D'_O$.*

*Proof.* If $D_O \not\leq D'_O$, there exists $\pi = \langle a_i, a_1, \ldots, a_n, a_g \rangle \in \theta(D)$ s.t. $\pi \notin \theta(D')$. We mark $a_i$ as $a_0$ and $a_g$ as $a_{n+1}$. Hence, either a precondition or negative precondition must be violated. Assume w.l.o.g. that a positive precondition is violated and there is some $v \in \mathit{prec}(a_k)$ s.t. $v \notin \gamma_{D'_O}(\emptyset, \langle a_0, \ldots, a_{k-1} \rangle)$. Therefore, $\exists\, a_j$ with $j < k$, $v \in \mathit{del}(a_j)$ and $v \notin \bigcup_{\ell=j+1}^{k-1} \mathit{add}(a_\ell)$. Since all predicates of $D'$ are also predicates of $D$, $v$ must also be part of $D$ and be absent in $\gamma_{D_O}(\emptyset, \langle a_0, \ldots, a_{k-1} \rangle)$ as the effects and preconditions are equal in $D_O$ and $D'_O$ concerning $v$. Hence, $\pi$ cannot be a plan for $D_O$. □

---

Algorithm 1: Domain Learning with Annotated Features

---

**Input**: A set of perfectly justified plans $\delta = \{\pi_1, \ldots, \pi_m\}$
**Output**: A minimal learned domain $D_{min}$

```
F = enumerate-all-features(δ) ; s_i = g = ∅
for each feature f ∈ F do
    if f is not valid then F.remove(f)
    else s_i.append(s_i^f) and g.merge(g_f)
Remove symmetric and dominated features from F
while TRUE do
    Generate candidate set candidates ⊆ F
    if δ is justified in D_candidates then
        Restrict s_i, g to candidates
        return: (D_candidates, s_i, g)
    else
        Find counterexample ce for justification
        Refine rules for candidate generation based on ce
```

---

Since $D_{res}$ is most restrictive, all input plans remain justified if they were in the hidden domain. However, its many predicates make it hard to identify the relevant ones. JUS, therefore, computes a domain with a minimal subset of features from $D_{res}$ that still justifies all input plans. As outlined in Algorithm 1, it first prunes features via dominance and symmetry and then applies two intertwined CEGAR-loops to refine the initial state and features until a solution is found.

## 5.1 Dominance and Symmetry

To find a minimal domain, JUS reduces the CEGAR search space by removing so-called *dominated* features, which do not affect the restrictiveness of $D_{res}$ for the input.

**Definition 17** (Dominance)**.** *Let $\delta$ be a set of plans for a hidden planning problem $\Pi$, $f$ and $f'$ valid features w.r.t. $\delta$, and $s_i^p$ and $g$ the partial initial state and goal implied by $\delta$. Then $f$* dominates *$f'$ w.r.t. $\delta$, written $f \leq_\delta f'$, if any $\pi \in \bar{\delta}$ for any $s_i$ compatible with $s_i^p$ that is a plan for $(D_f, s_i, g)$ is also a plan for $(D_{f'}, s_i, g)$.*

Dominance is defined w.r.t. the input plans: a dominated feature cannot prohibit any plan not already prohibited by its dominating feature. Thus, it suffices to consider only non-dominated features. Testing dominance reduces to comparing patterns, pattern functions, initial state, goal, and predicate arity. We first formalize *relevant state variables*:

**Definition 18** (Relevant state variables)**.** *Given a feature $f$, a plan $\pi$ with implied goal $g$, and a domain $D$, the set $rv_f(\pi)$ of* relevant state variables *is prec$(\pi)$ $\cup$ neg-prec$(\pi)$ $\cup$ $g$.*

Relevant ground state variables are all action preconditions in a plan including all goal variables. For a set of plans, we write $rv_f(\delta) = \bigcup_{\pi \in \delta}(\textit{prec}(\pi) \cup \textit{neg-prec}(\pi)) \cup g$.

**Theorem 1.** *Given two features $f$ and $f'$, a set of plans $\delta$ and their implied partial initial state $s_i$ and goal $g$, the following conditions are sufficient to show $f \leq_\delta f'$.*

1. *$f$ and $f'$ have the same arity $k$.*
2. *$\forall p_f(x_1, \dots, x_k) \in rv_f(\delta)$: Either $p_f(x_1, \dots, x_k)$, $p_{f'}(x_1, \dots, x_k) \in s_i$ or $\neg p_f(x_1, \dots, x_k)$, $\neg p_{f'}(x_1, \dots, x_k) \in s_i$. The same holds for all relevant state variables of $f'$ compared to $f$.*
3. *$f$ and $f'$ have the same effects.*
4. *All (negative) pre- and goal-conditions of $f'$ must also occur in $f$.*

In special cases, condition 3 can be relaxed further:

**Corollary 1.** *If $f$ and $f'$ do not contain any negative precondition (resp. positive precondition), condition 3 can be relaxed to:*
*3.1 All delete (resp. add) effects of $f'$ must also occur in $f$ and all add (resp. delete) effects of $f$ must occur in $f'$.*

*Proof.* Proof by contradiction. Consider a plan $\pi_o$ and two features $f, f'$, adhering to all conditions of Theorem 1. Assume $f \not\leq_{\pi_o} f'$. Hence, some plan $\pi = \langle a_1, \dots, a_n \rangle \in \bar{\pi}_o$ for the problem $\Pi_f = (D_f, s_i, g)$ must exist, which is no plan for $\Pi_{f'} = (D_{f'}, s_i, g)$. We represent the problems with an initial ($a_0$) and goal ($a_{n+1}$) action. The new plan is $\pi = \langle a_0, \dots, a_{n+1} \rangle$. Hence, $\gamma_{D_f}(\emptyset, \pi)$ must be defined, whereas $\gamma_{D_{f'}}(\emptyset, \pi) = \texttt{undef}$. For $\pi$ not to be applicable in $D_{f'}$, an action $a_j \in \pi$ must exist with an unmet precondition $p$. Assume w.l.o.g. $p$ to be a positive precondition. Then, $p \notin \gamma_{D_{f'}}(\emptyset, \langle a_0, \dots, a_{j-1} \rangle)$. Therefore, $p$ has been deleted after the last time it has been added. Since all preconditions of $f'$ are also preconditions of $f$, $p \in \gamma_{D_f}(\emptyset, \langle a_0, \dots, a_{j-1} \rangle)$. In $D_f$, $p$ must be added by an action $a_k$ in $\langle a_0, \dots, a_{j-1} \rangle$ and not be deleted by any $a_x$ with $k < x < j$. In $D_{f'}$ either $p$ is not added by $a_k$ or there exists an action $a_x$ that deletes $p$ just in $D_{f'}$. If $f$ contains positive and negative preconditions, both options imply that the effects of $f$ and $f'$ are not equal, contradicting assumption 3. In the other case, according to assumption 3.1, all delete effects of $f'$ must also be part of $f$ contradicting the existence of $a_x$ and all add effects of $f$ occur in $f'$ contradicting the existence of $a_k$. □

Removing dominated features from $D_{res}$ preserves maximal restrictiveness for the given input plans, but not necessarily for other plans from the same hidden domain or even subsets of the input. This is because there might exist other problems for $D_{res}$ that require different values for the initial state, which allows different plans to be applicable. For a concise example, consider a ground domain (which can easily be extended to a lifted one) with actions $a = (\{f_1, f_2\}, \emptyset, \{f_3\}, \{f_1, f_2\})$ and $b = (\{f_1\}, \emptyset, \{f_3\}, \emptyset)$ and the plans $\{\langle a \rangle, \langle b \rangle\}$. The plans imply $s_i = \{f_1, f_2\}$ and $g = \{f_3\}$. Since $f_1$ has an additional precondition compared to $f_2$, $f_1 \leq_{\{\langle a \rangle, \langle b \rangle\}} f_2$. Consider now the initial state $\{f_1\}$. Without $f_2$, $a$ equals $b$ keeping both plans solutions. With $f_2$, only $\langle b \rangle$ is a solution. Therefore, removing $f_2$ eliminates the ability to forbid the plan $\langle a \rangle$ for the unknown problem.

After removing dominated features, we can further reduce the set by eliminating *symmetric* features.

**Definition 19** (Symmetry)**.** *Opposite influences are prec to neg-prec, add to del, and prec + del to neg-prec + add. We call two features $f$ and $f'$* symmetric *if they contain the same action patterns and their pattern functions produce opposite influences for the same patterns.*

By symmetry, all grounded state variables of the features take opposite values after the first influencing action. Hence, removing one of the two symmetric features preserves the domain's restrictiveness.

## 5.2 CEGAR Loop

To find a minimal feature set justifying the input, JUS iteratively refines *candidate* features, starting small and updating them using counterexamples (Algorithm 1). Candidates are drawn from features remaining after removing dominated and symmetric ones and validated against the requirements of the input plans. If validation fails, JUS generates rules indicating required features and selects new candidates. This process is inspired by counterexample-guided abstraction refinement (CEGAR) (Clarke et al. 2003; Seipp and Helmert 2013). JUS selects candidates based on *hitting sets*, which have been used in multiple different settings within AI planning (Lin, Grastien, and Bercher 2023; Bavandpour et al. 2025; Welt et al. 2025):

**Definition 20** (Hitting Set)**.** *Given a set of sets $\Lambda = \{\lambda_1, \dots, \lambda_n\}$, a* hitting set *$hs(\Lambda)$ is a set in which $hs(\Lambda) \cap \lambda_i \neq \emptyset$ for all $1 \leq i \leq n$. If $\Lambda = \emptyset$, $hs(\Lambda) = \emptyset$.*

Candidate features are selected by computing a minimal hitting set of the set of all *feature repairs* for all previous counterexamples.

**Definition 21** (Feature Repair)**.** *A feature $f$ is called a* repair *for a counterexample ce for the justification of a set of plans $\delta$ if it is possible to create a problem $(D_f, s_i, g)$ so that ce is not a plan for it, but all plans in $\delta$ are.*

A hitting set is used because each counterexample must be prohibited. Since the current candidates are insufficient, at least one feature that repairs each counterexample must be added, i.e., one from each repair set, which is exactly the hitting set problem. JUS considers a feature as a repair only if it is not already a candidate.

The simplest counterexample and repair arises before selecting candidates: the empty feature set permits the empty plan. Thus, the first repair consists of all features containing at least one goal variable, and is added to $\Lambda$ initially. Given candidates, JUS checks whether all plans are justified and generates a counterexample otherwise by testing all plans with exactly one missing action for well-justification. For perfect justification, JUS uses the following SAT encoding, based on the encoding introduced by Kautz and Selman (1992), for each plan $\langle a_0, \dots, a_n \rangle$ and learned initial state $s_i$ and goal $g$ with the variables:

- $a_i, 0 \leq i \leq n$: indicating if the $i$-th action is present
- $v_i^t, 0 \leq t \leq n+1$: value of $i$-th state variable at time $t$

The clauses for a partial initial state $s_i$ and a goal $g$ are built in the following way:

$$\bigwedge_{j \in s_i} v_j^0 \wedge \bigwedge_{j \notin s_i} \neg v_j^0 \wedge \bigwedge_{v_j \in g} v_j^{n+1} \wedge \bigwedge_{\neg v_j \in g} \neg v_j^{n+1}$$

$$\forall a_i : a_i \rightarrow \bigwedge_{v_j \in prec(a_i)} v_j^i \wedge \bigwedge_{v_j \in neg\text{-}prec(a_i)} \neg v_j^i \wedge \bigwedge_{v_j \in add(a_i)} v_j^{i+1} \wedge \bigwedge_{v_j \in del(a_i)} \neg v_j^{i+1}$$

$$\forall a_i : \neg a_i \rightarrow (v_j^i \leftrightarrow v_j^{i+1}) \text{ if } v_j \in \mathit{eff}(a_i)$$

$$\forall a_i : v_j^i \leftrightarrow v_j^{i+1} \text{ if } v_j \notin \mathit{eff}(a_i)$$

The encoding enforces that actions apply only if preconditions hold, effects persist, and unaffected variables remain unchanged. It also ensures consistency with the partial initial state and goal satisfaction. To avoid the trivial solution where all actions are present, we add $\vee_{i=0}^{n} \neg a_i$. Thus, if the formula is unsatisfiable, the plan is perfectly justified; otherwise, the true action variables define a counterexample.

While this encoding yields a counterexample, it is insufficient for finding new feature repairs. Before adding features, one must check whether a different initial state (under the current candidates) could prohibit it. JUS therefore employs a second CEGAR loop within counterexample generation, refining the initial state until no further refinement is possible or the candidates form a solution. Refining the initial state for a counterexample is done by progressing it in the candidate domain. JUS then compares the resulting initial state $s_i^c$ with the original $s_i$. If $s_i^c \subseteq s_i$, no refinement is possible without invalidating the input plans. Otherwise, the literals in $rep = s_i^c \setminus s_i$ are *literal repairs*, as setting them in the initial state can prohibit the counterexample.

JUS uses a SAT solver to select an initial state across multiple counterexamples. For each counterexample with literal repairs $l_0, \ldots, l_n$, it adds one clause over variables $v_i$ (representing the variable of $l_i$): a disjunction of $v_i$ for literals requiring true and $\neg v_i$ for those requiring false values. Solving this SAT formula either yields an adapted initial state to test or, via a minimal unsatisfiable subset, a set of counterexamples that cannot be ruled out by any adaptation. For example, if a variable $v$ is not set initially, one counterexample may yield repair $v$ and another $\neg v$; the minimal unsatisfiable set contains exactly these two counterexamples. After generating counterexamples, each non-candidate feature is tested for repair. As with literal repair, each counterexample is propagated through $D_f$ and $s_i^c$ is compared to $s_i$. If any literal repairs are found, the feature is added to the repair set.

The algorithm terminates either when it finds a domain where all input plans are justified, or after all features have been considered, yielding empty repair and hitting sets. The latter occurs only if no such domain exists.

## 6 Evaluation

We implemented JUS with support for perfect justification only, since our input only entails such. We evaluated SIFT, JUS and JUS with a restriction to include only features close to SIFT (superset of the features SIFT generates), each with their full $D_{res}$ and the minimized version. Each of the domains is compared to the hidden domain.

**Benchmark Domains and Plan Generation** We draw our benchmark domains from the aibasel-repository,[1] retaining only domains with at most the requirements `:strips` and `:typing` and that provide multiple problems per domain file. As a plan generator, we adapted the symK-planner (Speck, Mattmüller, and Nebel 2020) to output perfectly justified plans. We limited the plans per problem to 50 and generated until we gathered 100 plans per domain.

**Metrics** We compare a learned domain $D_\ell$ with the hidden domain $D_h$ based on two instances $I_\ell$ and $I_h$ where $I_\ell$ is learned from the plans for $I_h$. From the part of the state-space defined by the progression of the plans used as input for the learning algorithms, a queue-based exploration up to a depth $k$ is done. During the exploration, the sets of applicable actions $A_{D_\ell}$ and $A_{D_h}$ are recorded. The exploration is guided by applying actions that are applicable in both domains, disregarding states that are only reachable in one domain. From the aggregated sets we report *soundness* $|A_{D_\ell} \cap A_{D_h}|/|A_{D_\ell}|$ and *completeness* $|A_{D_\ell} \cap A_{D_h}|/|A_{D_h}|$.

**Settings** Experiments ran all on the BwUniCluster 3.0 using Intel Xeon Platinum 8358 processors. For the plan generation, we imposed a time limit of 30 minutes and 8 GiB of RAM per problem and 1 CPU core. For the learning task, the limits were 6 hours, 32 GiB of RAM and 1 CPU core, for the comparison 18 hours, 250 GiB and 2 CPU cores, with a depth limit of 10 and a limit of 5 minutes for each branch.

**Learning Domains** After excluding all domains that did not fit the criteria or did not generate sufficient plans, we show the time and memory consumption of the different approaches in Table 1. The table only shows one SIFT version since SIFT outputs the maximum and minimized domain.

As one can see, SIFT is able to learn 16 out of 21 domains, whereas JUS is only capable of learning 7. JUS-Max learned 11, since it was able to skip the minimization step. The versions with limited features managed to learn 11 each and proved that no domain with just these features exists in which the input is perfectly justified (as marked by failure in the learning). The performance advantage of SIFT was to be expected as JUS needs to consider many more features and needs to ensure perfect justification.

**Results** Of the five domains which were learned by all approaches, only four were able to be evaluated within the given limits. We present the results for these four domains in Table 2. For each domain, we calculated the averages of soundness and completeness over all problems using all available plans for the problem. We excluded any problem that violated the given limits for any domain.

As expected, the $D_{res}$ has 100% soundness across all approaches, since it retains all predicates keeping all inapplicable actions of the hidden domain inapplicable. For the minimized versions, soundness differs substantially. SIFT preserves perfect soundness, whereas JUS drops to 22.0% and JUS-SIFT to 12.1%, indicating that JUS struggles to retain sufficiently many predicates.

[1] https://github.com/aibasel/downward-benchmarks

| Domain | SIFT Mem | SIFT Time | JUS Mem | JUS Time | JUS-Max Mem | JUS-Max Time | JUS-SIFT Mem | JUS-SIFT Time | JUS-SIFT-Max Mem | JUS-SIFT-Max Time |
|---|---|---|---|---|---|---|---|---|---|---|
| barman-opt14 | - | X | - | X | X | - | - | X | - | X |
| blocks | 0.3 | 0 | 0.2 | 3 | 0.1 | 2 | 0.0 | 0 | 0.0 | 0 |
| depot | 0.5 | 1 | - | X | 1.2 | 82 | 0.1 | 0 | 0.1 | 0 |
| driverlog | 0.3 | 0 | 5.0 | 266 | 4.8 | 37 | 0.0 | 0 | 0.0 | 0 |
| freecell | X | - | X | - | X | - | X | - | X | - |
| grid | 2.5 | 1 | - | X | 0.9 | 23 | 0.0 | 0 | 0.0 | 0 |
| gripper | 0.0 | 0 | 0.0 | 0 | 0.0 | 0 | 0.0 | 0 | 0.0 | 0 |
| logistics98 | 1.4 | 0 | - | X | 4.0 | 16 | 0.0 | 0 | 0.0 | 0 |
| miconic | 0.7 | 0 | 0.3 | 3 | 0.2 | 2 | 0.0 | 0 | 0.0 | 0 |
| movie | 0.1 | 0 | - | X | 0.7 | 10 | 0.0 | 0 | 0.0 | 0 |
| mystery | 2.6 | 1 | 0.3 | 12 | 0.4 | 3 | 0.0 | 0 | 0.0 | 0 |
| pipesworld-nt | 28.1 | 58 | - | X | - | X | 0.4 | 24 | 0.6 | 21 |
| rovers | - | X | X | - | - | X | - | X | - | X |
| storage | 0.4 | 1 | X | - | X | - | 0.7 | 2 | 0.7 | 2 |
| termes-opt18 | 4.6 | 24 | - | X | - | X | - | X | - | X |
| termes-sat18 | 22.8 | 29 | - | X | - | X | - | X | - | X |
| thoughtful-sat14 | X | - | X | - | X | - | X | - | X | - |
| tpp | X | - | X | - | X | - | - | X | - | X |
| visitall-opt11 | 0.0 | 0 | 0.0 | 0 | 0.0 | 0 | F | F | F | F |
| visitall-opt14 | 0.2 | 0 | 0.0 | 0 | 0.0 | 0 | F | F | F | F |
| zenotravel | 22.9 | 236 | - | X | - | X | - | X | - | X |

Table 1: Learning results: memory in GiB, time in min, X denotes out-of-memory and timeout, resp.; F marks failure to learn; for JUS-Max $D_{res}$ was learned; JUS-SIFT was limited to a smaller set of features, which is still a superset of SIFT's

| Domain | SIFT C | SIFT S | SIFT-$D_{res}$ C | SIFT-$D_{res}$ S | JUS C | JUS S | JUS-$D_{res}$ C | JUS-$D_{res}$ S | JUS-SIFT C | JUS-SIFT S | JUS-SIFT-$D_{res}$ C | JUS-SIFT-$D_{res}$ S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| blocks | 29.4 | 100.0 | 28.0 | 100.0 | 100.0 | 25.5 | 22.7 | 100.0 | 98.91 | 7.73 | 28.0 | 100.0 |
| driverlog | 39.0 | 100.0 | 37.8 | 100.0 | 74.8 | 26.3 | 34.5 | 100.0 | 99.3 | 10.6 | 28.2 | 100.0 |
| gripper | 49.1 | 100.0 | 49.1 | 100.0 | 50.5 | 21.0 | 39.1 | 100.0 | 100.0 | 22.0 | 49.1 | 100.0 |
| miconic | 40.5 | 100.0 | 40.5 | 100.0 | 63.6 | 15.1 | 30.0 | 100.0 | 100.0 | 8.7 | 40.5 | 100.0 |
| Average | 39.5 | 100.0 | 30.8 | 100.0 | 72.1 | 22.0 | 31.6 | 100.0 | 99.5 | 12.2 | 36.4 | 100.0 |

Table 2: Evaluation results: averages over instances where all algorithms produced a result showing completeness (C) and soundness (S), average calculated over all domains, per domain maxima are shaded

The completeness is reversed. All $D_{res}$ score below 37% completeness, as they are over-restrictive. SIFT improves this through the minimization marginally to 39.5%, whereas JUS manages to increase it to 72.1% and 99.5% with JUS-SIFT. Together, this shows a trade-off where JUS minimizes more aggressively, as it just considers the plans that threaten justification as invalid and SIFT uses its value-alternation assumption to generate inapplicable action sequences. Closing this gap by giving JUS access to more forbidden plans is mandatory and a central challenge for future work.

## 7 Conclusion

We identify a structural limitation of SIFT due to its value-alternation assumption, which excludes what we call *achievement predicates*, and show that such predicates occur in current AI planning benchmarks. To address this, we introduce JUS, a domain learning algorithm based on SIFT that learns from justified plans rather than arbitrary sequences. JUS extends SIFT's binary sign assignment to a six-valued influence, enabling representation of achievement predicates. JUS further employs a new dominance relation to reduce features in the minimization step. Our evaluation shows that JUS achieves much higher completeness than SIFT at the cost of reduced soundness, highlighting the challenges of dropping the value-alternation assumption: the search space grows and fewer negative examples are available, as only counterexamples to perfect justification can serve as forbidden plans.

JUS's minimization step is still too permissive. We plan to address this by adding negative examples via more input plans and optimal plan lengths, naturally extending the CEGAR loop to detect forbidden shorter plans. Interactively involving experts in the process is another option for future work as are parallelized feature validity checks.

## Acknowledgments

A. Lodemann, M. Ventura and B. Glimm acknowledge funding from the German Research Foundation – GRK 3012 – 520750254 (kemai.uni-ulm.de). The authors acknowledge support by the state of Baden-Württemberg through bwHPC. G. Behnke acknowledges the project "Exploiting Problem Structures in SAT-based Planning", file number OCENW.M.22.050, research program Open Competition by the Dutch Research Council.

## References

Aineto, D.; Jiménez Celorrio, S.; and Onaindia, E. 2019. Learning action models with minimal observability. *Artificial Intelligence*, 275: 104–137.

Bachor, P.; and Behnke, G. 2024. Learning Planning Domains from Non-redundant Fully-Observed Traces: Theoretical Foundations and Complexity Analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 20028–20035. AAAI Press.

Bachor, P.; Dekker, P. M.; and Behnke, G. 2025. Is This Plan Necessarily Redundant? On the Computational Complexity of Unobserved Domain Learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 11–20. AAAI Press.

Bavandpour, N. K.; Lauer, P.; Lin, S.; and Bercher, P. 2025. Repairing Planning Domains Based on Lifted Test Plans. In *Proceedings of the 28th European Conference on Artificial Intelligence*, 4774–4781. IOS Press.

Bercher, P.; Sreedharan, S.; and Vallati, M. 2025. A Survey on Model Repair in AI Planning. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, 10371–10380. International Joint Conferences on Artificial Intelligence Organization.

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50: 752–794.

Cresswell, S.; and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 42–49. AAAI Press.

Cresswell, S.; McCluskey, T.; and West, M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 338–341. AAAI Press.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2: 189–208.

Fink, E.; and Yang, Q. 1992. Formalizing Plan Justifications. In *Proceedings of the Ninth Conference of the Society for Computational Studies of Intelligence*, 9–14. The Society.

Gösgens, J.; Jansen, N.; and Geffner, H. 2025. Learning lifted strips models from action traces alone: A simple, general, and scalable solution. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 189–197. AAAI Press.

Juba, B.; Le, H. S.; and Stern, R. 2021. Safe Learning of Lifted Action Models. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 379–389. IJCAI Organization.

Kautz, H.; and Selman, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 359–363. John Wiley & Sons, Inc.

Lin, S.; Grastien, A.; and Bercher, P. 2023. Towards Automated Modeling Assistance: An Efficient Approach for Repairing Flawed Planning Domains. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence*, 12022–12031. AAAI Press.

McCluskey, T. L.; Vaquero, T. S.; and Vallati, M. 2017. Engineering Knowledge for Automated Planning: Towards a Notion of Quality. In *Proceedings of the 9th Knowledge Capture Conference*, 1–8. Association for Computing Machinery.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

Mourão, K.; Zettlemoyer, L.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS operators from noisy and incomplete observations. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, 614–623. AUAI Press.

Seipp, J.; and Helmert, M. 2013. Counterexample-Guided Cartesian Abstraction Refinement. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 347–351. AAAI Press.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, 9967–9974. AAAI Press.

Tantakoun, M.; Muise, C.; and Zhu, X. 2025. LLMs as planning formalizers: A survey for leveraging large language models to construct automated planning models. In *Findings of the Association for Computational Linguistics*, 25167–25188. Association for Computational Linguistics.

Welt, M.; Lodemann, A.; Olz, C.; Bercher, P.; and Glimm, B. 2025. Calculating Optimal Corrections for Unsolvable Planning Problems. In *Proceedings of the 28th European Conference on Artificial Intelligence*, 4637–4644. IOS Press.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning Action Models from Plan Examples Using Weighted MAX-SAT. *Artificial Intelligence*, 171: 107–143.