In ICML 2020 Workshop on Human in the Loop Learning (HILL 2020)
Vienna, Austria (Virtual)

Learning Interpretable Models for Black-Box Agents

Pulkit Verma [1] **Siddharth Srivastava** [1]

Abstract

This paper develops a new approach for learning a STRIPS-like model of a non-stationary black-box autonomous agent that can plan and act. In this approach, the user may ask an autonomous agent a series of questions, which the agent answers truthfully. Our main contribution is an algorithm that generates an interrogation policy in the form of a contingent sequence of questions to be posed to the agent. Answers to these questions are used to learn a minimal, functionally indistinguishable class of agent models. This approach requires a minimal query-answering capability from the agent. Empirical evaluation of our approach shows that despite the intractable space of possible models, our approach can learn interpretable agent models for a class of black-box autonomous agents in a scalable manner.

1 Introduction

Growing deployment of autonomous agents leads to a pervasive problem: how would we ascertain whether an autonomous agent will be safe, reliable, or useful in a given situation? This problem becomes particularly challenging when we consider that most autonomous systems are not designed by their users; their internal software may be unavailable or difficult to understand; and they may adapt and learn from the environment where they are deployed, invalidating design-stage knowledge of agent models.

Such scenarios feature two properties that limit the applicability of existing approaches for model learning: they feature (a) *non-stationary* agents and environments, which evolve with the situation and do not present sufficient observational data for pure statistical learning and (b) *black-box* autonomous agents that may not be aware of a user's preferred model representation.

[1]School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ 85281 USA. Correspondence to: Pulkit Verma <verma.pulkit@asu.edu>.

2^{nd} *ICML Workshop on Human in the Loop Learning*, 2020.

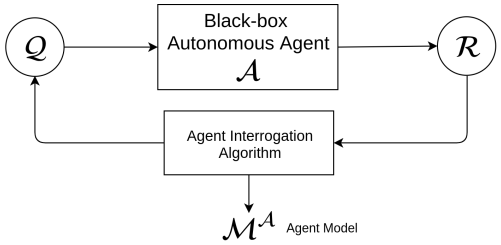

Figure 1: The agent interrogation framework

This paper presents a new approach for estimating an interpretable, relational model of a black-box autonomous agent that can plan and act, by interrogating it. Our approach is inspired by the interview process that one typically uses to determine whether a human would be qualified for a given task. Consider a situation where Hari(ette) ($\mathcal{H}$) wants their autonomous robot ($\mathcal{A}$) to clean up their lab, but s/he is unsure whether it is up to the task and wishes to estimate $\mathcal{A}$'s internal model in an interpretable representation that s/he is comfortable with (e.g., a relational STRIPS-like language (Fikes & Nilsson, 1971; Fox & Long, 2003)). Thus, $\mathcal{H}$ may ask $\mathcal{A}$ a series of questions, e.g., "What do you think will happen if you picked up bottle 1, bottle 2 and bottle 3 in succession?" Naïve approaches to the problem would require too many questions and place strong requirements on $\mathcal{A}$'s knowledge of $\mathcal{H}$'s preferred representations[1]. If this problem could be solved efficiently, lay persons would be able to efficiently determine the applicability of a wide class of adaptive, non-stationary black-box agents by interrogating them, thus improving the overall usability as well as field-debuggability of autonomous systems.

We use a rudimentary class of queries that makes our approach applicable on a broad class of agent implementations including simulator-based and analytical model-based agents: we use *plan outcome queries* that ask $\mathcal{A}$ what, according to it, would be the outcome of executing the longest executable prefix of a hypothetical plan π on a hypothetical initial state s. These are the only queries that $\mathcal{A}$ needs to support for our approach to be applicable. Fig. 1 illustrates

[1]Just 2 actions and 5 grounded propositions would yield $9^{2\times5} \sim 10^9$ possible STRIPS models – each proposition could be absent, positive or negative in the precondition and effects of each action. A query strategy that inquires about each occurrence would be not only unscalable but also inapplicable on simulator-based agents that do not know their actions' preconditions and effects.

the overall paradigm. Our approach generates a sequence of queries ($\mathcal{Q}$) depending on the agent's responses ($\mathcal{R}$) during the query process; the result of the overall interrogation process is a complete model of $\mathcal{A}$. In order to generate queries, we present a top-down process that eliminates large classes of agent-inconsistent models by computing queries that discriminate between pairs of *abstract models*. When an abstract model's answer to a query differs from the agent answer, we effectively eliminate the the entire set of possible concrete models that are refinements of this abstract model.

In developing the first steps towards this paradigm we assume that the user wishes to estimate $\mathcal{A}$'s internal model as a STRIPS-like relational model with conjunctive preconditions, add lists, and delete lists (and that the agent's model is expressible as such), although our framework can be extended to handle other types of formal domain representations. Further, we assume that the agent has functional definitions for the relations and actions in the user's vocabulary (these definitions can be programmed as Boolean functions over the state), and that it always answers truthfully.

Related Work A number of researchers have explored the problem of learning agent models from observations of its behavior (Yang et al., 2007; Cresswell et al., 2009; Cresswell & Gregory, 2011; Stern & Juba, 2017). To the best of our knowledge, ours is the first approach to address the problem of generating query strategies for inferring relational models of non-stationary black-box agents. Camacho & McIlraith (2019) recently presented an approach for learning LTL models from an agent's observed state trajectories using an Oracle with knowledge of the target LTL representation. The oracle could also generate counterexamples when the estimated model differed from the true model. Amir & Chang (2008) use logical filtering (Amir & Russell, 2003) to learn partially observable action models from action and observation traces. LOCM (Cresswell et al., 2009) and LOCM2 (Cresswell & Gregory, 2011) present another class of algorithms that use finite-state machines to create action models using plan traces. LOUGA (Kučera & Barták, 2018) uses a combination of genetic algorithm and an ad-hoc method to learn planning operators using plan traces. FAMA (Aineto et al., 2019) reduces model recognition to a planning problem, whose solution when used with a post-processing step similar to LOUGA, gives the action model. FAMA, and many such approaches, fail to handle negative literals in action's preconditions. Additionally, using post-processing step can result in failure to converge in the presence of spurious predicates in the observed plan traces. Khardon & Roth (1996) address the problem of making model-based inference faster *given a set of queries*, under the assumption that a static set of models represents the true knowledge base. Incremental Learning Model (Ng & Petrick, 2019) use reinforcement learning to learn the non-stationary model without using plan traces. Most of these learners rely on ground truth model to determine when the training may be stopped and the action model learnt is correct. So in absence of ground truth model, these methods will not know when to stop learning. While these prior approaches address the problem of learning models from training data, they do not address the problem of estimating agent models in situations where the agent or the environment are not stationary – i.e., where the agent may learn, where it may be provided with system updates, or where the environment changes such that the distribution that the training data came from is no longer representative of the deployment.

In contrast, our approach is not limited to stationary settings since it acquires current information using auto-generated queries. So every time the underlying agent model changes, it can interrogate the agent and use updated responses. Moreover, our approach does not require $\mathcal{A}$ to provide intermediate states in an execution, or to have an Oracle that could provide counterexamples or assess the correctness of the current model estimate. In contrast to approaches for white-box model-maintenance (Bryce et al., 2016), our approach does not require $\mathcal{A}$ to know about $\mathcal{H}$'s preferred representation language.

The field of active learning (Settles, 2009) addresses the related problem of selecting which data-labels to acquire for learning single-step decision-making models using statistical measures of information. But effective feature set here is the set of all possible plans, so conventional methods for evaluating information gain of each plan will be insufficient. In contrast, our approach uses a hierarchical abstraction to select questions to ask while inferring a multi-step decision-making (planning) model. Information-theoretic metrics could also be used in our approach when such information is available.

The rest of this paper is organized as follows. The next section presents some background terminology used in this paper and explains our approach. Section 3 discusses the empirical evaluation of our approach; and Section 4 highlights our main conclusions and directions for future work.

2 The Agent-Interrogation Task

We assume that $\mathcal{H}$ needs to estimate $\mathcal{A}$'s model using a STRIPS-like planning model (Fikes & Nilsson, 1971) with conditional effects (in accordance with PDDL (Fox & Long, 2003)) represented as a pair $\mathcal{M} = \langle \mathbb{P}, \mathbb{A} \rangle$, where $\mathbb{P} = \{p_1, \ldots, p_n\}$ is a finite set of predicates; $\mathbb{A} = \{a_1, \ldots, a_k\}$ is a finite set of parameterized actions. Each action $a_j \in \mathbb{A}$ is represented as a tuple $\langle header(a_j), pre(a_j), \mathit{eff}(a_j) \rangle$, where $header(a_j)$ is the action header representing action name and action parameters, the states where action a_j can

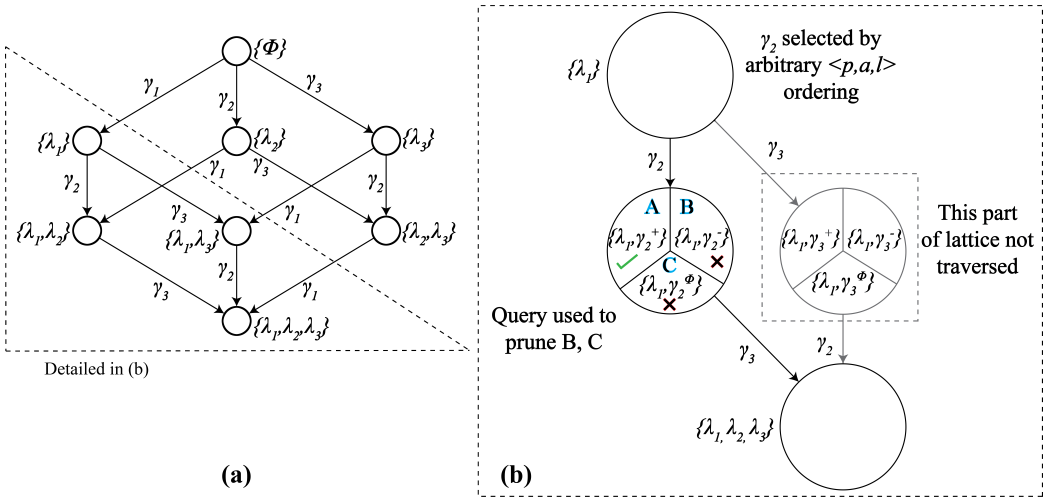

Figure 2: (a) A subset lattice created using pal-tuples; (b) Detailed view of a portion of lattice (marked in (a)) illustrating how partitions are created and pruned.

be applied are defined by preconditions $pre(a_j)$, the resulting predicates that will change to true or false are defined as effects *eff*(a_j). The effects can also be conditional, i.e. of the form $c \rightarrow$ *eff*, where c is the condition under which the effect *eff* should be applied. Each predicate can be instantiated using the parameters of an action where number of parameters are bounded by the maximum arity of the action. For example, consider the action *load_truck*$(?v1, ?v2, ?v3)$ and predicate $at(?x, ?y)$ in Logistics domain. The predicate can be instantiated using action parameters $v1$, $v2$, and $v3$ as $at(?v1, ?v1)$, $at(?v1, ?v2)$, $at(?v1, ?v3)$, $at(?v2, ?v1)$, $at(?v2, ?v2)$, $at(?v2, ?v3)$, $at(?v3, ?v1)$, $at(?v3, ?v2)$, and $at(?v3, ?v3)$. We represent set of all such possible predicates instantiated with action parameters as $\mathbb{P}^*$.

The only information $\mathcal{H}$ has is the $header(a)$ of actions $a \in \mathbb{A}$ that $\mathcal{A}$ can perform. We denote the set of headers for all actions by $\mathbb{A}_H$. As noted in the Introduction, $\mathcal{A}$ has functional definitions of the predicates (with their parameters) in $\mathcal{H}$'s vocabulary, and therefore there is sufficient information for a dialog between $\mathcal{H}$ and $\mathcal{A}$ about the outcomes of hypothetical action sequences.

We define the overall problem of agent interrogation as follows. Given a class of queries and an agent with an unknown model who can answer these queries, determine the model of the agent. More precisely, an *agent interrogation task* is defined as a tuple $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q} \rangle$ where $\mathcal{M}^{\mathcal{A}}$ is the true model of the agent (unknown to the interrogator) being interrogated, and $\mathbb{Q}$ is the class of queries that can be posed to the agent by the interrogator. Let Θ be the set of possible answers to queries. Thus, strings $\theta^* \in \Theta^*$ denote the information received by $\mathcal{H}$ at any point in the query process. *Solutions to the agent interrogation task* takes the form of a set of possible models and uses a query policy $\theta^* \rightarrow \mathbb{Q} \cup \{Stop\}$ that maps sequences of answers to the next query that the interrogator should ask. The process stops with the *Stop* query. In other words, $\forall$ answer $\theta \in \Theta$, all valid query policies map all sequences $x\theta$ to *Stop* whenever $x \in \Theta^*$ is mapped to *Stop*. This policy is computed and executed online.

Components of Agent Models In order to formulate our solution approach, we consider a model $\mathcal{M}$ to be comprised of components called *palm* tuples of the form $\lambda = \langle p, a, l, m \rangle$ where p is an instantiated predicate from the common vocabulary $\mathbb{P}^*$; a is an action from the set of parameterized actions $\mathbb{A}$, $l \in \{pre, eff\}$ and $m \in \{+, -, \emptyset\}$. For convenience, we use subscripts p, a, l or m to denote the corresponding component in a palm tuple. The presence of a palm tuple λ in a model denotes the fact that in that model, the predicate λ_p appears in an action λ_a at a location λ_l as a true (false) literal when sign λ_m is positive (negative), and is absent when $\lambda_m = \emptyset$. This allows us to define the set-minus operation $M \setminus \lambda$ on this model as removing the palm-tuple λ from the model.

We consider two palm tuples $\lambda_1 = \langle p_1, a_1, l_1, m_1 \rangle$ and $\lambda_2 = \langle p_2, a_2, l_2, m_2 \rangle$ to be *variants* of each other ($\lambda_1 \sim \lambda_2$) iff they differ only on m, i.e. $\lambda_1 \sim \lambda_2 \Leftrightarrow (\lambda_{1_p} = \lambda_{2_p}) \wedge (\lambda_{1_a} = \lambda_{2_a}) \wedge (\lambda_{1_l} = \lambda_{2_l}) \wedge (\lambda_{1_m} \neq \lambda_{2_m})$. Hence mode assignments to a *pal* tuple $\gamma = \langle p, a, l \rangle$ can result in 3 palm variants $\gamma^+ = \langle p, a, l, + \rangle$, $\gamma^- = \langle p, a, l, - \rangle$, and $\gamma^\emptyset = \langle p, a, l, \emptyset \rangle$.

Model Abstraction We are now ready to define the notion of abstractions used in our solution approach. Several approaches have explored the use of abstraction in planning (Sacerdoti, 1974; Giunchiglia & Walsh, 1992; Helmert et al., 2017; Bäckström & Jonsson, 2013; Srivastava et al., 2016). The following definition extends the concept of

predicate and propositional domain abstractions (Srivastava et al., 2016) to allow for the projection of a single λ tuple.

Definition 1. Let $\mathcal{U}$ be the set of all possible models. The *abstraction of a model* $\mathcal{M}$, on the basis of a palm tuple λ, is given by $f_\lambda : \mathcal{M} \mapsto (\mathcal{M} \setminus \lambda)$, where $f_\lambda : \mathcal{U} \to \mathcal{U}$. A set X is said to be a model abstraction of a set of models M with respect to a λ-tuple, if $X = \{f_\lambda(m) : m \in M\}$.

We also use the notation $\mathcal{M}' \sqsubset_\lambda \mathcal{M}$ to represent the situation where $f_\lambda(\mathcal{M}) = \mathcal{M}'$. We use this abstraction framework to define a subset-lattice over abstract models (Fig. 2(a)). Note that at each node we can have all possible variants of a pal tuple. For example, in the topmost node in Fig. 2(b), we can have models corresponding to γ_1^+, γ_1^-, and $\gamma_1^\emptyset$. Each node in the lattice represents a collection of possible abstract models at the same level of abstraction. As we move up in the lattice, we get more abstracted version of the models and we get more concretized models as we move down.

Definition 2. A ***model lattice*** $\mathcal{L}$ is a 5-tuple $\mathcal{L} = \langle N, E, \Gamma, \ell_N, \ell_E \rangle$, where N is a set of lattice nodes, Γ is the set of all pal tuples $\langle p, a, l \rangle$, $\ell_N : N \to 2^{2^\Lambda}$ is a node label function where $\Lambda = \Gamma \times \{+, -, \emptyset\}$, E is the set of lattice edges, and $\ell_E : E \to \Gamma$ is a function mapping edges to edge labels such that for each edge $n_i \to n_j, \ell_N(n_j) = \{\Lambda \cup \{\gamma^k\} | \Lambda \in \ell_N(n_i), \gamma = \ell_E(n_i \to n_j), k \in \{+, -, \emptyset\}\}$.

The supremum $\top$ of the lattice $\mathcal{L}$ is the most abstracted node of the lattice, whereas the infimum $\bot$ is the most concretized node. Also, a node $n \in N$ in this lattice $\mathcal{L}$ can be uniquely identified as the sequence of pal tuples that label edges leading to it from the supremum. As shown in Fig. 2(b), even though theoretically $\ell : n \mapsto 2^{2^\Lambda}$, only one of the sets is stored at each node as the others are pruned out based on $\mathcal{Q}$. Also, in these model lattices every node has an edge going out from it corresponding to each pal tuple that is not present in the paths leading to it from the most abstracted node. At any stage during the interrogation, nodes in such a lattice are used to represent the set of models that are possible given the agent's responses up to that point. At every step, our query-generation algorithm will create queries that help us determine the next descending edge to take from each lattice node.

Form of Agent Queries As discussed earlier, we pose queries to the agent and based on the responses we try to infer the agent's model. We express queries as functions mapping models to answers. More precisely, let $\mathcal{U}$ be the set of possible models and $\mathcal{R}$ a set of possible responses. A *query* $\mathcal{Q}$ is a function $\mathcal{Q} : \mathcal{U} \to \mathcal{R}$.

In this paper we utilize only one class of queries: *plan outcome queries*, which are parameterized by a state $s_\mathcal{I}$ and a plan π.

Plan outcome queries ($\mathcal{Q}_{PO}$) ask the agent the length of the longest prefix of the plan π that can be executed successfully when starting in the state $s_\mathcal{I} \in 2^\mathbb{P}$, and the resulting final state. E.g., "Given that the truck *t1* and package *p1* are at location *l1*, what would happen if you executed load_truck(p1,t1,l1), drive(t1,l1,l2), unload_truck(p1,t1,l2)?

A response to these queries can be of the form "I can execute the plan till step ℓ and at the end of it *p1* is in truck *t1* which is at location *l1*". Formally, the response $\mathcal{R}_{PO}$ for plan outcome queries is a tuple $\langle \ell, s_\mathcal{F} \rangle$, where ℓ is the number of steps for which the plan π was successfully able to run, and $s_\mathcal{F} \in 2^\mathbb{P}$ is the final state of the agent after executing ℓ steps of the plan. If the plan π is not executable according to the agent model $\mathcal{M}^\mathcal{A}$ then $\ell < len(\pi)$, otherwise if π is executable then $\ell = len(\pi)$. The final state $s_\mathcal{F} \in 2^\mathbb{P}$ such that $\mathcal{M}^\mathcal{A} \models \pi[0:\ell](s_\mathcal{I}) = s_\mathcal{F}$, i.e. starting with a state $s_\mathcal{I}$, $\mathcal{M}^\mathcal{A}$ successfully executed first ℓ steps of plan π. Thus, $\mathcal{Q}_{PO} : \mathcal{U} \to \mathbb{N} \times 2^\mathbb{P}$, where $\mathbb{N}$ is the set of natural numbers.

Not all queries are useful, as some of them might not increase our knowledge of the agent model at all. Hence we define some properties associated with each query to ascertain its usability. A query is useful only if it can distinguish between two models. More precisely, a query $\mathcal{Q}$ is said to *distinguish* a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$, denoted as $\mathcal{M}_i \rceil\!\lfloor^\mathcal{Q} \mathcal{M}_j$, iff $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$.

Given a pair of abstract models, we wish to determine whether one of them can be pruned – i.e., whether there's a query on which its answer is inconsistent with the agent's answer. Since this is computationally expensive to determine and we wish to reduce the number of queries made to the agent, we first evaluate whether the two models can be distinguished by any query, independent of consistency with the agent. If the models are not distinguishable, it doesn't make sense to try to prune one of them under the given query class. Formally,

Two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *distinguishable*, denoted as $\mathcal{M}_i \rceil\!\lfloor \mathcal{M}_j$, iff there exists a query that can distinguish between them, i.e. $\exists \mathcal{Q}\ \mathcal{M}_i \rceil\!\lfloor^\mathcal{Q} \mathcal{M}_j$.

In determining prunability, we need to consider the fact that the agent's response may be at a different level of abstraction if the given pair of models is abstract. When comparing the responses of two models at different levels of abstraction, we must also evaluate if the response of abstracted model $\mathcal{M}'$ is consistent with that of the agent, i.e. $\mathcal{Q}(\mathcal{M}^\mathcal{A}) \models \mathcal{Q}(\mathcal{M}')$. For plan outcome queries, consider that $\mathcal{Q}_{PO}(\mathcal{M}^\mathcal{A}) = \langle \ell, \langle p_1, \ldots, p_k \rangle \rangle$ and $\mathcal{Q}_{PO}(\mathcal{M}') = \langle \ell', \langle p'_1, \ldots, p'_j \rangle \rangle$. Now we can say that $\mathcal{Q}_{PO}(\mathcal{M}^\mathcal{A}) \models \mathcal{Q}_{PO}(\mathcal{M}')$ iff $(\ell = \ell')$, $j \leq k$ and $\forall i \in \{1, \ldots, j\} \bigwedge_i p_i = \bigwedge_i p'_i$.

Definition 3. Given an agent-interrogation task $\langle \mathcal{M}^\mathcal{A}, \mathbb{Q} \rangle$, two models M_i and M_j are *prunable* denoted as $\mathcal{M}_i \langle\rangle \mathcal{M}_j$, iff $\exists \mathcal{Q} \in \mathbb{Q} : M_i \rceil\!\lfloor^\mathcal{Q} M_j \wedge (\mathcal{Q}(\mathcal{M}^\mathcal{A}) \models \mathcal{Q}(M_i) \wedge$

(a) $\mathcal{M}^{\mathcal{A}}$'s `load_truck(?p,?t,?l)` action (unknown to $\mathcal{H}$)

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t),`
`at(?p,?l)`		$\neg$`(at(?p,?l))`

(b) $\mathcal{M}_1$'s `load_truck(?p,?t,?l)` action

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t)`
`at(?p,?l)`		

(c) $\mathcal{M}_2$'s `load_truck(?p,?t,?l)` action

`at(?t,?l)`	$\longrightarrow$	`in(?p,?t)`

(d) $\mathcal{M}_3$'s `load_truck(?p,?t,?l)` action

`at(?t,?l)`	$\longrightarrow$	`()`

Figure 3: *load_truck* actions of the agent model $\mathcal{M}^{\mathcal{A}}$ and three abstracted models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. Here $X \longrightarrow Y$ means that X is the precondition of an action and Y is the effect.

$\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(M_j)) \vee (Q(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(M_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(M_j))$.

2.1 Solving the Interrogation Task

Our approach iteratively generate pairs of abstract models and eliminates one of them by asking $\mathcal{A}$ queries and comparing its answer with that generated using the abstract models.

Example 1. Consider the case of a delivery robot. Assume that $\mathcal{H}$ is considering two abstract models $\mathcal{M}_1$ and $\mathcal{M}_2$ having only the predicates $at(?p, ?l)$, $at(?t, ?l)$, $in(?p, ?t)$ and the agent's model is $\mathcal{M}^{\mathcal{A}}$ (Fig. 3). $\mathcal{H}$ can ask the agent what will happen if $\mathcal{A}$ loads up package p1 into truck t1 at location l1 twice. The agent would respond that it could execute the plan only till length 1, and the state at the time of this failure was $at(t1, l1) \wedge in(p1, t1)$.

Algorithm 1 shows our overall algorithm for interrogating autonomous agents. It takes the agent $\mathcal{A}$, the set of instantiated predicates $\mathbb{P}^*$, and set of all action headers $\mathbb{A}_H$ as input and gives the set of functionally equivalent estimated models represented by $poss\_models$ as output. We initialize $poss\_models$ as empty set (line 3) representing that we are starting at the most abstract node in model lattice.

In each iteration of the main loop (line 4), we keep track of the current node in the lattice. We pick a pal tuple γ corresponding to one of the descending edges in the lattice from n given by some input ordering of Γ. The correctness of the algorithm does not depend on this ordering. We then generate all the new sets of models at the current node represented by the new_models (line 5). We also initialize an empty set at each lattice node to store the pruned models (line 6).

The inner loop (line 7) iterates over the set of all possible models $poss\_models$. Each abstract model represented by $\mathcal{M}^{abs}$ is then refined with the pal tuple γ giving three different models and form pairs from these models and iterate over these pairs (line 8 and 9). Here $\mathcal{M}^m_\gamma$ represents the abstract models equivalent to $\mathcal{M}^{abs} \cup \{\gamma^m\}$, where $m \in \{+, -, \emptyset\}$.

Algorithm 1 Agent Interrogation Algorithm

1: **Input:** $\mathcal{A}, \mathbb{A}_H, \mathbb{P}^*$
2: **Output:** poss_models
3: Initialize poss_models $= \{\emptyset\}$
4: **for** γ in some input pal ordering Γ **do**
5: new_models $\leftarrow$ poss_models $\times\{\gamma^+, \gamma^-, \gamma^\emptyset\}$
6: pruned_models$= \{\emptyset\}$
7: **for** each $\mathcal{M}^{abs}$ in poss_models **do**
8: $\{\mathcal{M}^+_\gamma, \mathcal{M}^-_\gamma, \mathcal{M}^\emptyset_\gamma\} = \{\mathcal{M}^{abs} \cup \gamma^+, \mathcal{M}^{abs} \cup \gamma^-, \mathcal{M}^{abs} \cup \gamma^\emptyset\}$
9: **for** each pair $\{\mathcal{M}_i, \mathcal{M}_j\}$ in $\{\mathcal{M}^+_\gamma, \mathcal{M}^-_\gamma, \mathcal{M}^\emptyset_\gamma\}$ **do**
10: $\mathcal{Q} \leftarrow$ generate_query$(\mathcal{M}_i, \mathcal{M}_j)$
11: $\mathcal{M}_{prune} \leftarrow$filter_models$(\mathcal{Q}, \mathcal{M}^{\mathcal{A}}, \mathcal{M}_i, \mathcal{M}_j)$
12: pruned_models$\leftarrow$ pruned_models$\cup\mathcal{M}_{prune}$
13: **end for**
14: **end for**
15: **if** pruned_models is $\emptyset$ **then**
16: update_pal_ ordering(Γ)
17: continue
18: **end if**
19: new_models $\leftarrow$ new_models $\setminus$ pruned_models
20: poss_models$\leftarrow$ poss_models $\cup$ new_models
21: **end for**

For each pair, we generate a query $\mathcal{Q}$ using *generate_query()* that can distinguish between the models in that pair (line 10). We then call *filter_models()* which poses the query $\mathcal{Q}$ to the agent and the two models. Based on their responses, we prune the models whose responses were not consistent with that of the agent (line 11). Then we update the estimated set of possible models represented by $poss\_models$ (line 19 and 20).

If we are unable to prune any models at a node (line 15), we update the order in which pal tuples are considered for refinement (line 16). We continue this process until we reach the most concretized node of the lattice (meaning all possible model components $\lambda \in \Lambda$ are refined). The remaining set of models represent the estimated set of models for the agent. This algorithm would require $O(|\mathbb{A}| \times |\mathbb{P}^*|)$ queries. However, our empirical studies show that we never generate so many queries. Next three sections describes the *generate_query()* (line 10) component, the *filter_models()* (line 11) component, and the *update_pal_ordering()* component (line 16) of the algorithm respectively.

2.2 Query Generation

The query generation process corresponds to *generate_query()* module in algorithm 1 which takes 2 models $\mathcal{M}_i$ and $\mathcal{M}_j$ as input and generates a query $\mathcal{Q}$ that can distinguish them, and if possible, satisfy prunability condition

(a) $\mathcal{M}_1$'s `load_truck(?p,?t,?l)` action
$\lambda^{\phi} = \langle$`at(?t,?l), load_truck(?p,?t,?l)`, *effect*,$\emptyset\rangle$

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t),`
`at(?p,?l)`		$\neg$`(at(?p,?l))`

(b) $\mathcal{M}_2$'s `load_truck(?p,?t,?l)` action
$\lambda^{+} = \langle$`at(?t,?l), load_truck(?p,?t,?l)`, *effect*,$+\rangle$

`at(?t,?l),`	$\longrightarrow$	`in(?p,?t),`
`at(?p,?l)`		$\neg$`(at(?p,?l)),`
		`at(?t,?l)`

Figure 4: Two abstracted model variants that are functionally equivalent

too.

Plan outcome queries can distinguish between models differing in either preconditions or effects of some action. We reduce the problem of creating plan outcome queries to a planning problem. The idea is to maintain a separate copy $\mathcal{P}^{\mathcal{M}_i}$ and $\mathcal{P}^{\mathcal{M}_j}$ of all the instantiated predicates $\mathcal{P}$, and formulate each precondition and effect of an action as a formula of predicates in both the copies of the predicates.

Let the planning problem $P_{PO} = \langle \mathcal{M}^{PO}, s_{\mathcal{I}}, s_{\mathcal{G}} \rangle$, where $\mathcal{M}^{PO}$ is a model with predicates $\mathcal{P}^{PO} = \mathcal{P}^{\mathcal{M}_i} \cup \mathcal{P}^{\mathcal{M}_j} \cup p_{\gamma}$, and actions $\mathbb{A}$ where for each action $a \in \mathbb{A}$, $pre(a) = pre(a^{\mathcal{M}_i}) \vee pre(a^{\mathcal{M}_j})$ and $eff(a) = (when(pre(a^{\mathcal{M}_i}) \wedge pre(a^{\mathcal{M}_j}))(eff(a^{\mathcal{M}_i}) \wedge eff(a^{\mathcal{M}_j}))\ (when((pre(a^{\mathcal{M}_i}) \wedge \neg pre(a^{\mathcal{M}_j})) \vee (\neg pre(a^{\mathcal{M}_i}) \wedge pre(a^{\mathcal{M}_j})))(p_{\gamma}))$, $s_{\mathcal{I}} = s_{\mathcal{I}}^{\mathcal{M}_i} \wedge s_{\mathcal{I}}^{\mathcal{M}_j}$ is the initial state where $s_{\mathcal{I}}^{\mathcal{M}_i}$ and $s_{\mathcal{I}}^{\mathcal{M}_j}$ are different copies of all predicates in the initial state, and $s_{\mathcal{G}}$ is the goal state and it is expressed as $\exists p\ (p^{\mathcal{M}_i} \wedge \neg p^{\mathcal{M}_j}) \vee (\neg p^{\mathcal{M}_i} \wedge p^{\mathcal{M}_j}) \vee p_{\gamma}$.

With this formulation whenever we have at least one action in both the models which has different effects in both of them, the goal will be reached. For example, consider the models $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{M}_1$ mentioned in Fig. 3. On applying the *load_truck*$(p1, t1, l1)$ action from the state where the action can be applied in both the models, one of them will lead to $at(p1, l1)$ being false and the other will not. Hence starting with an initial state $s_{\mathcal{I}} = at(t1, l1) \wedge at(p1, l1)$, the plan to reach the goal will be *load_truck*$(p1, t1, l1)$.

Also, whenever we have an action a which cannot be applied in the same state s_d in both the models, the planner will generate a plan to take the agent from the initial state to state s_d, and append action a to that plan. This new plan will be the solution to the planning problem P_{PO}. For example, consider the models $\mathcal{M}_1$ and $\mathcal{M}_2$ mentioned in Fig. 3. In a state where $at(t1, l1)$ is true and $at(p1, l1)$ is false, we can apply *load_truck*$(p1, t1, l1)$ in $\mathcal{M}_2$ but not in $\mathcal{M}_1$. Hence for an initial state $s_{\mathcal{I}} = at(t1, l1) \wedge \neg at(p1, l1)$, the plan to reach the goal will be *load_truck*$(p1, t1, l1)$.

If no solution exists for the planning problem P_{PO}, then it implies that the two models are functionally equivalent, and no combination of initial state and a plan can distinguish between them. An intuitive example of this will be a pair of models where in one of the model the same literal appears in both the preconditions and effects, whereas in the other model the same literal appears only in the precondition. For example, consider the models $\mathcal{M}_1$ and $\mathcal{M}_2$ shown in Fig. 4, any plan including *load_truck(?p,?t,?l)* action will have $at(?t, ?l)$ in precondition. According to both the models, it will also be present after executing the action, hence these models cannot be distinguished based on this query generation process, even though their STRIPS encoding is not *syntactically same*. Previous works like LOUGA (Kučera & Barták, 2018) handle this by making assumption that a predicate is deleted from a state by an action only if it is already present in the state, and a predicate is added only if it is not present in the state already. This assumption prevents models like $\mathcal{M}_2$ from being considered as valid models. We also apply these assumptions to reduce the computation, and this process is called *normalization*. The following theorem formalizes these notions.

Theorem 1. Given a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$, the planning problem P_{PO} has a solution iff $\mathcal{M}_i$ and $\mathcal{M}_j$ have a distinguishing plan outcome query $\mathcal{Q}_{PO}$.

Proof (Sketch). $\mathcal{Q}_{PO}$ comprises of an initial state $s_{\mathcal{I}}$ and plan π. The initial state $s_{\mathcal{I}}$ in $\mathcal{Q}_{PO}$ and P_{PO} is same. Starting with this initial state, an action becomes a part of the plan π only when it can be applied in any one or both of the models $\mathcal{M}_i$ and $\mathcal{M}_j$. So two cases arise here, if the action can be executed in both the models, the effect of both the actions is applied to the state and next action is searched. Otherwise if the action is applicable only in one of the models, but not the other, the effect of the action is a dummy proposition p_{γ} which is also the goal. So as soon an action is found that is possible in one of the models but not the other, or if it gives different resulting states in both the models, the resulting plan becomes the plan needed by query $\mathcal{Q}_{PO}$. Hence if the planning problem P_{PO} gives a solution plan π, then there exists a query $\mathcal{Q}_{PO}$ that consists of $s_{\mathcal{I}}$ and π as input.

Also, as described previously, whenever there exists a distinguishing plan-outcome query, the starting state $s_{\mathcal{I}}$ is part of $\mathcal{Q}_{PO}$, and the way we generate the P_{PO} problem ensures we will get a plan π as the solution. □

2.3 Filtering Possible Models

This section describes the *filter_models()* module in algorithm 1 which takes as input the agent model $\mathcal{M}^{\mathcal{A}}$, the two abstract models being compared $\mathcal{M}_i$ and $\mathcal{M}_j$, and the query $\mathcal{Q}$ (generated by the *generate_query()* module explained in section 2.2), and returns the subset $\mathcal{M}_{prune}$ which is not consistent with $\mathcal{M}^{\mathcal{A}}$.

Firstly, the algorithm *asks the query* $\mathcal{Q}$ to both the models $\mathcal{M}_i$ and $\mathcal{M}_j$ and the agent $\mathcal{M}^{\mathcal{A}}$. Based on the responses of all three, it determines if the two models are prunable, i.e. $\mathcal{M}_i \langle\rangle \mathcal{M}_j$. As mentioned in Def. 3, checking for prunability involves checking if responses to the query $\mathcal{Q}$ by one of the models $\mathcal{M}_i$ or $\mathcal{M}_j$ is consistent with that of the agent or not.

If the models are prunable, let the model not consistent with the agent be $\mathcal{M}'$ where $\mathcal{M}' \in \{\mathcal{M}_i, \mathcal{M}_j\}$. Now recall that a model is a set of palm tuples. As shown in Fig. 2, based on response to a query, if a model is found to be inconsistent for the first time at a node n in the lattice, with an incoming edge of label γ, any model with same mode of γ as $\mathcal{M}'$ will also be inconsistent. This is because a palm tuple uniquely identifies the mode in which a predicate will appear in an action's location which can be precondition or effect. And since this tuple is inconsistent with the agent, any model containing this will also contain the same mode of predicate in that action's precondition or effect. This idea paves way for the concept of partitions which is discussed below.

Given lattice nodes n_i and n_j, the edge $n_j \rightarrow n_i$ labeled γ, and the set Λ of palm tuples present at the parent node n_j, a *partition* of node n_i is the set of disjoint subsets $\Lambda \cup \{\gamma^+\}$, $\Lambda \cup \{\gamma^-\}$, and $\Lambda \cup \{\gamma^\emptyset\}$. So depending on the model $\mathcal{M}'$ which is inconsistent with agent model $\mathcal{M}^{\mathcal{A}}$, we can prune out the whole partition containing $\mathcal{M}'$. This partition is returned by *filter_models()* module as $\mathcal{M}_{prune}$.

2.4 Updating pal ordering Γ

Models may not be prunable if the query is not executable by $\mathcal{A}$ and none of the model's query responses are consistent with that of the agent. For eg, consider two abstract models $\mathcal{M}_2$ and $\mathcal{M}_3$ being considered by the human interrogator $\mathcal{H}$ (Fig. 3). At this level of abstraction, $\mathcal{H}$ does not have knowledge of predicate $at(?p, ?l)$, hence it will generate a plan outcome query with initial state $at(?t, ?l)$ and plan *load_truck*$(p1, t1, l1)$ to distinguish between $\mathcal{M}_2$ and $\mathcal{M}_3$. But this cannot be executed by agent $\mathcal{A}$ as the precondition $at(?p, ?l)$ is not satisfied. In such cases, we cannot discard any partitions. Hence if no prunable query is possible, i.e. the palm tuple set Λ being considered is last in $poss\_models$, we update the pal ordering. Recall that in response to the plan outcome query we get the failed action $a_{Fail} = \pi[\ell+1]$ and the final state $s_{\mathcal{F}}$. Let us assume that the query was executable on $\mathcal{M}_i$, but not on $\mathcal{M}_j$. Now assuming $\mathcal{M}_i$ is an abstracted version of $\mathcal{M}^{\mathcal{A}}$, let the state it reaches after executing first ℓ steps of the plan be $\overline{s}_{\mathcal{F}}$. Now we can infer that one of the literal present in $s_{\mathcal{F}} \setminus \overline{s}_{\mathcal{F}}$ (represented as $\{l_1, \ldots, l_k\}$) is causing the action a_{Fail} to fail. We now generate a new query with $s_{\mathcal{I}} = \overline{s}_{\mathcal{F}}$ and keeping a subset of $\{\neg l_1 \ldots \neg l_k\}$. With this $s_{\mathcal{I}}$ as initial state, the agent $\mathcal{A}$ should be able to execute the plan $\pi = a_{Fail}$. In next step we change the initial state $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge l_k$ and remove l_k from the subset we found earlier. If $\mathcal{A}$ still executes $\pi = a_{Fail}$, then l_k was not the literal responsible for the failure of a_{Fail} and we change $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge l_k \wedge l_{k-1}$, otherwise we can infer that l_k was indeed one of the literals responsible for failure of a_{Fail}, and we change $s_{\mathcal{I}}$ to $\overline{s}_{\mathcal{F}} \wedge \neg l_k \wedge l_{k-1}$. We do this k times to determine the literals responsible for failure of action a_{Fail}. For each of such literals causing the failure, we get their correct palm tuples. For eg. if we inferred that $at(?p, ?l)$ was not present in the state and hence was causing the action *load_truck*$(?p, ?t, ?l)$ to fail, we get the correct palm tuple as $\langle at(?p, ?l), \textit{load\_truck}(?p, ?t, ?l), precondition, + \rangle$. We need not refine in terms of the corresponding pal tuple $\langle at(?p, ?l), \textit{load\_truck}(?p, ?t, ?l), precondition \rangle$ in future, so we update the pal ordering Γ by removing it from Γ.

2.5 Correctness of Agent Interrogation Algorithm

In this section we prove that the set of estimated models returned by the agent interrogation algorithm are correct and are functionally equivalent to the agent's model, and no correct model was ever discarded in the process.

Theorem 2. The Agent Interrogation Algorithm (algorithm 1) will always terminate and return a set of models, each of which are functionally equivalent to agent's model $\mathcal{M}^{\mathcal{A}}$.

Proof (Sketch). Because of the formulation of the planning problem P_{PO}, for a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$, P_{PO} has a solution iff $\mathcal{M}_i$ and $\mathcal{M}_j$ have a distinguishing plan outcome query $\mathcal{Q}_{PO}$. Direct result of this will be that for the refinement in terms of *pal* tuple $\gamma = \langle p, a, l{=}precondition \rangle$ of two models $\mathcal{M}_i$ and $\mathcal{M}_j$, if $\mathcal{M}_i$ and $\mathcal{M}_j$ are not distinguishable $\mathcal{M}_i \not\langle\rangle \mathcal{M}_j$, then their refinements when adding γ will be distinguishable only if the refinements belong to different partitions. Same holds for the effects case too, but now depending on presence of a positive or negative literal in the state, we might not be able to distinguish when one of the mode is absent. Also, if we prune away an abstract model $\mathcal{M}^{abs}$, then no possible concretization of $\mathcal{M}^{abs}$ will result into a model consistent with the agent model $\mathcal{M}^{\mathcal{A}}$. This is because whenever we get a prunable query, we are discarding only the inconsistent models, thereby ensuring that no correct model was ever discarded. For the cases when we don't get a prunable query, we infer the correct precondition of the failed action, hence the number of refined palm tuples always increase as the number of iterations of the algorithm increase, thereby ensuring the termination of the algorithm in finite time. And finally, the models leftover at the most concretized node (after all the palm tuples are refined) are going to be the set of models which the algorithm could not discard, hence all of these models are guaranteed to be functionally equivalent to the agent model. □

Domain	$\lvert\mathbb{P}^*\rvert$	$\lvert\mathbb{A}\rvert$	$\lvert\overline{\mathcal{Q}}\rvert$	$\lvert M\rvert$	Algorithm 1	
					$\lvert\hat{\mathcal{Q}}\rvert$	Time/$\mathcal{Q}$ (sec)
gripper	5	3	$15 * 2^5$	1	37	0.14
blocks*	9	4	$36 * 2^9$	1	92	1.73
elevator	10	4	$40 * 2^{10}$	64	109	5.91
logistics	11	6	$66 * 2^{11}$	64	98	11.62
parking	18	4	$72 * 2^{18}$	32	173	12.01
satellite	17	5	$85 * 2^{17}$	4096	127	19.53

Table 1: Comparison of the number of queries and average time per query in our approach vs a naive baseline. $\lvert\hat{\mathcal{Q}}\rvert$ denotes the number of queries used in our approach, as opposed to the number of queries $\lvert\overline{\mathcal{Q}}\rvert$ that would be required in a naive solution (see Sec. 3). $\lvert M\rvert$ denotes the number of equivalent models that Algorithm 1 would've returned without normalization. Times shown are averages of time taken per query across 10 runs of the agent interrogation algorithm.

Full name of IPC domain is *blocks-world.

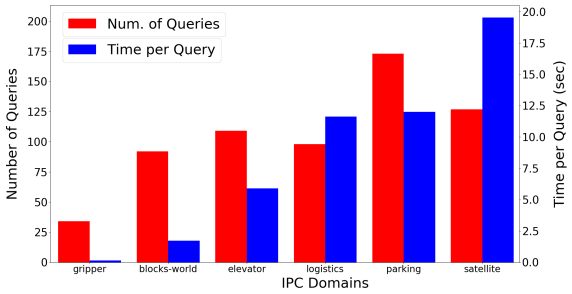

Figure 5: Bar chart showing the number of queries and time per query for the seven IPC domains.

3 Empirical Evaluation

The approach developed in this paper uses very sparse but selective information from the agent to infer its mode. Existing approaches for model learning (see Sec. 1) cannot work with such information (outcomes of plans but not their intermediate states). However, we can consider a naive querying method for solving the problem as a baseline: all possible states using the instantiated predicates can be generated to be used as initial states, and for each initial state a query can be formed with a plan containing single action (one such plan for each action). Based on the agent's responses, each action's preconditions and actions can be inferred. This method is guaranteed to find the solution but the complexity of this approach is exponential in the number of predicates.

To test our approach, in each run we created an agent (unknown to Alg. 1) with one of the 7 IPC domain models. We then evaluated the performance of Alg. 1 in estimating that agent's model using 10 different problems from that domain. We generate initial states for queries using an increasing number of objects until a distinguishing query is found. In all our experiments, such a query was found using at most 7-8 objects. This number is found to correlate with the maximum arity of the predicates in the domain. Hence the number of queries is not affected by the number of objects as the approach finds the minimum number of objects needed to distinguish between the abstracted models and uses it, and providing more objects does not change the behaviour of the algorithm. This was further validated from the test runs as the value of $\lvert\hat{\mathcal{Q}}\rvert$ for a domain did not change across the 10 problems they were tested on.

Table 1 and Fig. 5 illustrate our results. Table 1 shows that the number of queries asked ($\lvert\hat{\mathcal{Q}}\rvert$) in our approach is much smaller than that needed for the naive method ($\lvert\overline{\mathcal{Q}}\rvert$). The number of equivalent models that the algorithm would've returned is unusually high for satellite domain as the case mentioned in Fig. 4, i.e. when a predicate appeared as a positive literal in the precondition of an action, but is either missing or positive in effect of that action, was present 12 times. Even for three other domains, this case caused the large number of equivalent models. Such large number of equivalent models affect the running time of the algorithm adversely.

All the experiments were run on 4.9 GHz Intel Xeon E5-2680 v4 CPUs with 64 GB RAM running Ubuntu 18.04. We used FF planner (Hoffmann & Nebel, 2001) to solve the planning problems.

Also, there is no definite pattern in the number of queries asked as the order in which queries were asked (depending of ordering of γ tuples) was random. So a better query asked earlier in the interrogation process can lead to a smaller number of queries asked.

4 Conclusion

We have presented a novel approach for estimating the model of an autonomous agent by interrogating it. In this paper we showed that the number of queries required to estimate the model is dependent only on the number of actions and predicates, and is independent of the size of the environment. An advantage of learning models of the form described in this paper is that they permit assessments of causality (Halpern, 2016).

Extending this approach to more general types of agents and environments featuring partial observability and/or non-determinism is a promising direction for future work.

Although our interface is a set of plans represented as logical statements, other works have explored using natural language as a way to provide plans as input (Lindsay et al., 2017). In future, this can be used to extend our work thereby making the communication more realistic and close to how a human interrogator might actually interact with an autonomous agent.

Acknowledgements

We thank Shashank Rao Marpally and Abhyudaya Srinet for their help with the implementation. This work was supported in part by the NSF under grants IIS 1844325 and IIS 1909370.

References

Aineto, D., Jiménez, S., Onaindia, E., and Ramírez, M. Model recognition as planning. In *Proc. ICAPS*, 2019.

Amir, E. and Chang, A. Learning partially observable deterministic action models. *Journal of Artificial Intelligence Research*, 33:349–402, 2008.

Amir, E. and Russell, S. Logical filtering. In *Proc. IJCAI*, 2003.

Bäckström, C. and Jonsson, P. Bridging the gap between refinement and heuristics in abstraction. In *Proc. IJCAI*, 2013.

Bryce, D., Benton, J., and Boldt, M. W. Maintaining evolving domain models. In *Proc. IJCAI*, 2016.

Camacho, A. and McIlraith, S. A. Learning interpretable models expressed in linear temporal logic. In *Proc. ICAPS*, 2019.

Cresswell, S. and Gregory, P. Generalised domain model acquisition from action traces. In *Proc. ICAPS*, 2011.

Cresswell, S., McCluskey, T., and West, M. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proc. ICAPS*, 2009.

Fikes, R. E. and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

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

Giunchiglia, F. and Walsh, T. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

Halpern, J. Y. *Actual Causality*. The MIT Press, 2016. ISBN 0262035022.

Helmert, M., Haslum, P., Hoffmann, J., et al. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*, 2017.

Hoffmann, J. and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Khardon, R. and Roth, D. Reasoning with models. *Artificial Intelligence*, 87(1-2):187–213, November 1996.

Kučera, J. and Barták, R. LOUGA: Learning planning operators using genetic algorithms. In Yoshida, K. and Lee, M. (eds.), *Knowledge Management and Acquisition for Intelligent Systems*, pp. 124–138, 2018.

Lindsay, A., Read, J., Ferreira, J., Hayton, T., Porteous, J., and Gregory, P. Framer: Planning models from natural language action descriptions. In *Proc. ICAPS*, 2017.

Ng, J. H. A. and Petrick, R. Incremental learning of planning actions in model-based reinforcement learning. In *Proc. IJCAI*, 2019.

Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

Settles, B. Active learning literature survey. Technical Report 1648, University of Wisconsin-Madison Department of Computer Sciences, 2009.

Srivastava, S., Russell, S., and Pinto, A. Metaphysics of planning domain descriptions. In *Proc. AAAI*, 2016.

Stern, R. and Juba, B. Efficient, safe, and probably approximately complete learning of action models. In *Proc. IJCAI*, 2017.

Yang, Q., Wu, K., and Jiang, Y. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, February 2007.