Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

Domain Model Acquisition From Binary Traces

Arash Haratian[1], **Arnaud Lequen**[1], **Daniel Gnad**[1,2], **Jendrik Seipp**[1]

[1]Linköping University, Sweden
[2]Heidelberg University, Germany
{arash.haratian, arnaud.lequen, jendrik.seipp}@liu.se, daniel.gnad@uni-heidelberg.de

Abstract

Automated planning systems require symbolic models, but real-world data is often collected in subsymbolic formats such as binary encodings. We address this gap by introducing a domain model acquisition algorithm that handles subsymbolic state representations. Our algorithm takes as input plan traces where states are represented as binary vectors rather than fluent sets, along with action signatures, predicate definitions, and type hierarchies. It then simultaneously learns the mapping from binary state representations to symbolic fluents and constructs lifted action models with preconditions and effects. Across all evaluated benchmark domains from the International Planning Competition, our algorithm accurately maps bits to fluents, and the learned action models closely match reference representations, despite having access only to binary encodings.

1 Introduction

Automated planning focuses on finding sequences of actions, called *plans*, that transform the world from an initial state into a state satisfying the goal conditions. Domain-independent planners require symbolic models that describe the world and the available actions. These symbolic models are commonly represented using languages like PDDL (Planning Domain Definition Language) (Fox and Long 2003; Haslum et al. 2019). Instances modeling similar problems share a common *domain*, which contains the predicates describing the world and a representation of the high-level actions available to the agent.

The domain model acquisition problem consists of learning such symbolic representations that accurately capture the dynamics of a given domain. While existing tools like GIPO (Simpson, Kitchin, and McCluskey 2007), Unified Planning (Micheli et al. 2025), and planning.domains (Muise 2016) provide valuable assistance in domain modeling, they typically require human experts to manually create these models. Automated domain model acquisition aims to reduce or eliminate this expert dependency by learning domain models from observed data, such as traces of world states and actions. Current approaches vary in their requirements, from complete traces with full state and action information to partial traces with missing or noisy data. However, existing domain model acquisition methods generally require complete information about action parameters, which limits their applicability in real-world scenarios where such detailed information is unavailable.

In real-world applications, however, data is often collected in subsymbolic form, whereas planning systems require symbolic information about the environment and the effects of actions. This challenge arises in many applications: in robotics, sensors provide continuous measurements, but robots need symbolic action models for planning; in process mining, detailed execution logs must be abstracted into symbolic workflow models; in reinforcement learning, binary encodings can be obtained from neural-network representations of MDP states by executing actions and observing the resulting states. Despite its importance, the conversion from subsymbolic to symbolic data has received little attention in domain learning research.

In this paper, we introduce the first approach to the domain model acquisition problem when traces contain only binary-encoded data. We assume that, although the underlying representation is subsymbolic, certain information about the domain is readily available or obtainable. For instance, we assume that the algorithm knows which types of objects exist in the world and which properties and relations hold between them. This information can either be provided manually or inferred using off-the-shelf vision or language models. For example, vision systems can reliably distinguish cups, tables, and robots without knowing their symbolic identifiers, and can recognize which objects are involved in an observed action. Even if these techniques are imperfect and provide superfluous information, our algorithm will still infer a correct transition model, as it only uses the predicates and types that are required.

Our algorithm exploits the types of the objects to structure the learned domain: it searches for a mapping from bits to typed fluents and, at the same time, synthesizes typed action schemas whose preconditions and effects explain the observed transitions. The resulting model uses typed action parameters and predicates over typed objects, making it directly usable by off-the-shelf domain-independent planners. Our experiments on standard IPC benchmark domains demonstrate that our approach can accurately recover both the mapping from bits to fluents and the action models.

2 Background

This section introduces our target language, in which the learned model will be expressed. Our definition of a PDDL planning task (Helmert 2009) is enriched with typing, since types constitute crucial information in our framework.

Definition 1 (Type tree)**.** A *type tree* $\mathcal{T}$ is a non-empty tree where each node is labeled by a symbol, called a *type*. For any type $\tau \in \mathcal{T}$, we call any descendant τ' a *strict subtype* of τ. Type τ' is a *subtype* of τ (denoted $\tau' \preceq \tau$) when τ' is a strict subtype of τ or when $\tau' = \tau$.

Definition 2 (Object class)**.** Let $\mathcal{O}$ be a set of elements called *objects*. Any subset of $\mathcal{O}$ is an *object class*. A class c_i is said to be a *subclass* of class c_j if $c_i \subseteq c_j$.

Definition 3 (Type hierarchy)**.** A *type hierarchy* $\mathcal{H}$ over type tree $\mathcal{T}$ is a set of object classes such that $\mathcal{O} \in \mathcal{H}$ for a set of objects $\mathcal{O}$, and such that each object class of $\mathcal{H}$ is mapped to a unique type of $\mathcal{T}$. This mapping $\tau : \mathcal{H} \to \mathcal{T}$ is such that for any pair c_i, c_j of object classes:

- c_i is a subclass of c_j iff $\tau(c_i)$ is a subtype of $\tau(c_j)$;
- $c_i \cap c_j = \varnothing$ iff $\tau(c_i)$ is not a subtype of $\tau(c_j)$ (and conversely).

We say that object $o \in \mathcal{O}$ is of type $\tau(o) := \tau(c)$ where c is the smallest (for inclusion $\subseteq$) class of $\mathcal{H}$ to which o belongs.

Definition 4 (Predicate, atoms and fluents)**.** A *predicate* p is a symbol, with which is associated:

- an arity $ar(p) \in \mathbb{N}$ (including $ar(p) = 0$ for nullary predicates);
- a type for each of its arguments. For $i \in \{1, \dots, ar(p)\}$, the type of its argument at position i is denoted $\tau_p(i) \in \mathcal{T}$.

An *atom* is a predicate for which each argument is associated with a symbol, which can be a variable symbol or an object of $\mathcal{O}$. When the i-th argument of the atom is an object $o \in \mathcal{O}$ (associated with type hierarchy $\mathcal{H}$), then we require that $\tau(o) \preceq \tau_p(i)$. The atom consisting of predicate p and symbols $x_1, \dots, x_{ar(p)}$ is denoted by $p(x_1, \dots, x_{ar(p)})$.

A *fluent* is an atom where each argument corresponds to an object of $\mathcal{O}$. A *state* is a set of fluents.

Definition 5 (Action schema)**.** An *action schema* is a tuple $a = \langle label(a), pre^+(a), pre^-(a), add(a), del(a) \rangle$ where $label(a)$ is a symbol naming the action, and $pre^+(a)$, $pre^-(a)$, $add(a)$, and $del(a)$ are sets of *atoms* instantiated with variables only. An action schema has parameters $params(a) = \langle x_1, \dots, x_{ar(a)} \rangle$, an ordered sequence of variables.

Each variable x_k has a type $\tau(x_k) \in \mathcal{T}$, and an action is *type consistent* if, for all variables $x \in params(a)$, every occurrence of x at position i of some predicate p is such that $\tau_p(i)$ is a subtype of $\tau(x)$. We will often abuse the notation and refer to an action schema by its label.

Definition 6 (Operator)**.** An *operator* o is defined analogously to an action schema, except that the sets $pre^+(o)$, $pre^-(o)$, $add(o)$, and $del(o)$ are sets of fluents. Similarly to action schemas, an operator is *type consistent* if, for all variables $x \in params(o)$, every occurrence of x at position i of some predicate p is such that $\tau_p(i)$ is a subtype of $\tau(x)$.

Definition 7 (PDDL planning problem)**.** A PDDL planning problem is a pair $\Pi = \langle \mathcal{D}, \mathcal{I} \rangle$ where $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{T} \rangle$ is the *domain* and $\mathcal{I} = \langle \mathcal{O}, \mathcal{H}, I, G \rangle$ is the *instance*. The domain $\mathcal{D}$ consists of a set $\mathcal{P}$ of predicates, a set of action schemas $\mathcal{A}$, and a type hierarchy $\mathcal{T}$. The instance $\mathcal{I}$ consists of a set of objects $\mathcal{O}$ and an associated type hierarchy $\mathcal{H}$, as well as two states, I and G, which are the *initial state* and the *goal condition*, respectively.

An operator o is *applicable* in a state s if $pre^+(o) \subseteq s$ and $pre^-(o) \cap s = \varnothing$. The state that results from the application of o in s is $s[o] := (s \setminus del(o)) \cup add(o)$.

A sequence of operators $o_1, \dots, o_n$ is called a *plan* for some planning task Π if there exists a sequence of states $s_0, \dots, s_n$ where $s_0 = I$, $G \subseteq s_n$, and for all $i \in \{1, \dots, n\}$, $s_i = s_{i-1}[o_i]$ and o_i is applicable in s_{i-1}. Such a sequence of states (which is unique for each plan) is called a *trajectory*.

If for all trajectories of a given planning problem $\Pi = \langle \mathcal{D}, \mathcal{I} \rangle$ a fluent f is either in all states or in no state, then f is *static* in Π. Let $p \in \mathcal{P}$ be a predicate. If, for all instances $\mathcal{I}$ and every fluent f whose predicate is p, f is static, then p is *static* for domain $\mathcal{D}$. When $\mathcal{P}$ is the set of predicates of $\mathcal{D}$, the set of static predicates is denoted $\mathcal{P}^S$, and we denote by $\mathcal{P}^D = \mathcal{P} \setminus \mathcal{P}^S$ the set of *dynamic* predicates.

Boolean Satisfiability Problem. Let Var be a set of propositional variables. The Boolean satisfiability problem (SAT) consists in finding a valuation that satisfies a propositional formula ϕ. Propositional formulas are defined as follows, where $x \in$ Var is a propositional variable:

$$\phi := \top \mid \bot \mid x \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi$$

3 Domain Model Acquisition

A domain model acquisition (DMA) problem consists of synthesizing a model $\mathcal{D}$ for a planning domain from a set of input traces. Traces are sequences of states and operators in which some required information may be missing. We define two domain model acquisition problems. Both consist of learning a representation of the actions of the domain, but they differ in the information provided in the traces.

3.1 Symbolic Domain Model Acquisition

The *symbolic* DMA problem consists of learning a model from traces in which states are fully known and represented as sets of fluents, while only the names of the operators are known. We adopt this setting from (Balyo et al. 2024). More specifically, the input is a sequence of traces, where the i-th trace of length n^i is $t^i = \langle s^i_0, \ell^i_1, s^i_1, \ell^i_2, s^i_2, \dots, \ell^i_{n^i}, s^i_{n^i} \rangle$, each s^i_k is a set of fluents, and each ℓ^i_k is an action label, together with the type hierarchy $\mathcal{H}$. The set of transitions associated with all traces is then $\mathcal{R} = \{\langle s^i_{k-1}, \ell^i_k, s^i_k \rangle \mid i \in \{1, \dots, m\}, k \in \{1, \dots, n^i\}\}$, where each transition $R = \langle s_b, \ell, s_a \rangle$ consists of a pre-state s_b, an action label ℓ, and a post-state s_a. We assume that s_b is the state *before* applying the action labeled ℓ, and that *afterwards* the resulting state is $s_a = s_b[\ell]$.

3.2 Subsymbolic Domain Model Acquisition

In this work, we address a generalization of the symbolic problem in which states are not fully known, but represented by bit vectors. More precisely, the problem is as follows:

Problem: Subsymbolic DMA Problem
Input:

- a sequence of traces where the i-th trace t^i is $\langle \overline{s}^i_0, \ell^i_1, \overline{s}^i_1, \ell^i_2, \overline{s}^i_2, \ldots, \ell^i_n, \overline{s}^i_n \rangle$, each $\overline{s}^i_k$ is a binary vector of size N^i representing an obfuscated state, and each ℓ^i_k is an action label;
- a type hierarchy $\mathcal{H}$;
- a set of objects $\mathcal{O}^i$ and their types associated with each trace t^i;
- a set $\mathcal{P}^D$ of predicates (assumed to be dynamic in the output domain), as well as their number of arguments, and the types of their arguments;
- for each action a expected in the domain, a subset of its parameters $\mathit{params}^{\text{eff}}(a) \subseteq \mathit{params}(a)$ and the type $\tau(x)$ of each parameter $x \in \mathit{params}^{\text{eff}}(a)$.

Output:

- a set of action schemas $\mathcal{A}$, compatible with a planning domain $\mathcal{D}$ consistent with the input;
- for each trace t^i, a mapping $\sigma^i : \{1, \ldots, N^i\} \to \mathcal{F}$, where $\mathcal{F}$ is the set of fluents built over $\mathcal{P}^D$ and $\mathcal{O}^i$.

Note that, for a given action schema a, not all of its parameters are known. Instead, only the subset $\mathit{params}^{\text{eff}}(a) \subseteq \mathit{params}(a)$ of parameters expected to appear in an effect atom of a is provided.

In the subsymbolic DMA problem, states are represented as binary vectors in which each position $b \in \{1, \ldots, N^i\}$ (called a *bit*) corresponds to an undetermined fluent in trace t^i. The binary value at each position b indicates whether the underlying fluent belongs to the state. The problem therefore requires discovering a bit-to-fluent mapping σ^i for each trace t^i while simultaneously learning the action model.

Example 1. Consider a simple scenario where a robotic arm can pick up and stack blocks, as long as it is powered by some energy source.

In the **symbolic setting**, states are fully observable sets of fluents. For instance, a state can be $s_b = \{\text{on}(A, B), \text{on}(B, \text{table}), \text{handempty}, \text{powered}(E)\}$. After executing action $\text{pickup}(A, B, E)$ (which picks up A from B using energy source E), the resulting state would be $s_a = \{\text{holding}(A), \text{on}(B, \text{table}), \text{powered}(E)\}$. The symbolic problem thus consists of learning action schemas from transitions of the form $\langle s_b, \text{pickup}, s_a \rangle$, where only the action label is provided.

In the **subsymbolic setting**, the same scenario is represented using bit vectors. While each bit corresponds to a fluent, the correspondence is initially unknown. For example, if the fluents are ordered as $\langle \text{on}(A, B), \text{on}(B, \text{table}), \text{handempty}, \text{holding}(A), \text{powered}(E) \rangle$, then state s_b is encoded as binary vector $\overline{s}_b = [1, 1, 1, 0, 1]$ and state s_a as $\overline{s}_a = [0, 1, 0, 1, 1]$. In addition to the label of the action, the number and types of the parameters appearing in effect atoms are also known. Other parameters, such as E here, are omitted. This design reflects the assumption that parameters appearing in observable effects can be directly determined from state changes (e.g., which blocks are manipulated), while parameters appearing only in preconditions must be inferred by the learning algorithm. The subsymbolic problem thus consists of learning action schemas from transitions of the form $\langle \overline{s}_b, \text{pickup}(A, B), \overline{s}_a \rangle$. In addition, a mapping from bits to fluents has to be found to reconstruct the concrete states s_b and s_a from $\overline{s}_b$ and $\overline{s}_a$.

Algorithm 1: BITL1: Learn Action Models from Binary States

```
1:  Phase 1.1: Learn effects and bit-to-fluent mappings
2:  Φ ← ∅
3:  for each ⟨s̄_b, ℓ, s̄_a⟩ ∈ R do
4:      for each changed bit b do
5:          Φ ← Φ ∧ EFFECTFORMULA(t^i, b, R)
6:  I ← FINDSATASSIGNMENT(Φ)
7:  for each action label ℓ do
8:      add(ℓ), del(ℓ) ← LEARNEFF(ℓ, R, I)

9:  Phase 1.2: Obtain symbolic representation of states
10: R^symb ← ∅
11: for each ⟨s̄_b, ℓ, s̄_a⟩ ∈ R of trace t^i do
12:     σ^i ← EXTRACTPARAMMAPPING(I)
13:     R^symb ← R^symb ∪ ⟨σ^i(s̄_b), ℓ, σ^i(s̄_a)⟩

14: Phase 2: Learn preconditions
15: for each action label ℓ do
16:     pre⁺(ℓ), pre⁻(ℓ) ← LEARNPRE(ℓ, R^symb, I)
```

4 BITL1: From Bits to Domain Models

We now present our approach to solving the subsymbolic DMA problem. Our algorithm, called BITL1, generalizes the L1 algorithm (Balyo et al. 2024), which solves the *symbolic* DMA problem. Our main contribution is to adapt L1 so that it can handle states represented by binary vectors and recover a mapping from bits to fluents.

Algorithm 1 provides an overview of BITL1, which consists of two main phases. First, the algorithm simultaneously learns the effects of the actions found in the traces, and a bit-to-fluent mapping (Lines 1–13). We compile this subproblem into an instance of the Boolean satisfiability problem (SAT). Second, the algorithm learns the preconditions of the actions (Lines 14–16). We describe these two phases in turn.

4.1 Learning Effects via SAT Encoding

Symbolic Case. We begin by describing the SAT encoding used for learning effects in the symbolic case, as originally introduced by Balyo et al. (2024). This encoding forms the foundation for our extension to binary representations. Since the symbolic problem has slightly different assumptions from the subsymbolic one, we directly present our adapted version of the constraints. The main difference lies

in the fact that we know the number and types of the parameters $x_1, \ldots, x_K$ in the effects of each action a.

For each transition $R = \langle s_b, \ell, s_a \rangle$ (Lines 3–5), the algorithm generates a SAT formula that ensures that action ℓ's effects are consistent with the observed state changes. Recall that in this setting, the objects associated with the action parameters are not known and must be determined. We thus introduce propositional variables of the form $\text{bind}(R, x_k, o)$, which state that in transition R, the action labeled ℓ has its parameter x_k bound to object o. This allows us to connect the lifted effects of the learned action with the concrete objects involved in the transition. We do so through formulas of the form below, where $\vec{x}$ is a vector of parameters of size $|\vec{x}|$ and $\vec{o}$ is a vector of objects of the same size.

$$\textsc{BindParams}(R, \vec{x}, \vec{o}) := \bigwedge_{k=1\ldots|\vec{x}|} \text{bind}(R, x_k, o_k) \tag{1}$$

Note that the vector of parameters $\vec{x}$ and the vector of objects $\vec{o}$ must be type consistent, i.e., pairwise compatible. More formally, $\vec{x}$ and $\vec{o}$ are type consistent if, for all $k \leq |\vec{x}|$, $\tau(o_k) \preceq \tau(x_k)$. We denote this by $\vec{o} \preceq \vec{x}$.

Similarly, for a given predicate p, $\vec{x}$ is type consistent with p if, for all $k \leq ar(p)$, we have $\tau(x_k) \preceq \tau_p(k)$, denoted by $\vec{x} \preceq p$. Suppose that a fluent $f = p(o_1, \ldots, o_{ar(p)})$ is established in R, i.e., $f \in s_a \backslash s_b$. Then p appears in the add effects of ℓ, as some atom $p(\vec{x})$, with $\vec{x} = \langle x_1, \ldots, x_{ar(p)} \rangle$. To enforce this, we introduce propositional variables of the form $\text{add}(\ell, p, \vec{x})$, which state that $p(\vec{x}) \in \mathit{add}(\ell)$. Since assigning the parameters of the action to the atom must be done in a type consistent way, we introduce the following constraint, that allows only consistent assignments:

$$\bigvee_{\substack{\vec{x} \text{ s.t.} \\ \vec{x} \preceq p}} \bigvee_{\substack{\vec{o} \text{ s.t.} \\ \vec{o} \preceq \vec{x}}} \text{add}(\ell, p, \vec{x}) \wedge \textsc{BindParams}(R, \vec{x}, \vec{o}) \tag{2}$$

When an atom $f = p(\vec{x})$ is deleted in the transition, we add a constraint similar to Equation 2, this time using variables $\text{del}(\ell, p, \vec{x})$, which state that $p(\vec{x}) \in \mathit{del}(\ell)$. In addition, we also forbid a parameter to be bound to more than one object in the transition. These constraints are used to extract the effects in Line 8.

Subsymbolic Case. In the subsymbolic setting, the fluents involved in transitions are obfuscated, and we do not know which predicates or objects are involved in each change. This information is critical in the symbolic case above. Here, we only know which bits changed. We next show how to adapt Equation 2 so that it learns, for each trace t^i, a bit-to-fluent assignment σ^i at the same time as the action model.

To this end, we introduce variables of the form $\text{assign}(i, b, f)$, which state that in trace i, the b-th bit of the binary vectors representing states should be mapped to fluent f. In other words, this means that $\sigma^i(b) = f$. The mapping σ^i for each trace t^i is then obtained by the function $\textsc{ExtractParamMapping}(I)$ in Line 9 of Algorithm 1.

For a given transition $R = \langle \bar{s}_b, \ell, \bar{s}_a \rangle$ where a bit changes its value from 0 to 1 (processed in Lines 3–5), the following SAT formula considers all possible fluents $p(\vec{o})$ that could be mapped to that bit. Once the candidate fluent has been chosen, the rest of the formula behaves like Equation 2 and enforces type consistency between the fluent, the action parameters, and the objects involved in the transition.

$$\begin{aligned} &\textsc{EffectFormula}(t^i, b, R) := \\ &\bigvee_{p \in \mathcal{P}^D} \bigvee_{\vec{o} \preceq p} \Bigg(\text{assign}(i, b, p(\vec{o})) \wedge \\ &\qquad \bigvee_{\substack{\vec{x} \text{ s. t.} \\ \vec{o} \preceq \vec{x} \preceq p}} \text{add}(\ell, p, \vec{x}) \wedge \textsc{BindParams}(R, \vec{x}, \vec{o}) \Bigg) \end{aligned} \tag{3}$$

As in the symbolic case, a similar formula is obtained by substituting $\text{add}(\ell, p, \vec{x})$ with $\text{del}(\ell, p, \vec{x})$ when a bit changes its value from 1 to 0. By adding one such formula for each transition R and for each bit b of each trace t^i (Lines 3–5), we ensure that the learned bit-to-fluent mappings σ^i and the learned action model are consistent with the input traces. From a model I of the resulting formula (Line 6), we obtain the action effects. This is summarized in Lines 6–8, where the function $\textsc{LearnEff}(\ell, \mathcal{R}, I)$ extracts the effects.

4.2 Learning Preconditions

At this point, the action effects have been learned, the parameters of the action schemas are bound to objects in each transition, and we have identified the fluents involved in each state. We can now complete the action model by learning the preconditions of the actions. Balyo et al. (2024) only propose a method for learning positive preconditions; we extend their approach to negative preconditions.

Positive Preconditions. From the previous phase (Lines 1–13), we can extract, for each transition $R = \langle s_b, \ell, s_a \rangle$, a partial function $u_R : \mathit{params}(\ell) \to \mathcal{O}^i$ from the parameters of ℓ to the objects found in the trace t^i. Indeed, it suffices to set $u_R(x) := o$ iff $\text{bind}(R, x, o)$ is true in the interpretation.

We then identify which fluents are candidates for appearing in the precondition of ℓ. To this end, we consider all fluents of s_b that only involve objects found in the image of u_R. These fluents hold before the execution of ℓ, and their arguments match the objects bound to the action parameters. We *lift* each such fluent $p(o_1, \ldots, o_{ar(p)})$ by replacing its objects with the associated action parameters, thus obtaining $p(u_R^{-1}(o_1), \ldots, u_R^{-1}(o_{ar(p)}))$. As a shorthand, when $\vec{o}$ is the vector of objects of the predicate, we let $p(u_R^{-1}(\vec{o}))$ denote the lifted fluent.

To obtain the preconditions for an action ℓ, we take the smallest set of fluents that appear consistently across all transitions involving ℓ. More formally, we have:

$$\mathit{pre}^+(\ell) = \bigcap_{\substack{R \in \mathcal{R} \\ R = (s_b, \ell, s_a)}} \left\{ p(u_R^{-1}(\vec{o})) \mid \begin{array}{l} p(\vec{o}) \in s_b \text{ and} \\ \vec{o} \subseteq \text{Im}(u_R) \end{array} \right\}$$

Negative Preconditions. Negative preconditions are learned in a very similar way. The only adaptation is to consider, for each transition, fluents that do *not* appear in the pre-state. This leads to the following set of fluents:

$$pre^{-}(\ell) = \bigcap_{\substack{R \in \mathcal{R} \\ R=(s_b,\ell,s_a)}} \left\{ p(u_R^{-1}(\vec{o})) \mid \begin{array}{l} p(\vec{o}) \notin s_b \text{ and} \\ \vec{o} \subseteq \mathrm{Im}(u_R) \end{array} \right\}$$

In Algorithm 1, the learning of preconditions is summarized in Lines 14–16 by function LEARNPRE($\ell, \mathcal{R}^{\text{symb}}, I$).

5 Experiments

We evaluate our algorithm, BITL1, on a set of standard planning benchmarks from the International Planning Competition (IPC). We adopt the benchmark set of Balyo et al. (2024), consisting of 16 STRIPS domains and two domains, Termes and Tidybot, that feature negative preconditions.

To adapt the benchmark set to our setting, we generate plans for several instances of each domain. We then convert them into traces with symbolic states to obtain suitable inputs for the symbolic DMA problem. The same traces are subsequently converted into subsymbolic traces by encoding states as binary vectors. Table 1 shows relevant properties of the benchmark set used in our experiments.

We run our evaluation on a machine with an Intel Core i5-1135G7 (11th Gen) processor running at 2.40GHz and with 16 GiB of RAM, under Ubuntu 24.04.3 LTS. The implementation uses Python 3.12 with the PySAT library (Ignatiev, Morgado, and Marques-Silva 2018) for SAT solving and the Glucose SAT solver (Audemard and Simon 2018). The learned action models are written in PDDL format using the Unified Planning library (Micheli et al. 2025). Our code and experiment data are available online (Haratian et al. 2026).

Baseline. To the best of our knowledge, we are the first to tackle the subsymbolic DMA problem, so there are no existing approaches to compare against directly. However, since the subsymbolic DMA problem generalizes the symbolic DMA problem, we compare our results with those of the symbolic DMA algorithm L1 by Balyo et al. (2024). Since the code is not publicly available, we reimplemented L1 in Python using PySAT, consistent with the original implementation. We compare the action models learned by BITL1 and L1 against the original benchmark domains.

Running Time. For most domains, our algorithm finds an action model in under 5 minutes. The only exceptions are Parking, Sokoban, and Visitall, which require between 30 minutes and 2 hours. For two domains, Thoughtful and Pegsol, we fail to synthesize an action model within the time limit of 2 hours because the generated SAT formulas contain too many constraints.

Results Overview. Table 2 provides an overview of how closely the learned action models match the original domains.

5.1 Learned Effects

L1 achieves state-of-the-art results in learning effects from traces, and BITL1 matches or outperforms L1 in 10 of the 16 domains for which it learns a model. In two additional domains, Storage and Childsnack, it also comes close to the model learned by L1. This is notable because BITL1 receives more obfuscated information than L1.

	Traces		Maximum Arity			Objects
Domain	**Tr**	**Obs**	**#P**	$\mathbf{M_P}$	$\mathbf{M_A}$	**Min-Max**
Barman	4	234	15	2	6	21-23
Childsnack	4	181	13	2	4	31-71
Elevators	6	142	8	2	5	16-20
Floortile	2	80	10	2	4	19
Hanoi	1	7	3	2	3	6
Nomystery	3	41	6	3	6	45-116
Parking	4	168	5	2	3	19
Pegsol	4	93	5	3	3	33
Rovers	3	30	25	3	6	13-16
Scanalyzer	4	61	6	4	8	8-16
Sokoban	4	353	6	3	6	57-129
Storage	4	17	8	2	5	7-12
Termes	4	548	6	2	4	16-17
Thoughtful	5	617	18	2	7	71-82
Tidybot	4	229	24	3	9	30
TPP	4	38	7	3	7	6-9
Transport	4	91	5	2	5	16-28
Visitall	4	404	3	2	2	25-100

Table 1: Details of the benchmark domains used in our experiments. The first two columns show the number of trajectories (Tr) and total number of state transitions (Obs). The next three columns show the number of predicates (#P), maximum arity of predicates (M_P), and maximum arity of actions (M_A). The last column shows the range of objects across different problem instances.

5.2 Learned Preconditions

Both L1 and BITL1 learn preconditions using the same subroutine, described in Section 4.2. The main difference is that, in the symbolic case, L1 learns preconditions independently of the effect-learning phase. In the subsymbolic setting, by contrast, the fluents in each state must first be determined before preconditions can be learned. As a consequence, the quality of the preconditions learned by BITL1 depends heavily on the quality of the bit-to-fluent mappings obtained in the first phase of the algorithm.

In the symbolic setting, static fluents (and thus static predicates) can be identified from the states of the traces. This is not the case in the subsymbolic setting, where no information about static fluents can be extracted from the traces. For this reason, we assume that static predicates are not part of the input. Even if they were, any effort to determine static fluents would be fruitless: since a static bit b never changes during any transition, it is impossible to determine which objects are involved in the underlying fluent, which is central to our approach for learning effects. Thus, all possible static fluents are equally likely candidates for the actual fluent underlying a static bit, making them impossible to distinguish from one another without additional information.

Static fluents sometimes represent a substantial part of the preconditions of actions. For example, in the Tidybot domain, there are on average 4 static preconditions per action, which BITL1 cannot learn. To provide further insight into

	L1					BITL1						Bit Mapping Avg (%)	
Domain	**+NP**	**−P**	**+P**	**−E**	**+E**	**+NP**	**−P w/o SP**	**+P**	**−E**	**+E**	**−P w/ SP**	**w/o Sym**	**w/ Sym**
Barman	42	**7**	**11**	**0**	**0**	**30**	20	14	16	16	28	28.68	32.86
Childsnack	**13**	4	**1**	**0**	**0**	**13**	**3**	**1**	1	1	9	14.94	57.19
Elevators	9	**0**	13	**0**	**0**	**8**	2	**2**	**0**	**0**	13	48.27	51.16
Floortile	**9**	4	7	**0**	**0**	10	**1**	**0**	**0**	**0**	8	31.88	45.50
Hanoi	**2**	**0**	1	**0**	**0**	**2**	1	**0**	**0**	**0**	2	100.00	100.00
Nomystery	**4**	2	3	**0**	**0**	**4**	**0**	**2**	**0**	**0**	3	12.00	32.04
Parking	8	**0**	3	**0**	**0**	**7**	4	**1**	**0**	**0**	4	98.59	100.00
Pegsol	10	0	3	0	0	—	—	—	—	—	—	—	—
Rovers	16	21	18	**12**	**0**	**11**	**15**	**7**	25	13	36	45.45	45.45
Scanalyzer	**29**	**1**	**6**	**0**	**0**	**29**	2	10	9	9	6	38.96	71.66
Sokoban	**11**	5	**0**	**0**	**0**	**11**	**0**	**0**	**0**	**0**	7	21.43	21.43
Storage	**12**	1	8	**0**	**0**	**12**	**5**	**9**	5	5	10	61.38	67.78
Termes	6	18	2	**0**	**0**	**5**	**9**	**0**	**0**	**0**	21	46.95	46.95
Thoughtful	108	34	12	0	0	—	—	—	—	—	—	—	—
Tidybot	57	59	39	4	4	**39**	**7**	**25**	**3**	**3**	74	70.19	70.19
TPP	**10**	**3**	**7**	**5**	**5**	13	6	10	6	6	13	72.48	73.95
Transport	**5**	**0**	**0**	**0**	**0**	6	2	1	**0**	**0**	5	42.93	58.61
Visitall	**1**	**0**	2	**0**	**0**	**1**	**0**	**0**	**0**	**0**	1	2.69	97.93

Table 2: Comparison of L1 and BITL1. For each domain, the table shows the number of superfluous $(+)$ and missing $(-)$ learned action model components: NP (negative preconditions), P (positive preconditions), and E (effects). For BITL1, an additional column shows −P w/ SP (missing positive preconditions when static predicates are considered). Lower values are better and shown in bold with gray background. The rightmost two columns show the average bit-to-fact mapping accuracy (%) for BITL1: w/o Sym requires exact object matches, while w/ Sym reports accuracy when mapped objects are allowed to be symmetric with the ground truth. Domains unsolved are marked with –.

the learning abilities of our algorithm, we present in Table 2 the amount of missing preconditions with and without counting static predicates for BITL1. Note that L1, working symbolically, still tries to learn static fluents in the actions.

BITL1 learns fewer redundant preconditions than L1, both positive and negative. Indeed, static predicates cause L1 to have many redundant (negative) preconditions, as the objects in the bindings of each transition may match one of the static predicates, causing L1 to add them as preconditions. Conversely, the actions learned by our approach miss more preconditions than L1. Nonetheless, Table 2 shows that our approach achieves overall results similar to the baseline, indicating that the learned bit-to-fluent mappings are sufficiently accurate for the precondition-learning routine.

Our extension to learning negative preconditions is viable, since the number of superfluous learned negative preconditions is generally reasonable. Note that some learned negative preconditions, although not part of the original domain, may still be consistent with it. For instance, any fluent that is mutually exclusive with a positive precondition can be added to the negative preconditions of the action. More generally, extending the routine to learn negative preconditions proves fruitful: for example, in the Tidybot domain, BITL1 learns 21 negative preconditions and only 3 negative preconditions are missing from the learned action models.

5.3 Bit-to-Fact Mapping Accuracy

Both the effects and the preconditions learned by our algorithm rely heavily on the bit-to-fluent mapping. We therefore evaluate the accuracy of these mappings by considering every assignment $\sigma(b) = f$ from bit b to fluent f. The assignment is correct if $\sigma(b)$ and f are identical, meaning that the predicate names and object names are the same. Under this criterion, we report the accuracy of the mappings in the penultimate column of Table 2.

Note, however, that this syntactic criterion is overly strict, as some objects can play the same role despite having different names. Such objects are *symmetric*, and their names can be swapped in the bit-to-fluent assignments without changing the semantics of the domain. To illustrate this, consider a BlocksWorld instance with two blocks, A and B, initially on the table. In the subsymbolic setting, since concrete names are not part of the trace, a sequence of actions placing A on B is indistinguishable from a sequence of actions placing B on A. Still, this does not prevent BITL1 from learning accurate action models.

The last column in Table 2 shows the accuracy of the bit-to-fluent mappings when symmetries are taken into account. Here, a mapping $\sigma(b) = f$ is correct if $\sigma(b)$ and f share the same predicate, and the objects of $\sigma(b)$ and f are symmetric. Note that this only gives an upper bound on the actual accuracy of the mapping, since swapping two symmetric objects in a fluent does not always yield a globally consistent mapping (for instance, if object o_1 in f is replaced by object o_2 in $\sigma(b)$, but o_1 in f' is replaced by o_3 in $\sigma(b')$). We obtained the symmetry information for each instance using the tool from (Sievers et al. 2019). Symmetries are computed without stabilizing the initial and the goal states.

6 Discussion

In this section, we analyze the bit-to-fluent assignments made by BITL1 across the benchmark domains, categorizing them according to their relationship to the ground-truth mappings. We focus on assignments that differ from the exact ground truth, excluding those that are either perfectly correct or involve symmetric objects.

We categorize the assignments based on two dimensions: whether they occur within the same predicate or across different predicates, and whether the types of the objects in the assigned fluent match or differ. We also consider whether the assigned type vector is a subtype of the correct one. Additionally, we separately track assignments involving symmetric objects, even though some may in fact be correct. This results in five categories of imperfect assignments:

- **Symmetric objects**: The algorithm assigns a bit to a fluent with the correct predicate name and types, but with objects symmetric to those in the ground-truth fluent. These assignments are often functionally equivalent to the correct one.
- **Same predicate, same types**: The algorithm assigns a bit to a fluent with different object names, but with the correct predicate name and object types. While not matching the exact ground-truth objects, these assignments preserve the predicate structure.
- **Different predicate, same types**: The algorithm assigns a bit to a fluent with a different predicate name but with the same type vector as the correct fluent. For example, assigning `(clean shot1)` instead of `(empty shot2)` in the Barman domain.
- **Different predicate, different types**: The algorithm assigns a bit to a fluent with a different predicate name and same length of type vector as the correct fluent, but the type vectors are different. For example, assigning `(at_kitchen_bread bread2)` instead of `(notexist sandw4)` in the Childsnack domain.
- **Different predicate, varying type vectors (subset)**: The algorithm assigns a bit to a fluent with a different predicate name and a type vector with different length than the correct fluent, but the type vector of the assigned fluent is a subset of the correct type vector. For example, assigning `(contains shot1 cocktail1)` instead of `(empty shot1)` in Barman. Notably, across all domains except Rovers, when the learned predicate has a different arity than the correct predicate, the learned predicate's arguments form a superset of the actual predicate's arguments.

The distribution of these assignment categories across benchmark domains is summarized in Table 3. Note that a small percentage of assignments (averaging 0.1% across domains) fall into the "same predicate, different types" category, primarily in the Barman domain due to imprecise modeling of object types in that domain specification.

An important observation is the presence of symmetric-object assignments, which average 14.5% across all domains. These assignments are not necessarily errors in the strict sense, as they map bits to functionally equivalent fluents that differ only in symmetric object names. Domains such as Visitall (95.2%), Childsnack (34.8%), and Scanalyzer (32.7%) show high percentages of symmetric assignments, indicating strong object symmetry in these domains.

Beyond symmetric assignments, assignments within the same predicate with matching types dominate in most domains (28.7% on average). This suggests that although the mapping of bits to fluents is not always exact, the algorithm generally identifies the correct predicate and type context for the parameters.

Importantly, the first two categories (symmetric objects and same predicate with same types) together account for 43.2% of all assignments on average. These categories are less problematic for learning action models because they preserve the correct predicate structure. When lifting the learned fluents to extract action schemas, the correct effects are often still recovered because the predicate names and type signatures are correct, even if the specific object names in the bindings differ. In particular, what matters for learning correct effects is the consistency of the action arguments in the bindings and how they are used in the lifting step.

In contrast, assignments in the remaining categories (different predicate with same types at 4.8%, different types at 2.1%, and varying type vectors at 8.9%, totaling approximately 16% of assignments) are the primary source of errors in the learned action models, as they lead to incorrect effects being included in the learned models. Assignments in these categories are harder to avoid, especially since some groups of predicates are closely related to each other and are used in the same set of actions. For example, in Barman the predicates `(empty ?x - shaker)`, `(clean ?x - shaker)`, `(shaked ?x - shaker)`, and `(unshaked ?x - shaker)` are all associated with the same object type and are used in the same set of actions. This makes them difficult to distinguish based on the traces alone, especially when the objects involved in a transition are not known.

Learning the transition model is the most challenging part of the problem, while inferring types, objects, and predicates can be considered a solved problem in many practical scenarios using modern vision and language models. Providing more types, objects, or predicates than necessary is not problematic from a computational standpoint: the algorithm only uses those that are required, and our results show that in most cases the problem remains solvable in reasonable time. Thus, our approach is robust to imperfect input information.

From a practical perspective, some error categories are easier to correct than others. Type-violating errors are generally easier to detect and potentially avoid through stricter type checking during learning. In contrast, same-predicate, same-type errors are inherently more challenging as they require distinguishing between parameters that are structurally identical from the perspective of types and predicates. These errors could be mitigated by using additional information, such as semantic annotations or more comprehensive trace coverage that exercises all possible parameter bindings.

	Symmetric Objects (%)	Same pred, same types (%)	Diff pred, same types (%)	Diff pred, diff types (%)	Diff pred, varying type vectors (%)
Barman	4.2	15.8	26.8	2.5	22.0
Childsnack	34.8	30.8	5.9	27.4	0.0
Elevators	2.9	50.3	0.0	1.5	0.0
Floortile	16.7	51.4	0.0	0.0	0.0
Hanoi	0.0	0.0	0.0	0.0	0.0
Nomystery	20.0	62.2	11.5	0.0	0.0
Parking	1.4	1.4	0.0	0.0	2.8
Rovers	0.0	7.1	23.6	1.9	41.7
Scanalyzer	32.7	23.8	0.0	0.0	10.0
Sokoban	0.0	76.25	0.0	0.0	2.3
Storage	6.4	6.4	6.4	0.0	35.8
Termes	0.0	53.0	0.0	0.0	0.0
Tidybot	0.0	18.2	0.0	0.0	11.6
TPP	1.4	11.1	0.0	0.0	16.4
Transport	15.7	51.1	0.0	0.0	0.0
Visitall	95.2	0.0	2.1	0.0	0.0
Average	**14.5**	**28.7**	**4.8**	**2.1**	**8.9**

Table 3: Bit-to-fluents assignment breakdown by domain (in %). For each domain, we compute the percentage of assignments in each category by first calculating the ratio for each instance, then averaging across all instances. The first two categories represent correct or equivalent assignments.

7 Related Work

Automated domain synthesis has developed into a mature research area with numerous methods for automated action model acquisition (Arora et al. 2018; Callanan et al. 2022). The existing literature spans several categories based on their fundamental assumptions: full observability (e.g., OBSERVER (Wang 1996), OpMaker (McCluskey, Richardson, and Simpson 2002), EXPO (Gil 1994), TRAIL (Benson 1997), OLAM (Lamanna et al. 2021)), partial observability and noisy states (e.g., ARMS (Yang, Wu, and Jiang 2007), LAMP (Zhuo et al. 2010), AMAN (Zhuo and Kambhampati 2013), LOUGA (Kučera and Barták 2018), ALICE (Mourao, Petrick, and Steedman 2009), AMLSI (Grand 2022)), graph-based action modeling (e.g., LOCM (Cresswell, McCluskey, and West 2013), LOCM2 (Cresswell and Gregory 2011), LOP (Gregory and Cresswell 2015), LSSS (Bonet and Geffner 2020)), non-deterministic modeling (LPROB (Pasula, Zettlemoyer, and Kaelbling 2007)), and minimal assumptions (FAMA (Aineto, Jiménez Celorrio, and Onaindia 2019)).

Safe model-free planning approaches (Stern and Juba 2017) guarantee that generated plans will not fail. The Safe Action Model (SAM) learning family of approaches leverages this for lifted domain model acquisition (Juba, Le, and Stern 2021), with extensions for numeric fluents (Mordoch, Juba, and Stern 2023), probabilistic models (Juba and Stern 2022), and partial observability (Le, Juba, and Stern 2024).

A line of research that is particularly relevant to our work focuses on learning action models from action traces alone, without requiring state information. The SIFT algorithm (Gösgens, Jansen, and Geffner 2025) introduces features and action patterns to represent assumptions about which predicates are affected by which actions, testing consistency efficiently by extracting pattern constraints from traces and reducing the consistency check to 2-CNF satisfiability. A follow-up approach extends SIFT to handle incomplete action information (Jansen, Gösgens, and Geffner 2025). While SIFT demonstrates strong theoretical properties and scalability, it operates within the symbolic planning framework, requiring actions to be represented symbolically with identifiable arguments. Our work differs fundamentally: we learn from subsymbolic representations where states are raw observations (e.g., images or sensor data) and actions may lack explicit symbolic structure. Moreover, we integrate both action and state information rather than relying solely on action traces, bridging the gap between subsymbolic perceptions and symbolic action models.

8 Conclusions and Future Work

We presented BITL1, an algorithm for domain model acquisition from subsymbolic state representations. Our SAT-based method simultaneously learns the bit-to-fluent mapping and constructs lifted action models from binary state encodings, bridging the gap between subsymbolic data collection and symbolic planning requirements.

Experiments on International Planning Competition benchmarks show that BITL1 maps subsymbolic bits to symbolic fluents with an average accuracy of 61.50% (exceeding 95% in some domains), and that the learned action models closely match those of L1 despite having access only to binary representations. While BITL1 cannot learn static predicates due to the absence of observable changes, it successfully handles dynamic predicates and demonstrates practical feasibility.

Future work includes extending the approach to continuous state representations via neural-network-based encodings, enabling direct learning from sensor data. Additionally, supporting numeric PDDL would allow learning action models with numeric fluents and effects. Obtaining object-level representations could further constrain the search space for bit-to-fluent mappings and thereby improve accuracy by focusing on predicates involving changed objects.

Acknowledgments

We thank Tomáš Balyo for his support regarding the L1 algorithm. This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

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

Arora, A.; Fiorino, H.; Pellier, D.; Métivier, M.; and Pesty, S. 2018. A review of learning planning action models. *The Knowledge Engineering Review*, 33(e20): 1–25.

Audemard, G.; and Simon, L. 2018. On the Glucose SAT Solver. *International Journal on Artificial Intelligence Tools*, 27(1): 1840001:1–1840001:25.

Balyo, T.; Suda, M.; Chrpa, L.; Šafránek, D.; Gocht, S.; Dvořák, F.; and Youngblood, G. 2024. Planning Domain Model Acquisition from State Traces without Action Parameters. In Marquis, P.; Ortiz, M.; and Pagnucco, M., eds., *Proceedings of the Twenty-First International Conference on Principles of Knowledge Representation and Reasoning (KR 2024)*, 812–822. IJCAI Organization.

Benson, S. S. 1997. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. thesis, Stanford University.

Bonet, B.; and Geffner, H. 2020. Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In De Giacomo, G., ed., *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, 2322–2329. IOS Press.

Callanan, E.; De Venezia, R.; Armstrong, V.; Paredes, A.; Chakraborti, T.; and Muise, C. 2022. MACQ: a holistic view of model acquisition techniques. *arXiv preprint arXiv:2206.06530*.

Cresswell, S.; and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 42–49. AAAI Press.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review*, 28(2): 195–213.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Machine Learning, Proceedings of the Eleventh International Conference (ICML 1994)*, 87–95. Morgan Kaufmann.

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*, volume 35, 189–197.

Grand, M. 2022. *Action Model Learning based on Regular Grammar Induction for AI Planning*. Ph.D. thesis, Université Grenoble Alpes.

Gregory, P.; and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 97–105. AAAI Press.

Haratian, A.; Lequen, A.; Gnad, D.; and Seipp, J. 2026. Code and Data for the ICAPS 2026 paper: Domain Model Acquisition From Binary Traces. https://doi.org/10.5281/zenodo.19140211.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence*, 173: 503–535.

Ignatiev, A.; Morgado, A.; and Marques-Silva, J. 2018. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT 2018)*, 428–437.

Jansen, N.; Gösgens, J.; and Geffner, H. 2025. Learning Lifted Action Models From Traces of Incomplete Actions and States. arXiv preprint arXiv:2508.21449.

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 (KR 2021)*, 379–389.

Juba, B.; and Stern, R. 2022. Learning Probably Approximately Complete and Safe Action Models for Stochastic Worlds. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022)*, 9795–9804.

Kučera, J.; and Barták, R. 2018. LOUGA: learning planning operators using genetic algorithms. In *Proceedings of the 15th Pacific Rim Knowledge Acquisition Workshop (PKAW 2018)*, volume 11016 of *LNCS*, 124–138.

Lamanna, L.; Saetti, A.; Serafini, L.; Gerevini, A. E.; and Traverso, P. 2021. Online Learning of Action Models for PDDL Planning. In Zhou, Z.-H., ed., *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*, 4112–4118. IJCAI.

Le, H. S.; Juba, B.; and Stern, R. 2024. Learning Safe Action Models with Partial Observability. In *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI 2024)*, volume 38, 20159–20167.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems (AIPS 2002)*, 121–130.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, Manipulating and Solving AI Planning Problems in Python. *SoftwareX*, 29: 102012.

Mordoch, A.; Juba, B.; and Stern, R. 2023. Learning Safe Numeric Action Models. In *Proceedings of the 37th AAAI Conference on Artificial Intelligence (AAAI 2023)*, 12079–12086.

Mourao, K.; Petrick, R.; and Steedman, M. 2009. Learning action effects in partially observable domains. In *Proceedings of the ICAPS 2009 Workshop on Planning and Learning*, 15–22.

Muise, C. 2016. Planning.Domains. In *ICAPS 2016 System Demonstrations and Exhibits*.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *J. Artif. Intell. Res.*, 29: 309–352.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019. Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 446–454. AAAI Press.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *The Knowledge Engineering Review*, 22(2): 117–134.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4405–4411.

Wang, X. 1996. *Learning planning operators by observation and practice*. Ph.D. thesis, Carnegie Mellon University.

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

Zhuo, H. H.; and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2444–2450.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning Complex Action Models with Quantifiers and Logical Implications. *Artificial Intelligence*, 174(18): 1540–1569.