Asking the Right Questions: Learning Interpretable Action Models Through Query Answering

Pulkit Verma, Shashank Rao Marpally, and **Siddharth Srivastava**
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ 85281, USA
{verma.pulkit, smarpall, siddharths}@asu.edu

Abstract

This paper develops a new approach for estimating an interpretable, relational model of a black-box autonomous agent that can plan and act. Our main contributions are a new paradigm for estimating such models using a rudimentary query interface with the agent and a hierarchical querying algorithm that generates an interrogation policy for estimating the agent's internal model in a user-interpretable vocabulary. Empirical evaluation of our approach shows that despite the intractable search space of possible agent models, our approach allows correct and scalable estimation of interpretable agent models for a wide class of black-box autonomous agents. Our results also show that this approach can use predicate classifiers to learn interpretable models of planning agents that represent states as images.

1 Introduction

The growing deployment of AI systems ranging from personal digital assistants to self-driving cars leads to a pervasive problem: how would a user ascertain whether an AI system 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 it may even change from initial specifications as a result of learning. Such scenarios feature *black-box* AI agents whose models may not be available in terminology that the user understands. They also show that in addition to developing better AI systems, we need to develop new algorithmic paradigms for assessing arbitrary AI systems and for determining the minimal requirements for AI systems in order to ensure interpretability and to support such assessments (Srivastava 2021).

This paper presents a new approach for addressing these questions. It develops an algorithm for estimating interpretable, relational models of AI agents by querying them. In doing so, it requires the AI system to have only a primitive query-response capability to ensure interpretability. Consider a situation where Hari(ette) ($\mathcal{H}$) wants a grocery-delivery robot ($\mathcal{A}$) to bring some groceries, 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 and Nilsson 1971; McDermott et al. 1998; Fox and Long 2003)). If $\mathcal{H}$ was dealing with a delivery person, s/he might ask them questions such as "would you pick up orders from multiple persons?" and "do you think it would be alright to bring refrigerated items in a regular bag?" If the answers are "yes" during summer, it would be a cause for concern. Naïve approaches for generating such questions to ascertain the limits and capabilities of an agent are infeasible.[1]

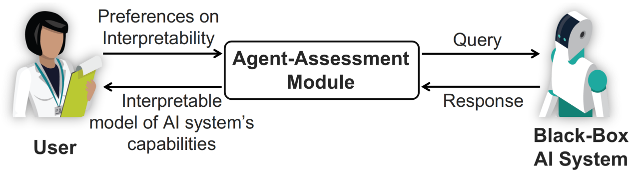

Figure 1: The agent-assessment module uses its user's preferred vocabulary, queries the AI system, and delivers a user-interpretable causal model of the AI system's capabilities. The AI system does not need to know the user's vocabulary or modeling language.

We propose an agent-assessment module (AAM), shown in Fig. 1, which can be connected with an arbitrary AI agent that has a rudimentary query-response capability: the assessment module connects $\mathcal{A}$ with a simulator and provides a sequence of instructions, or a plan as a *query*. $\mathcal{A}$ executes the plan in the simulator and the assessment module uses the simulated outcome as the response to the query. Thus, given an agent, the assessment module uses as input: a user-defined vocabulary, the agent's instruction set, and a compatible simulator. These inputs reflect natural requirements of the task and are already quite commonly supported: AI systems are already designed and tested using compatible simulators, and they need to specify their instruction sets in order to be usable. The user provides the concepts that they

[1] Just 2 actions and 5 grounded propositions would yield $7^{2\times5} \sim 10^8$ possible STRIPS-like models – each proposition could be absent, positive or negative in the precondition and effects of each action, and cannot be positive (or negative) in both preconditions and effect simultaneously. A query strategy that inquires about each occurrence of each proposition would be not only unscalable but also inapplicable to simulator-based agents that do not know their actions' preconditions and effects.

can understand and these concepts can be defined as functions on simulator states.

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. Such models can be easily translated into interpretable descriptions such as "under situations where *preconditions* hold, if the agent $\mathcal{A}$ executes actions $a_1, \ldots, a_k$ it would result in *effects*," where preconditions and effects use only the user-provided concepts. Furthermore, such models can be used to investigate counterfactuals and support assessments of causality (Halpern 2016).

This fundamental framework (Sec. 3) can be developed to support different types of agents as well as various query and response modalities. E.g., queries and responses could use a speech interface for greater accessibility, and agents with reliable inbuilt simulators/lookahead models may not need external simulators. This would allow AAM to pose queries such as "what do you think would happen if you did $\langle$*query plan*$\rangle$", and the learnt model would reflect $\mathcal{A}$'s self-assessment. The "agent" could be an arbitrary entity, although the expressiveness of the user-interpretable vocabulary would govern the scope of the learnt models and their accuracy. Using AAM with such agents would also help make them compliant with Level II assistive AI – systems that make it easy for operators to learn how to use them safely (Srivastava 2021).

Our algorithm for the assessment module (Sec. 3.1) generates a sequence of queries ($\mathcal{Q}$) depending on the agent's responses (θ) during the query process; the result of the overall process is a complete model of $\mathcal{A}$. To generate queries, we use 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's answer, we effectively eliminate the entire set of possible concrete models that are refinements of this abstract model. Sec. 3 presents our overall framework with algorithms and theoretical results about their convergence properties.

Our empirical evaluation (Sec. 4) shows that this method can efficiently learn correct models for black-box versions of agents using hidden models from the IPC [2]. It also shows that AAM can use image-based predicate classifiers to infer correct models for simulator-based agents that respond with an image representing the result of query plan's execution.

2 Related Work

A number of researchers have explored the problem of learning agent models from observations of its behavior (Gil 1994; Yang, Wu, and Jiang 2007; Cresswell, McCluskey, and West 2009; Zhuo and Kambhampati 2013). Such action-model learning approaches have also found practical applications in robot navigation (Balac, Gaines, and Fisher 2000), player behavior modeling (Krishnan, Williams, and Martens 2020), etc. To the best of our knowledge, ours is the first approach to address the problem of generating query strategies for inferring relational models of black-box agents.

Amir and Chang (2008) use logical filtering (Amir and Russell 2003) to learn partially observable action models from the observation traces. LOCM (Cresswell, McCluskey, and West 2009) and LOCM2 (Cresswell and Gregory 2011) present another class of algorithms that use finite-state machines to create action models from observed plan traces. Camacho and McIlraith (2019) present an approach for learning highly expressive LTL models from an agent's observed state trajectories using an oracle with knowledge of the target LTL representation. This oracle can also generate counterexamples when the estimated model differs from the true model. In contrast, our approach does not require such an oracle. Also, unlike Stern and Juba (2017), our approach does not need intermediate states in execution traces. In contrast to approaches for white-box model maintenance (Bryce, Benton, and Boldt 2016), our approach does not require $\mathcal{A}$ to know about $\mathcal{H}$'s preferred vocabulary.

LOUGA (Kučera and Barták 2018) combines a genetic algorithm with an ad-hoc method to learn planning operators from observed plan traces. FAMA (Aineto, Celorrio, and Onaindia 2019) reduces model recognition to a planning problem and can work with partial action sequences and/or state traces as long as correct initial and goal states are provided. While both FAMA and LOUGA require a postprocessing step to update the learnt model's preconditions to include the intersection of all states where an action is applied, it is not clear that such a process would necessarily converge to the correct model. Our experiments indicate that such approaches exhibit oscillating behavior in terms of model accuracy because some data traces can include spurious predicates, which leads to spurious preconditions being added to the model's actions. FAMA also assumes that there are no negative literals in action preconditions.

Bonet and Geffner (2020) present an algorithm for learning relational models using a SAT-based method when the action schema, predicates, etc. are not available. This approach takes as input a predesigned correct and complete directed graph encoding the structure of the entire state space. The authors note that their approach is viable for problems with small state spaces. While our method provides an end-to-end solution, it can also be used in conjunction with such approaches to create the inputs they need. Khardon and 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.

In contrast to these directions of research, our approach directly queries the agent and is guaranteed to converge to the true model while presenting a running estimate of the accuracy of the derived model; hence, it can be used in settings where the agent's model changes due to learning or a software update. In such a scenario, our algorithm can restart to query the system, while approaches that derive models from observed plan traces would require arbitrarily long data collection sessions to get sufficient uncorrelated data.

Incremental Learning Model (Ng and Petrick 2019) uses reinforcement learning to learn a nonstationary model without using plan traces, and requires extensive training to learn the full model correctly. Chitnis et al. (2021) present an

[2] https://www.icaps-conference.org/competitions

approach for learning probabilistic relational models where they use goal sampling as a heuristic for generating relevant data, while we reduce that problem to query synthesis using planning. Their approach is shown to work well for stochastic environments, but puts a much higher burden on the AI system for inferring its model. This is because the AI system has to generate a conjunctive goal formula while maximizing exploration, find a plan to reach that goal, and correct the model as it collects observations while executing the plan.

The field of active learning (Settles 2012) addresses the related problem of selecting which data-labels to acquire for learning single-step decision-making models using statistical measures of information. However, the effective feature set here is the set of all possible plans, which makes conventional methods for evaluating the information gain of possible feature labelings infeasible. In contrast, our approach uses a hierarchical abstraction to select queries to ask, while inferring a multistep decision-making (planning) model. Information-theoretic metrics could also be used in our approach whenever such information is available.

3 The Agent-Interrogation Task

We assume that $\mathcal{H}$ needs to estimate $\mathcal{A}$'s model as a STRIPS-like planning model represented as a pair $\mathcal{M} = \langle \mathbb{P}, \mathbb{A} \rangle$, where $\mathbb{P} = \{p_1^{k_1}, \dots, p_n^{k_n}\}$ is a finite set of predicates with arities k_i; $\mathbb{A} = \{a_1, \dots, a_k\}$ is a finite set of parameterized actions (operators). 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 consisting of action name and action parameters, $pre(a_j)$ represents the set of predicate atoms that must be true in a state where a_j can be applied, $\mathit{eff}(a_j)$ is the set of positive or negative predicate atoms that will change to true or false respectively as a result of execution of the action a_j. Each predicate can be instantiated using the parameters of an action, where the number of parameters are bounded by the maximum arity of the action. E.g., consider the action *load_truck*$(?v1, ?v2, ?v3)$ and predicate $at(?x, ?y)$ in the IPC Logistics domain. This predicate can be instantiated using action parameters $?v1$, $?v2$, and $?v3$ as $at(?v1, ?v1)$, $at(?v1, ?v2)$, $at(?v1, ?v3)$, $at(?v2, ?v2)$, $at(?v2, ?v1)$, $at(?v2, ?v3)$, $at(?v3, ?v3)$, $at(?v3, ?v1)$, and $at(?v3, ?v2)$. We represent the set of all such possible predicates instantiated with action parameters as $\mathbb{P}^*$.

AAM uses the following information as input. It receives its instruction set in the form of $header(a)$ for each $a \in \mathbb{A}$ from the agent. AAM also receives a predicate vocabulary $\mathbb{P}$ from the user with functional definitions of each predicate. This gives AAM sufficient information to perform a dialog with $\mathcal{A}$ 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 which can answer these queries, determine the model of the agent. More precisely, an *agent interrogation task* is defined as a tuple $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q}, \mathbb{P}, \mathbb{A}_H \rangle$, where $\mathcal{M}^{\mathcal{A}}$ is the true model (unknown to AAM) of the agent $\mathcal{A}$ being interrogated, $\mathbb{Q}$ is the class of queries that can be posed to the agent by AAM, and $\mathbb{P}$ and $\mathbb{A}_H$ are the sets of predicates and action headers that AAM uses based on inputs from $\mathcal{H}$ and $\mathcal{A}$. The objective of the agent interrogation task is to derive the agent model $\mathcal{M}^{\mathcal{A}}$ using $\mathbb{P}$ and $\mathbb{A}_H$. Let Θ be the set of possible answers to queries. Thus, strings $\theta^* \in \Theta^*$ denote the information received by AAM at any point in the query process. Query policies for the agent interrogation task are functions $\theta^* \rightarrow \mathbb{Q} \cup \{\mathit{Stop}\}$ that map sequences of answers to the next query that the interrogator should ask. The process stops with the *Stop* query. In other words, for all answers $\theta \in \Theta$, 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 vocabulary $\mathbb{P}^*$; a is an action from the set of parameterized actions $\mathbb{A}$, $l \in \{\mathit{pre}, \mathit{eff}\}$, and $m \in \{+, -, \emptyset\}$. For convenience, we use the 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 mode 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 assignment to a *pal* tuple $\gamma = \langle p, a, l \rangle$ can result in 3 palm tuple 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 now define the notion of abstraction used in our solution approach. Several approaches have explored the use of abstraction in planning (Sacerdoti 1974; Giunchiglia and Walsh 1992; Helmert et al. 2007; Bäckström and Jonsson 2013; Srivastava, Russell, and Pinto 2016). The definition of abstraction used in this work extends the concept of predicate and propositional domain abstractions (Srivastava, Russell, and Pinto 2016) to allow for the projection of a single *palm* tuple λ.

An abstract model is one in which all variants of at least one pal tuple are absent. Let Λ be the set of all possible palm tuples which can be generated using a predicate vocabulary $\mathbb{P}^*$ and an action header set $\mathbb{A}_H$. Let $\mathcal{U}$ be the set of all consistent (abstract and concrete) models that can be expressed as subsets of Λ, such that no model has multiple variants of the same palm tuple. We define abstraction of a model as:

Definition 1. The *abstraction of a model* $\mathcal{M}$ with respect to a palm tuple $\lambda \in \Lambda$, is defined by $f_\lambda : \mathcal{U} \rightarrow \mathcal{U}$ as $f_\lambda(\mathcal{M}) = \mathcal{M} \setminus \lambda$.

We extend this notation to define the abstraction of a set of models $\mathcal{M}$ with respect to a palm tuple λ as $X = \{f_\lambda(m) : m \in \mathcal{M}\}$. We use this abstraction framework to define a subset-lattice over abstract models (Fig. 2(b)). Each node in the lattice represents a collection of possible abstract models which are possible variants of a pal tuple γ. E.g., in the node labeled 1 in Fig. 2(b), we have models corresponding

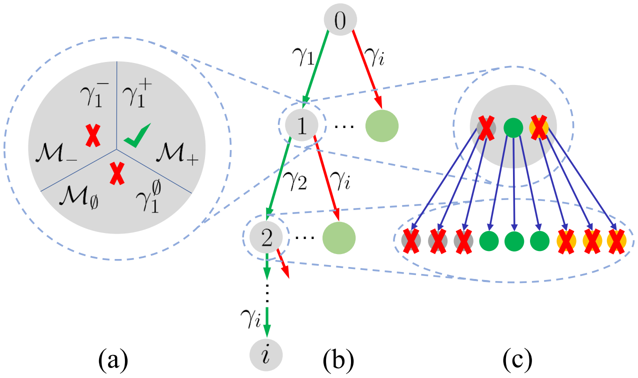

Figure 2: (b) Lattice segment explored in random order of $\gamma_i \in \Gamma$; (a) At each node, 3 abstract models are generated and 2 of them are discarded based on query responses; (c) An abstract model rejected at any level is equivalent to rejecting 3 models at the level below, 9 models two levels down, and so on.

to γ_1^+, γ_1^-, and $\gamma_1^\emptyset$. Two nodes in the lattice are at the same level of abstraction if they contain the same number of pal tuples. Two nodes n_i and n_j in the lattice are connected if all the models at n_i differ with all the models in n_j by a single palm tuple. As we move up in the lattice following these edges, we get more abstracted versions of the models, i.e., containing less number of pal tuples; and we get more concretized models, i.e., containing more number of pal tuples, as we move downward. We now define this model lattice:

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 $\langle p, a, l \rangle$, $\ell_N : N \rightarrow 2^{2^\Lambda}$ is a node label function where $\Lambda = \Gamma \times \{+, -, \emptyset\}$ is the set of all palm tuples, E is the set of lattice edges, and $\ell_E : E \rightarrow \Gamma$ is a function mapping edges to edge labels such that for each edge $n_i \rightarrow n_j$, $\ell_N(n_j) = \{\xi \cup \{\gamma^k\} | \xi \in \ell_N(n_i), \gamma = \ell_E(n_i \rightarrow n_j), k \in \{+, -, \emptyset\}\}$, and $\ell_N(\top) = \{\phi\}$ where $\top$ is the supremum containing the empty model ϕ.

A node $n \in N$ in this lattice $\mathcal{L}$ can be uniquely identified by the sequence of pal tuples that label the edges leading to it from the supremum. As shown in Fig. 2(a), even though theoretically $\ell_N : N \rightarrow 2^{2^\Lambda}$, not all the models are stored at any node as at least one is pruned out based on some query $\mathcal{Q} \in \mathbb{Q}$. Additionally, 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 possible models given the agent's responses up to that point. At every step, our algorithm creates queries online that help us determine the next descending edge to take from a lattice node; corresponding to the path $0, \ldots, i$ in Fig. 2(b). This also avoids generating and storing the complete lattice, which can be doubly exponential in number of predicates and actions.

Form of agent queries As discussed earlier, based on $\mathcal{A}$'s responses θ, we pose queries to the agent and infer $\mathcal{A}$'s model. We express queries as functions that map models to answers. Recall that $\mathcal{U}$ is the set of all possible (concrete and abstract) models, and Θ is the set of possible responses. A *query* $\mathcal{Q}$ is a function $\mathcal{Q} : \mathcal{U} \rightarrow \Theta$.

In this paper, we utilize only one class of queries: *plan outcome queries* ($\mathcal{Q}_{PO}$), which are parameterized by a state s_I and a plan π. Let P be the set of predicates $\mathbb{P}^*$ instantiated with objects O in an environment. $\mathcal{Q}_{PO}$ queries ask $\mathcal{A}$ the length of the longest prefix of the plan π that it can execute successfully when starting in the state $s_\mathcal{I} \subseteq P$ as well as the final state $s_\mathcal{F} \subseteq P$ that this execution leads to. E.g., "Given that the truck $t1$ and package $p1$ are at location $l1$, what would happen if you executed the plan $\langle load\_truck(p1, t1, l1)$, $drive(t1, l1, l2)$, $unload\_truck(p1, t1, l2)\rangle$?"

A response to such 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 θ_{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 could be executed, and $s_\mathcal{F} \subseteq P$ is the final state after executing ℓ steps of the plan. If the plan π cannot be executed fully according to the agent model $\mathcal{M}^\mathcal{A}$ then $\ell < len(\pi)$, otherwise $\ell = len(\pi)$. The final state $s_\mathcal{F} \subseteq P$ is such that $\mathcal{M}^\mathcal{A} \models \pi[1 : \ell](s_\mathcal{I}) = s_\mathcal{F}$, i.e., starting with a state $s_\mathcal{I}$, $\mathcal{M}^\mathcal{A}$ successfully executed first ℓ steps of the plan π. Thus, $\mathcal{Q}_{PO} : \mathcal{U} \rightarrow \mathbb{N} \times 2^P$, 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 $\mathcal{M}_i \daleth^\mathcal{Q} \mathcal{M}_j$, iff $\mathcal{Q}(\mathcal{M}_i) \neq \mathcal{Q}(\mathcal{M}_j)$.

Definition 3. Two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are said to be *distinguishable*, denoted as $\mathcal{M}_i \daleth \mathcal{M}_j$, iff there exists a query that can distinguish between them, i.e., $\exists \mathcal{Q}\ \mathcal{M}_i \daleth^\mathcal{Q} \mathcal{M}_j$.

Given a pair of abstract models, we wish to determine whether one of them can be pruned, i.e., whether there is a query for which at least one of their answers 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 of their response with that of the agent. If the models are not distinguishable, it is redundant to try to prune one of them under the given query class.

Next, we determine if at least one of the two distinguishable models is consistent with the agent. When comparing the responses of two models at different levels of abstraction, we must consider the fact that the agent's response may be at a different level of abstraction if the given pair of models is abstract. Taking this into account, we formally define what it means for an abstract model $\mathcal{M}_i$'s response to be consistent with that of agent model $\mathcal{M}^\mathcal{A}$:

Definition 4. Let $\mathcal{Q}$ be a query such that $\mathcal{M}_i \daleth^\mathcal{Q} \mathcal{M}_j$; $\mathcal{Q}(\mathcal{M}_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle\rangle$, $\mathcal{Q}(\mathcal{M}_j) =$

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

```
at(?t,?l),      ⟶    in(?p,?t),
at(?p,?l)            ¬(at(?p,?l))
```

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

```
at(?t,?l),      ⟶    in(?p,?t)
at(?p,?l)
```

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

```
at(?t,?l)       ⟶    in(?p,?t)
```

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

```
at(?t,?l)       ⟶    ()
```

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

$\langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. $\mathcal{M}_i$'s response to $\mathcal{Q}$ is *consistent* with that of $\mathcal{M}^{\mathcal{A}}$, i.e., $\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i)$ if $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $len(\pi^{\mathcal{Q}}) = \ell^i$ and $\{p_1^i, \ldots, p_m^i\} \subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$.

Using this notion of consistency, we can now reason that given a set of distinguishable models $\mathcal{M}_i$ and $\mathcal{M}_j$, and their responses in addition to the agent's response to the distinguishing query, the models are prunable if and only if exactly one of their responses is consistent with that of the agent. Formally, we define prunability as:

Definition 5. Given an agent-interrogation task $\langle \mathcal{M}^{\mathcal{A}}, \mathbb{Q}, \mathbb{P}, \mathbb{A}_H \rangle$, two models $\mathcal{M}_i$ and $\mathcal{M}_j$ are *prunable*, denoted as $\mathcal{M}_i \langle\rangle \mathcal{M}_j$, iff $\exists \mathcal{Q} \in \mathbb{Q} : \mathcal{M}_i \text{l}^{\mathcal{Q}} \mathcal{M}_j \wedge (\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_j)) \vee (Q(\mathcal{M}^{\mathcal{A}}) \not\models \mathcal{Q}(\mathcal{M}_i) \wedge \mathcal{Q}(\mathcal{M}^{\mathcal{A}}) \models \mathcal{Q}(\mathcal{M}_j))$.

3.1 Solving the Interrogation Task

We now discuss how we solve the agent interrogation task by incrementally adding palm variants to the class of abstract models and pruning out inconsistent models by generating distinguishing queries.

Example 1. Consider the case of a delivery agent. Assume that AAM is considering two abstract models $\mathcal{M}_1$ and $\mathcal{M}_2$ having only one action $load\_truck(?p, ?t, ?l)$ and the predicates $at(?p, ?l)$, $at(?t, ?l)$, $in(?p, ?t)$, and that the agent's model is $\mathcal{M}^{\mathcal{A}}$ (Fig. 3). AAM can ask the agent what will happen if $\mathcal{A}$ loads 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 would be $at(t1, l1) \wedge in(p1, t1)$.

Algorithm 1 shows AAM's overall algorithm. It takes the agent $\mathcal{A}$, the set of instantiated predicates $\mathbb{P}^*$, the set of all action headers $\mathbb{A}_H$, and a set of random states $\mathbb{S}$ as input, and gives the set of functionally equivalent estimated models represented by $poss\_models$ as output. $\mathbb{S}$ can be generated in a preprocessing step given $\mathbb{P}^*$. AIA initializes $poss\_models$ as a set consisting of the empty model ϕ (line 3) representing that AAM is starting at the supremum $\top$ of the model lattice.

In each iteration of the main loop (line 4), AIA maintains an abstraction lattice and keeps track of the current node in the lattice. It picks a pal tuple γ corresponding to one of the descending edges in the lattice from a node given by some input ordering of Γ. The correctness of the algorithm does not depend on this ordering. It then stores a temporary copy of $poss\_models$ as new_models (line 5) and initialize an empty set at each node to store the pruned models (line 6).

Algorithm 1 Agent Interrogation Algorithm (AIA)

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

The inner loop (line 7) iterates over the set of all possible abstract models that AIA has not rejected yet, stored as new_models. It then loops over pairs of modes (line 8), which are later used to generate queries and refine models. For the chosen pair of modes, *generate_query()* is called (line 9) which returns two models concretized with the chosen modes and a query $\mathcal{Q}$ which can distinguish between them based on their responses.

AIA then calls *filter_models()* which poses the query $\mathcal{Q}$ to the agent and the two models. Based on their responses, AIA prunes the models whose responses are not consistent with that of the agent (line 11). Then it updates the estimated set of possible models represented by $poss\_models$ (line 18).

If AIA is unable to prune any model at a node (line 14), it modifies the pal tuple ordering (line 15). AIA continues this process until it reaches the most concretized node of the lattice (meaning all possible palm tuples $\lambda \in \Lambda$ are refined at this node). The remaining set of models represents the estimated set of models for $\mathcal{A}$. The number of resolved palm tuples can be used as a running estimate of accuracy of the derived models. AIA requires $O(|\mathbb{P}^*| {\times} |\mathbb{A}|)$ queries as there are $2 \times |\mathbb{P}^*| {\times} |\mathbb{A}|$ pal tuples. However, our empirical studies show that we never generate so many queries.

3.2 Query Generation

The query generation process corresponds to the *generate_query()* module in AIA which takes a model $\mathcal{M}'$, the pal tuple γ, and 2 modes $i, j \in \{+, -, \emptyset\}$ as input; and returns the models $\mathcal{M}_i = \mathcal{M}' \cup \{\gamma^i\}$ and $\mathcal{M}_j = \mathcal{M}' \cup \{\gamma^j\}$, and a

Algorithm 2 Query Generation Algorithm

```
1: Input: M', i, j, γ, 𝕊
2: Output: Q, M_i, M_j
3: M_i, M_j ← add_palm(M', i, j, γ)
4: for s_I in 𝕊 do
5:     dom, prob ← get_planning_prob (s_I, M_i, M_j)
6:     π ← planner(dom, prob)
7:     Q ← ⟨s_I, π⟩
8:     if π then break end if
9: end for
10: return Q, M' ∪ {γ^i}, M' ∪ {γ^j}
```

plan outcome query $\mathcal{Q}$ distinguishing them, i.e., $\mathcal{M}_i \urcorner^{\mathcal{Q}} \mathcal{M}_j$.

Plan outcome queries have 2 components, an initial state $s_{\mathcal{I}}$ and a plan π. AIA gets $s_{\mathcal{I}}$ from the input set of random states $\mathbb{S}$ (line 4). Using $s_{\mathcal{I}}$ as the initial state, the idea is to find a plan, which when executed by $\mathcal{M}_i$ and $\mathcal{M}_j$ will lead them either to different states, or to a state where only one of them can execute the plan further. Later we pose the same query to $\mathcal{A}$ and prune at least one of $\mathcal{M}_i$ and $\mathcal{M}_j$. Hence, we aim to prevent the models inconsistent with the agent model $\mathcal{M}^{\mathcal{A}}$ from reaching the same final state as $\mathcal{M}^{\mathcal{A}}$ after executing the query $\mathcal{Q}$ and following a different state trajectory. To achieve this, we reduce the problem of generating a plan outcome query from $\mathcal{M}_i$ and $\mathcal{M}_j$ into a planning problem.

The reduction proceeds by creating temporary models $\mathcal{M}''_i$ and $\mathcal{M}''_j$. We now discuss how to generate them. We add the pal tuple $\gamma = \langle p, a, l \rangle$ in modes i and j to $\mathcal{M}'$ to get $\mathcal{M}'_i$ and $\mathcal{M}'_j$, respectively. If the location $l = \mathit{eff}$, we add the palm tuple normally to $\mathcal{M}'$, i.e., $\mathcal{M}'_m = \mathcal{M}' \cup \langle p, a, l, m \rangle$, where $m \in \{i, j\}$. If $l = \mathit{pre}$, we add a dummy predicate p_u in disjunction with the predicate p to the precondition of both the models. We then modify the models $\mathcal{M}'_i$ and $\mathcal{M}'_j$ further in the following way:

$$
\begin{aligned}
\mathcal{M}''_m = \mathcal{M}'_m \cup \{ &\langle p_u, a', l', + \rangle : \forall a', l' \; \langle a', l' \rangle \notin \\
&\{\langle a^*, l^* \rangle : \exists m^* \; \langle p, a^*, l^*, m^* \rangle \in \mathcal{M}' \}\} \\
\cup \{ &\langle p_u, a', l', - \rangle : \forall a', l' \; \langle a', l' \rangle \in \\
&\{\langle a^*, l^* \rangle : l^* {=} \mathit{eff} \wedge \exists m^* \langle p, a^*, l^*, m^* \rangle {\in} \mathcal{M}' \}\}
\end{aligned}
$$

p_u is added only for generating a distinguishing query and is not part of the models $\mathcal{M}_i$ and $\mathcal{M}_j$ returned by the query generation process. Without this modification, an inconsistent abstract model may have a response consistent with $\mathcal{A}$.

We now show how to reduce plan outcome query generation into a planning problem P_{PO} (line 5). P_{PO} uses conditional effects in its actions (in accordance with PDDL (McDermott et al. 1998; Fox and Long 2003)). The model used to define P_{PO} has predicates from both models $\mathcal{M}''_i$ and $\mathcal{M}''_j$ represented as $\mathcal{P}^{\mathcal{M}''_i}$ and $\mathcal{P}^{\mathcal{M}''_j}$ respectively, in addition to a new dummy predicate p_ψ. The action headers are the same as $\mathbb{A}_H$. Each action's precondition is a disjunction of the preconditions of $\mathcal{M}''_i$ and $\mathcal{M}''_j$. This makes an action applicable in a state s if either $\mathcal{M}''_i$ or $\mathcal{M}''_j$ can execute it in s. The effect of each action has 2 conditional effects, the first applies the effects of both $\mathcal{M}''_i$ and $\mathcal{M}''_j$'s action if preconditions of both $\mathcal{M}''_i$ and $\mathcal{M}''_j$ are true, whereas the second makes the dummy predicate p_ψ true if precondition of only one of $\mathcal{M}''_i$ and $\mathcal{M}''_j$ is true. Formally, we express this planning problem as $P_{PO} = \langle \mathcal{M}^{PO}, s_{\mathcal{I}}, G \rangle$, where $\mathcal{M}^{PO}$ is a model with predicates $\mathbb{P}^{PO} = \mathcal{P}^{\mathcal{M}''_i} \cup \mathcal{P}^{\mathcal{M}''_j} \cup p_\psi$, and actions $\mathbb{A}^{PO}$ where for each action $a \in \mathbb{A}^{PO}$, $\mathit{pre}(a) = \mathit{pre}(a^{\mathcal{M}''_i}) \vee \mathit{pre}(a^{\mathcal{M}''_j})$ and $\mathit{eff}(a)$ =

$$
\begin{aligned}
&(\mathit{when}\,(\mathit{pre}(a^{\mathcal{M}''_i}) \wedge \mathit{pre}(a^{\mathcal{M}''_j}))(\mathit{eff}(a^{\mathcal{M}''_i}) \wedge \mathit{eff}(a^{\mathcal{M}''_j}))) \\
&(\mathit{when}\,((\mathit{pre}(a^{\mathcal{M}''_i}) \wedge \neg \mathit{pre}(a^{\mathcal{M}''_j})) \vee \\
&\qquad (\neg \mathit{pre}(a^{\mathcal{M}''_i}) \wedge \mathit{pre}(a^{\mathcal{M}''_j})))\,(p_\psi)),
\end{aligned}
$$

The initial state $s_{\mathcal{I}} = s_{\mathcal{I}}^{\mathcal{M}''_i} \wedge s_{\mathcal{I}}^{\mathcal{M}''_j}$, where $s_{\mathcal{I}}^{\mathcal{M}''_i}$ and $s_{\mathcal{I}}^{\mathcal{M}''_j}$ are copies of all predicates in $s_{\mathcal{I}}$, and G is the goal formula expressed as $\exists p \; (p^{\mathcal{M}''_i} \wedge \neg p^{\mathcal{M}''_j}) \vee (\neg p^{\mathcal{M}''_i} \wedge p^{\mathcal{M}''_j}) \vee p_\psi$.

With this formulation, the goal is reached when an action in $\mathcal{M}''_i$ and $\mathcal{M}''_j$ differs in either a precondition (making only one of them executable in a state), or an effect (leading to different final states on applying the action). E.g., consider the models with differences in *load_truck*$(p1, t1, l1)$ as shown in Fig. 3. From the state $at(t1, l1) \wedge \neg at(p1, l1)$, $\mathcal{M}_2$ can execute *load_truck*$(p1, t1, l1)$ but $\mathcal{M}_1$ cannot. Similarly, in state $at(t1, l1) \wedge at(p1, l1)$, executing *load_truck*$(p1, t1, l1)$ will cause $\mathcal{M}^{\mathcal{A}}$ and $\mathcal{M}_1$ to end up in states differing in predicate $at(p1, l1)$. Hence, given the correct initial state, the solution to the planning problem P_{PO} will give the correct distinguishing plan.

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). The input to the planning problem P_{PO} consists of an initial state $s_{\mathcal{I}}$. If the planner can solve P_{PO} with initial state $s_{\mathcal{I}}$ to give a plan π, the distinguishing query is a combination of $s_{\mathcal{I}}$ and π. Similarly, if $\mathcal{M}_i \urcorner^{\mathcal{Q}_{PO}} \mathcal{M}_j$, then giving the initial state $s_{\mathcal{I}}$ as part of planning problem P_{PO}, the plan π will be a solution which is part of $\mathcal{Q}_{PO}$. □

3.3 Filtering Possible Models

This section describes the *filter_models()* module in Algorithm 1 which takes as input $\mathcal{M}^{\mathcal{A}}$, $\mathcal{M}_i$, $\mathcal{M}_j$, and the query $\mathcal{Q}$ (Sec. 3.2), and returns the subset $\mathcal{M}_{prune}$ which is not consistent with $\mathcal{M}^{\mathcal{A}}$.

First, AAM *poses the query* $\mathcal{Q}$ to $\mathcal{M}_i$, $\mathcal{M}_j$, and the agent $\mathcal{A}$. 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. 5, checking for prunability involves checking if response 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.

Theorem 2. Let $\mathcal{M}_i, \mathcal{M}_j \in \{\mathcal{M}_+, \mathcal{M}_-, \mathcal{M}_\emptyset\}$ be the models generated by adding the pal tuple γ to $\mathcal{M}'$ which is an abstraction of the true agent model $\mathcal{M}^{\mathcal{A}}$. Suppose $\mathcal{Q} = \langle s_{\mathcal{I}}^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models $\mathcal{M}_i, \mathcal{M}_j$, i.e. $\mathcal{M}_i \urcorner^{\mathcal{Q}} \mathcal{M}_j$, and the response of models $\mathcal{M}_i, \mathcal{M}_j$, and $\mathcal{M}^{\mathcal{A}}$ to the query $\mathcal{Q}$ are $\mathcal{Q}(\mathcal{M}_i) = \langle \ell^i, \langle p^i_1, \ldots, p^i_m \rangle \rangle$, $\mathcal{Q}(\mathcal{M}_j) = \langle \ell^j, \langle p^j_1, \ldots, p^j_n \rangle \rangle$, and

$\mathcal{Q}(\mathcal{M}^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \dots, p_k^{\mathcal{A}} \rangle \rangle$. When $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $\mathcal{M}_i$ is not an abstraction of $\mathcal{M}^{\mathcal{A}}$ if $len(\pi^{\mathcal{Q}}) \neq \ell^i$ or $\{p_1^i, \dots, p_m^i\} \not\subseteq \{p_1^{\mathcal{A}}, \dots, p_k^{\mathcal{A}}\}$.

Proof (Sketch). Proving by induction, the base case is adding a single pal tuple $\langle p, a, l \rangle$ to an empty model (which is a consistent abstraction of $\mathcal{M}^{\mathcal{A}}$) resulting in 3 models. The 2 models pruned based on Def. 4 can be shown to be inconsistent with $\mathcal{M}^{\mathcal{A}}$, leaving out the one consistent model. For the inductive step, it can be shown that after adding a pal tuple to a consistent model it is not consistent with $\mathcal{M}^{\mathcal{A}}$ only if it does not execute the full plan (the precondition is inconsistent), or if the end state reached by the model is not a subset of the state of the agent (the effect is inconsistent). □

If the models are prunable, then the palm tuple being added in the inconsistent model cannot appear in any model consistent with $\mathcal{A}$. As we discard such palm tuples at abstract levels (as depicted in Fig. 2 (a)), we prune out a large number of models down the lattice (as depicted in Fig. 2 (c)), hence we keep the intractability of the approach in check and end up asking less number of queries.

3.4 Updating PAL ordering

This section describes the *update_pal_ordering()* module in AIA (line 15). It is called when the query generated by *generate_query()* module is not executable by $\mathcal{A}$, i.e., $len(\pi^{\mathcal{Q}}) \neq \ell^{\mathcal{A}}$. E.g., consider two abstract models $\mathcal{M}_2$ and $\mathcal{M}_3$ being considered by AAM (Fig. 3). At this level of abstraction, AAM does not have knowledge of the 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 the agent $\mathcal{A}$ as its precondition $at(?p, ?l)$ is not satisfied, and hence we cannot discard any of the models.

Recall that in response to the plan outcome query we get the failed action $a_{\mathcal{F}} = \pi[\ell+1]$ and the final state $s_{\mathcal{F}}$. Since the query plan π is generated using $\mathcal{M}_i$ and $\mathcal{M}_j$ (which differ only in the newly added palm tuple), they both would reach the same state $\overline{s}_{\mathcal{F}}$ after executing first ℓ steps of π. Thus, we search $\mathbb{S}$ for a state $s \supset \overline{s}_{\mathcal{F}}$ where $\mathcal{A}$ can execute $a_{\mathcal{F}}$. Similar to Stern and Juba (2017), we infer that any predicate which is false in s will not appear in $a_{\mathcal{F}}$'s precondition in the positive mode. Next, we iterate through the set of predicates $p' \subseteq s \setminus \overline{s}_{\mathcal{F}}$ and add them to $\overline{s}_{\mathcal{F}}$ to check if $\mathcal{A}$ can still execute $a_{\mathcal{F}}$. Thus, on adding a predicate $p \in p'$ to the state $\overline{s}_{\mathcal{F}}$, if $\mathcal{A}$ cannot execute $a_{\mathcal{F}}$, we add p in negative mode in $a_{\mathcal{F}}$'s precondition, otherwise in $\emptyset$ mode. All pal tuples whose modes are correctly inferred in this way are therefore removed from the pal ordering.

Equivalent Models It is possible for AIA to encounter a pair of models $\mathcal{M}_i$ and $\mathcal{M}_j$ that are not prunable. In such cases, the models $\mathcal{M}_i$ and $\mathcal{M}_j$ are functionally equivalent and none of them can be discarded. Hence, both the models end up in the set $poss\_models$ in line 18 of AIA.

3.5 Correctness of Agent Interrogation Algorithm

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

Theorem 3. The Agent Interrogation Algorithm (algorithm 1) will always terminate and return a set of models, each of which are functionally equivalent to the agent's model $\mathcal{M}^{\mathcal{A}}$.

Proof (Sketch). Theorem 1 and Theorem 2 prove that whenever we get a prunable query, we discard only inconsistent models, thereby ensuring that no correct model is discarded. When we do not get a prunable query, we infer the correct precondition of the failed action using *update_pal_ordering()*, hence the number of refined palm tuples always increases with the number of iterations of AIA, thereby ensuring its termination in finite time. □

Domain	$\lvert\mathbb{P}^*\rvert$	$\lvert\mathbb{A}\rvert$	$\lvert\hat{\mathcal{Q}}\rvert$	$\mathbf{t}_\mu$ (ms)	$\mathbf{t}_\sigma$ (μs)
Gripper	5	3	17	18.0	0.2
Blocksworld	9	4	48	8.4	36
Miconic	10	4	39	9.2	1.4
Parking	18	4	63	16.5	806
Logistics	18	6	68	24.4	1.73
Satellite	17	5	41	11.6	0.87
Termes	22	7	134	17.0	110.2
Rovers	82	9	370	5.1	60.3
Barman	83	17	357	18.5	1605
Freecell	100	10	535	2.24†	33.4†

Table 1: The number of queries ($|\hat{\mathcal{Q}}|$), average time per query (t_μ), and variance of time per query (t_σ) generated by AIA with FD. Average and variance are calculated for 10 runs of AIA, each on a separate problem. †Time in sec.

4 Empirical Evaluation

We implemented AIA in Python to evaluate the efficacy of our approach.[3] In this implementation, initial states ($\mathbb{S}$, line 1 in Algorithm 1) were collected by making the agent perform random walks in a simulated environment. We used a maximum of 60 such random initial states for each domain in our experiments. The implementation assumes that the domains do not have any constants and that actions and predicates do not use repeated variables (e.g., $at(?v, ?v)$), although these assumptions can be removed in practice without affecting the correctness of algorithms. The implementation is optimized to store the agent's answers to queries; hence the stored responses are used if a query is repeated.

We tested AIA on two types of agents: symbolic agents that use models from the IPC (unknown to AIA), and simulator agents that report states as images using PDDLGym. We wrote image classifiers for each predicate for the latter series of experiments and used them to derive state representations for use in the AIA algorithm. All experiments were executed on 5.0 GHz Intel i9-9900 CPUs with 64 GB RAM running Ubuntu 18.04.

The analysis presented below shows that AIA learns the correct model with a reasonable number of queries, and compares our results with the closest related work,

[3]Code available at https://git.io/Jtpej

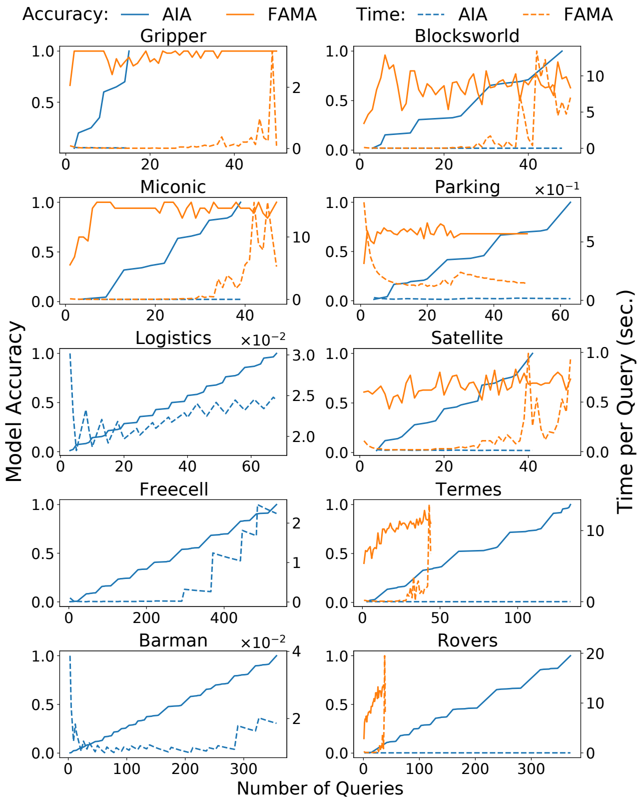

Figure 4: Performance comparison of AIA and FAMA in terms of model accuracy and time taken per query with an increasing number of queries.

FAMA (Aineto, Celorrio, and Onaindia 2019). We use the metric of *model accuracy* in the following analysis: the number of correctly learnt palm tuples normalized with the total number of palm tuples in $\mathcal{M}^{\mathcal{A}}$.

Experiments with symbolic agents We initialized the agent with one of the 10 IPC domain models, and ran AIA on the resulting agent. 10 different problem instances were used to obtain average performance estimates.

Table 1 shows that the number of queries required increases with the number of predicates and actions in the domain. We used Fast Downward (Helmert 2006) with LM-Cut heuristic (Helmert and Domshlak 2009) to solve the planning problems. Since our approach is planner-independent, we also tried using FF (Hoffmann and Nebel 2001) and the results were similar. The low variance shows that the method is stable across multiple runs.

Comparison with FAMA We compare the performance of AIA with that of FAMA in terms of stability of the models learnt and the time taken per query. Since the focus of our approach is on automatically generating useful traces, we provided FAMA randomly generated traces of length 3 (the length of the longest plans in AIA-generated queries) of the form used throughout this paper ($\langle s_{\mathcal{I}}, a_1, a_2, a_3, s_G \rangle$).

Fig. 4 summarizes our findings. AIA takes lesser time per query and shows better convergence to the correct model.

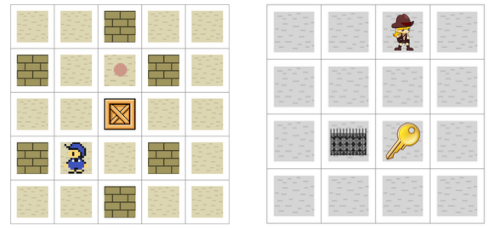

Figure 5: PDDLGym's simulated Sokoban (left) and Doors (right) environments used for the experiments.

FAMA sometimes reaches nearly accurate models faster, but its accuracy continues to oscillate, making it difficult to ascertain when the learning process should be stopped (we increased the number of traces provided to FAMA until it ran out of memory). This is because the solution to FAMA's internal planning problem introduces spurious palm tuples in its model if the input traces do not capture the complete domain dynamics. For Logistics, FAMA generated an incorrect planning problem, whereas for Freecell and Barman it ran out of memory (AIA also took considerable time for Freecell). Also, in domains with negative preconditions like Termes, FAMA was unable to learn the correct model. We used Madagascar (Rintanen 2014) with FAMA as it is the preferred planner for it. We also tried FD and FF with FAMA, but as the original authors noted, it could not scale and ran out of memory on all but a few Blocksworld and Gripper problems where it was much slower than with Madagascar.

Experiments with simulator agents AIA can also be used with simulator agents that do not know about predicates and report states as images. To test this, we wrote classifiers for detecting predicates from images of simulator states in the PDDLGym (Silver and Chitnis 2020) framework. This framework provides ground-truth PDDL models, thereby simplifying the estimation of accuracy. We initialized the agent with one of the two PDDLGym environments, Sokoban and Doors shown in Fig. 5. AIA inferred the correct model in both cases and the number of instantiated predicates, actions, and the average number of queries (over 5 runs) used to predict the correct model for Sokoban were 35, 3, and 201, and that for Doors were 10, 2, and 252.

5 Conclusion

We presented a novel approach for efficiently learning the internal model of an autonomous agent in a STRIPS-like form through query answering. Our theoretical and empirical results showed that the approach works well for both symbolic and simulator agents.

Extending our predicate classifier to handle noisy state detection, similar to prevalent approaches using classifiers to detect symbolic states (Konidaris, Kaelbling, and Lozano-Perez 2014; Asai and Fukunaga 2018) is a good direction for future work. Some other promising extensions include replacing query and response communication interfaces between the agent and AAM with a natural language similar to Lindsay et al. (2017), or learning other representations like Zhuo, Muñoz-Avila, and Yang (2014).

Acknowledgements

We thank Abhyudaya Srinet for his help with the implementation. This work was supported in part by the NSF under grants IIS 1844325, IIS 1942856, and OIA 1936997.

Ethics Statement

Learning the internal model of an AI agent is one of the main focus areas of the AI community in the recent past. This work would enable a layperson to assess such autonomous agents and to verify if they are safe to work with. This would increase the adaption rate of AI systems, as it would remove the dependence of systems using AI on experts who could verify the internal working of the agent.

Our system asks the agent queries and assumes that the agent can be connected to a simulator to ensure the correctness of responses. Our approach for such model learning comes with soundness and completeness guarantees. This implies that it will find the agent model if there exists one, and the model that it learns will be correct as per the simulations. As in any approach that uses simulators, this method is susceptible to errors in programming and simulator design. This can be addressed independently through research on formal verification of simulators used in AI.

References

Aineto, D.; Celorrio, S. J.; and Onaindia, E. 2019. Learning Action Models With Minimal Observability. *Artificial Intelligence* 275: 104–137.

Amir, E.; and Chang, A. 2008. Learning Partially Observable Deterministic Action Models. *Journal of Artificial Intelligence Research* 33: 349–402.

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

Asai, M.; and Fukunaga, A. 2018. Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary. In *Proc. AAAI*.

Bäckström, C.; and Jonsson, P. 2013. Bridging the Gap Between Refinement and Heuristics in Abstraction. In *Proc. IJCAI*.

Balac, N.; Gaines, D.; and Fisher, D. 2000. Learning Action Models for Navigation in Noisy Environments. In *ICML Workshop on Machine Learning of Spatial Knowledge*.

Bonet, B.; and Geffner, H. 2020. Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In *Proc. ECAI*.

Bryce, D.; Benton, J.; and Boldt, M. W. 2016. Maintaining Evolving Domain Models. In *Proc. IJCAI*.

Camacho, A.; and McIlraith, S. A. 2019. Learning Interpretable Models Expressed in Linear Temporal Logic. In *Proc. ICAPS*.

Chitnis, R.; Silver, T.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Perez, T. 2021. GLIB: Efficient Exploration for Relational Model-Based Reinforcement Learning via Goal-Literal Babbling. In *Proc. AAAI*.

Cresswell, S.; and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *Proc. ICAPS*.

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

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3-4): 189–208.

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

Gil, Y. 1994. Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains. In *Proc. ICML*.

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

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

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS*.

Helmert, M.; Haslum, P.; Hoffmann, J.; et al. 2007. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proc. ICAPS*.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

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

Konidaris, G.; Kaelbling, L. P.; and Lozano-Perez, T. 2014. Constructing Symbolic Representations for High-Level Planning. In *Proc. AAAI*.

Krishnan, A.; Williams, A.; and Martens, C. 2020. Towards Action Model Learning for Player Modeling. In *Proc. AIIDE*.

Kučera, J.; and Barták, R. 2018. LOUGA: Learning Planning Operators Using Genetic Algorithms. In *Knowledge Management and Acquisition for Intelligent Systems*.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning Models from Natural Language Action Descriptions. In *Proc. ICAPS*.

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

Ng, J. H. A.; and Petrick, R. P. A. 2019. Incremental Learning of Planning Actions in Model-Based Reinforcement Learning. In *Proc. IJCAI*.

Rintanen, J. 2014. Madagascar: Scalable Planning with SAT. In *Proc. 8th International Planning Competition*.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence* 5(2): 115–135.

Settles, B. 2012. *Active Learning*. Morgan & Claypool Publishers. ISBN 1608457257.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*.

Srivastava, S. 2021. Unifying Principles and Metrics for Safe and Assistive AI. In *Proc. AAAI*.

Srivastava, S.; Russell, S.; and Pinto, A. 2016. Metaphysics of Planning Domain Descriptions. In *Proc. AAAI*.

Stern, R.; and Juba, B. 2017. Efficient, Safe, and Probably Approximately Complete Learning of Action Models. In *Proc. IJCAI*.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning Action Models from Plan Examples Using Weighted MAX-SAT. *Artificial Intelligence* 171(2-3): 107–143.

Zhuo, H. H.; and Kambhampati, S. 2013. Action-Model Acquisition from Noisy Plan Traces. In *Proc. IJCAI*.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning Hierarchical Task Network Domains from Partially Observed Plan Traces. *Artificial Intelligence* 212: 134 – 157.