Data-Efficient Paradigms for Personalized Assessment of Taskable AI Systems

by

Pulkit Verma

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2024 by the
Graduate Supervisory Committee:

Siddharth Srivastava, Chair
Nancy Cooke
Georgios Fainekos
Yu Zhang

ARIZONA STATE UNIVERSITY

May 2024

ABSTRACT

Recent advances in Artificial Intelligence (AI) have brought AI closer to laypeople than ever before. This 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 it is considered that most autonomous systems are not designed by their users; the internal software of these systems may be unavailable or difficult to understand; and the functionality of these systems may even change from initial specifications as a result of learning. To overcome these challenges, this dissertation proposes a paradigm for third-party autonomous assessment of black-box taskable AI systems. The four main desiderata of such assessment systems are: (i) interpretability: generating a description of the AI system's functionality in a language that the target user can understand; (ii) correctness: ensuring that the description of AI system's working is accurate; (iii) generalizability creating a solution approach that works well for different types of AI systems; and (iv) minimal requirements: creating an assessment system that does not place complex requirements on AI systems to support the third-party assessment, otherwise the manufacturers of AI system's might not support such an assessment.

To satisfy these properties, this dissertation presents algorithms and requirements that would enable user-aligned autonomous assessment that helps the user understand the limits of a black-box AI system's safe operability. This dissertation proposes a personalized AI assessment module that discovers the high-level "capabilities" of an AI system with arbitrary internal planning algorithms/policies and learns an accurate symbolic description of these capabilities in terms of concepts that a user understands. Furthermore, the dissertation includes the associated theoretical results and the empirical evaluations. The results show that (i) a primitive query-response interface

can enable the development of autonomous assessment modules that can derive a causally accurate user-interpretable model of the system's capabilities efficiently; and (ii) such descriptions are easier to understand and reason with for the users than the agent's primitive actions.

DEDICATION

To my parents and sister, without whose unwavering support this would not have been possible.

ACKNOWLEDGMENTS

This work would not have been in this form without the unwavering guidance and support of Prof. Siddharth Srivastava. Siddharth has taught me the importance of using the correct technical terminology and correct formalism, which have made me a better researcher than I could have ever been without him. Throughout all of my learning curve and the various mistakes I made during my time as a doctoral student, Siddharth has always given me space to learn, and has been enormously patient. I plan to go into academia, and I would consider myself successful if, by the end of my career, I am half as good a mentor as him.

I would also like to thank my committee member Prof. Nancy Cooke, who has taught me the importance of not forgetting about humans when developing AI systems that have to work with humans. I am also indebted to Prof. Georgios Fainekos for incorporating aspects of temporal logic into my thesis. I would also like to thank Prof. Yu Zhang for his constant encouragement through the various impromptu interactions I had with him.

This arduous journey was also incomplete without the support and fun working environment my fellow labmates from Autonomous Agents and Intelligent Robots Lab: Naman Shah, Rushang Karia, Kislay Kumar, Midhun Pookkotil Madhusoodanan, Rashmeet Kaur Nayyar, Chirav Dave, Daniel Molina, Shashank Rao Marpally, Abhyudaya Srinet, Deepak Kala Vasudevan, Kiran Prasad, Mehdi Dadvar, Trevor Angle, Daniel Bramblett, Kyle Joseph Atkinson, Gaurav Vipat, Dylan Fulop, Jayesh Nagpal, Shivanshu Verma, and Daksh Dobhal. They have really made my PhD life a fun journey despite the numerous late night (and sometimes overnight) research discussions. I also thank Sarath Sreedharan, Sachin Grover, Swaroop Mishra, Neeraj Varshney, and Yantian Zha, who despite not being from my lab have always been the ones I

could fall back on for research discussions and often advice. My research style also improved a lot while working at Meta AI with Rohan Chitnis, Alborz Geramifard, Nitin Kamra, and Harshit Sikchi.

I was also grateful to have a large group of friends around my lab who kept me updated on the happenings in the world of AI outside my area of expertise, including Akkamahadevi Hanni, Andrew Boateng, Kuntal Kumar Pal, Paras Sheth, Mihir Parmar, Mirali Purohit, Aranyak Maity, Garima Agrawal, Tasneema Fahim Azad, Kevin Jatin Vora, Maitry Trivedi, Pratanu Mandal, Tejas Gokhale, Man Luo, Mudit Verma, Kartik Valmeekam, Anjana Arunkumar, Sriraam Gopalakrishnan, Zahra Zahedi, and Alberto Olmo. PhD is a long journey and sometimes you end up losing a lot of friendships. Despite this, I was fortunate to have a great support network outside of my PhD life who kept me grounded including Vinodhini SD, Shruthi Nagaraj, Prateek Agrawal, Ankit K Jain, Avisha Das, Manoj Bode, Paras Vishnoi, Tuhin Bhattacharya, Sahiba Sachdev, Bharathwaj G, Anurag Rawat, Vedika Verma, Apoorv Shrivastava, and last but not the least, Malpooa.

This section would not be complete without a mention to my mentor in high school Dr. Swati Amalnerkar, who supported and motivated me continuously and was the first person who explained me what a PhD is and what it means to have one. And finally, I would not be writing this, if not for the unconditional and constant support of my parents, Geeta Verma and Yogesh Kumar Verma, my sister Prachi Verma, and all of my family. This PhD has taken a lot of patience on their part.

I would have loved to go into the details of how each one of the many people I named helped me in my PhD journey, but unfortunately ASU allows for only three pages of dedication and acknowledgements. Finally, I thank NSF and ONR for funding the research presented in this dissertation.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Chapter 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 capability descriptions, or *models* may not be available in terminology that the user understands.

This issue becomes even more complex when we have an AI agent that is evolving and adapting to changes in the environment it is operating in? And how do we ensure its reliable and safe usage? Numerous factors could cause unpredictable changes in agent behaviors: sensors and actuators may fail due to physical damage, the agent may adapt to a dynamic environment, users may change deployment and use-case scenarios, etc. Most prior work on the topic presumes that the functionalities and the capabilities of AI agents are static, while some works start with a *tabula-rasa* and learn the entire model from scratch. However, in many real-world scenarios, the agent model is transient and only parts of its functionality change at a time.

To alleviate these issues, 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).

Ongoing research on the topic focuses on the significant problem of how to answer users' questions about the system's behavior while assuming that the user and AI share a common action vocabulary (Chakraborti *et al.*, 2017; Dhurandhar *et al.*, 2018; Anjomshoae *et al.*, 2019; Barredo Arrieta *et al.*, 2020). Furthermore, most non-experts hesitate to ask questions about new AI tools (Mou and Xu, 2017) and often do not know which questions to ask for assessing the safe limits and capabilities of an AI system. This problem is aggravated in situations where an AI system can carry out planning or sequential decision making. Lack of understanding about the limits of an imperfect system can result in unproductive usage or, in the worst-case, serious accidents (Randazzo, 2018). This, in turn, limits the adoption and productivity of AI systems.

This dissertation presents a new paradigm for third-party assessment of black-box taskable AI systems. It develops algorithms 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; 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]

[1]Just 2 actions and 5 grounded propositions would yield $7^{2\times5} \sim 10^8$ possible STRIPS-like models

Most current approaches to learn such interpretable descriptions in form of STRIPS-like languages use observations of the AI system's interaction with the environment as input. In this dissertation, we show that such approaches end up learning spurious correlations as part of the description. A few approaches that generate the observations themselves are not sample efficient and hence require more time and observations than the methods we introduce in this dissertation to learn an accurate description of the AI system. Finally, all these approaches do not specifically perform any kind of assessment of the AI systems, but focus on learning STRIPS-like models of the AI systems. We use this idea of learning such models, but use active interrogation of these AI systems to learn the models so as to have finer control over the data generation (in form of observations) process. This novel idea of using responses to well-directed queries about the AI systems' behavior leads to learning accurate models with fewer data. Additionally, we also define what is third-party assessment of an AI system, and use model learning as a tool to assess black-box taskable AI systems.

Before delving deeper into the specific problem settings, algorithms and results, I will first present the important properties that a third-party assessment system should satisfy and the outline of the contents of this thesis.

1.1 Desiderata for Third-Party Assessment

As mentioned earlier, there are four desiderata for a third-party assessment system that must be developed for any black-box AI systems.

– 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.

Interpretability An autonomous assessment system should be able to compute a user interpretable model of the black-box AI system's capabilities. Here "interpretable" may refer to a description that a user understands, it can be in terms of concepts that (i) a user understands, or (ii) a user can be easily trained on.

Correctness The output generated by the assessment system should be accurate. At least the most accurate description that can be generated in terms of concepts that a user understands. This is important because when something about the AI system's working is described to the user and it has errors, then the user's trust on the system might be affected, which will be detrimental to the adoptability of the system.

Generalizability The autonomous assessment system should be able to work with wide variety of AI systems. Hence its working should be independent of the internals of the black-box AI system. It should also not depend on the deployed environment and should be able to work with reasonable designs for any taskable AI system.

Minimal Requirements Since the assessment system is intended to be used by people who have not developed the AI systems being assessed, it should not put a lot of requirements on the design or implementation of the black-box AI systems to support the assessment. Otherwise, the manufacturers would not want to support such an assessment at the cost of throttling innovation.

To fulfill these desiderata we leverage (i) the framework of state abstractions to discover the high-level interpretable capabilities of a taskable AI system; and (ii) the framework of planning domain descriptions to learn and express the symbolic models of each of the discovered capabilities. This dissertation builds algorithms and techniques to use these frameworks for assessment and the next section presents this dissertation's high level outline.

1.2 Thesis Outline

The next chapter gives a background of the kind of interpretable models that are learned as output of the assessment system. This deals with the desideratum of interpretability discussed in the previous section. The rest of the dissertation is organized as follows.

- Chapter 3 discusses the design and formal requirements that an assessment system should satisfy, defines a query-response interface, and presents an algorithm for assessment of an AI system in deterministic, stationary, and fully observable settings. The desideratum of minimal requirements is set up and discussed in this chapter. This chapter builds upon our publications Verma and Srivastava (2020) and Verma *et al.* (2021a).
- Chapter 4 extends the framework for assessment in settings where the AI system's model is not stationary, i.e., it is an adaptive AI system. Rest of the assumptions of deterministic and fully observable system still holds. This and the next three chapters show how we address the desiderata of correctness and generalizability. This chapter builds upon our publication Nayyar *et al.* (2022).
- Chapter 5 discovers the high-level capabilities of the agent and then learn their description, instead of learning the description of low-level actions of the agent. This chapter builds upon our publications Verma *et al.* (2021b) and Verma *et al.* (2022).
- Chapter 6 describes the assessment framework for the stochastic settings and learns the description of its probabilistic capabilities. This chapter builds upon our publication Verma *et al.* (2023a).
- Chapter 7 combines shows that the models we learn for deterministic, full

observable, stationary settings are causally accurate. This chapter also defines the notion of causal accuracy for our setting. This chapter also addresses the desideratum of correctness in more detail as compared to the previous chapters. This chapter builds upon our publications Verma and Srivastava (2021) and Verma and Srivastava (2024a).

- Chapter 8 performs an extended analysis of complexity of the queries we use for deterministic, full observable, stationary settings. This chapter builds upon our publication Verma and Srivastava (2024a).
- Chapter 9 uses the description learned for assessment to direct exploration of reinforcement learning agents to make it sample efficient. This chapter builds upon our publications Karia *et al.* (2023) and Karia *et al.* (2024b).

Chapter 10 concludes the thesis with my learnings and possible extensions of this dissertation work. A preliminary version of this discussion was presented in Verma and Srivastava (2024b).

Chapter 2

BACKGROUND

We express the descriptions of an AI system's capabilities using a STRIPS-like representation (Fikes and Nilsson, 1971). This is because, when used with a user's vocabulary, such a representation can be readily transcribed into statements such as "in situations where X holds, if the agent executes actions $a_1, \ldots, a_k$ it would result in Y", where X and Y are in the user's vocabulary (Camacho and McIlraith, 2019). Such representations have been shown to be intuitive for humans in understanding deliberative behaviors of other agents (Malle, 2004; Miller, 2019).

In this chapter, we briefly discuss the formalism of the planning models and then describe some of the common terminology that we use in the thesis.

2.1 Planning Models

We start with the deterministic models expressible in the Planning Domain Definition Language (PDDL) (McDermott *et al.*, 1998), then explain the Probabilistic Planning Domain Definition Language (PPDDL) models, followed by the Fully Observable Non-Deterministic (FOND) models.

2.1.1 PDDL Models

Deterministic models expressed using the Planning Domain Definition Language (PDDL) (McDermott *et al.*, 1998) are formally defined as:

Definition 1. A ***deterministic planning model*** is a tuple $M = \langle P, A \rangle$, where $P = \{p_1^{r_1}, \ldots, p_n^{r_n}\}$ is a finite set of predicates with arities $r_i, i \in [1, n]$; $A = \{a_1, \ldots, a_k\}$ is a finite set of parameterized actions (operators). Each action $a_j \in A$ is represented as a 3-tuple $\langle \mathit{header}(a_j), \mathit{pre}(a_j), \mathit{eff}(a_j) \rangle$, where $\mathit{header}(a_j)$ is the action header consisting of action name and action parameters, $\mathit{pre}(a_j)$ represents the set of positive and negative predicate atoms that must be true or false, respectively 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 a_j.

In the rest of the dissertation, we use the term "model" to refer to planning models. Given a model M and a set of objects O, let $S_{M,O}$ be the space of all states defined as maximally consistent sets of literals over the predicate vocabulary of M with O as the set of objects. We omit the subscript when it is clear from the context. An action $a \in A$ is applicable in a state $s \in S$ if $s \models \mathit{pre}(a)$. The result of executing a is a state $a(s) = s' \in S$ such that $s' \models \mathit{eff}(a)$, and all atoms not in $\mathit{eff}(a)$ have literal forms as in s. We extend this notation to express the result of executing a plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ in a state s, i.e., $a_n(\ldots a_2(a_1(s)) \ldots) = s_n$ as $\pi(s) = s_n$.

2.1.2 PPDDL Models

Younes and Littman (2004) formalized as Probabilistic Planning Domain Definition Language (PPDDL). We assume that the model of the agent $\mathcal{A}$ can be represented in PPDDL as $M^{\mathcal{A}}$, and it is formally defined as:

Definition 2. A ***probabilistic-planning model*** is a tuple $M = \langle P, A \rangle$, where $P = \{p_1^{r_1}, \ldots, p_k^{r_k}\}$ is a finite set of predicates with arities $r_j, j \in [1, k]$; and A is a set of parameterized actions. Each $a \in A$ is represented as a 3-tuple

$\langle header(a), pre(a), \mathit{eff}(a)\rangle$, where $header(a)$ is the action header consisting of the action name and parameters, $pre(a)$ represents a set of positive and negative predicates that must be true or false, respectively, in a state where a can be applied, and $\mathit{eff}(a)$ is a set of effect pairs $\langle e_i(a), pr_i(a)\rangle$. Here $e_i(a)$ is a set of positive or negative predicate atoms that will change to true or false, respectively, each with a probability $pr_i(a) \in [0, 1]$ as a result of execution of the action a.

The only major difference between PDDL and PPDDL models that we learn is that in PPDDL, effect is a set of conjuctive formula, each with an associated probability, whereas in PDDL the effect is just one conjunctive formula.

2.1.3 FOND Models

We also use a fully-observable non-deterministic (FOND) planning model (Cimatti *et al.*, 1998). This can be viewed as a probabilistic planning model without the probabilities associated with each effect pair. Hence on executing an action, one of its possible effects is chosen with an equal probability. The solution to these planning models is a *partial policy* $\Pi : S \rightarrow A$ that maps each state to an action that the agent should execute in that state. As shown by Cimatti *et al.* (1998) and Daniele *et al.* (1999), the solution is a (i) *weak solution* if the resulting plan may achieve the goal without any guarantee; (ii) *strong solution* if the resulting plan is guaranteed to reach the goal; and (iii) *strong cyclic solution* if the resulting plan is guaranteed to reach the goal under the assumption that in the limit, each action will lead to each of its effects.

2.2 Common Terminology

This section describes some of the common terminology that we use throughout the thesis.

2.2.1 Lifted Instantiated Predicate

Each predicate can be instantiated using the parameters of an action. The number of action parameters is 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. The predicate `(at ?x ?y)` 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 lifted instantiated predicates P^*.

2.2.2 Observations

We compare our approach to the class of model learners that use the observations generated by the agent to learn the agent model. Such observations are defined as:

Definition 3. Given a state space S, and a set of actions A, an ***observation trace*** o is an alternating sequence of states and actions of the form $\langle s_0, a_1, s_1, a_2, \ldots, s_{n-1}, a_n, s_n \rangle$ such that $s_i \in S$, $a_i \in A$, and $\forall i \in [1, n]$ $a_i(s_{i-1}) = s_i$.

These observation traces can be split into multiple action triplets (Stern and Juba, 2017) as defined below.

Definition 4. Given an observation trace $o = \langle s_0, a_1, s_1, a_2, \ldots, s_{n-1}, a_n, s_n \rangle$, an ***action triplet*** is a 3-tuple sub-sequence of o of the form $\langle s_{i-1}, a_i, s_i \rangle$, where $i \in [1, n]$ and applying an action a_i in state s_{i-1} results in state s_i, i.e., $a_i(s_{i-1}) = s_i$.

The states s_{i-1} and s_i are called pre- and post-states of action a_i, respectively.

Chapter 3

AGENT INTERROGATION

This chapter proposes a new approach for estimating an interpretable and 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. Empirical evaluation of our approach shows that despite the massive number of possible agent models, our approach results in the 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.

3.1 Personalized Assessment

In this section, we propose a personalized agent-assessment module (AAM), shown in Fig. 1, which can be connected with an arbitrary AI agent that supports 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

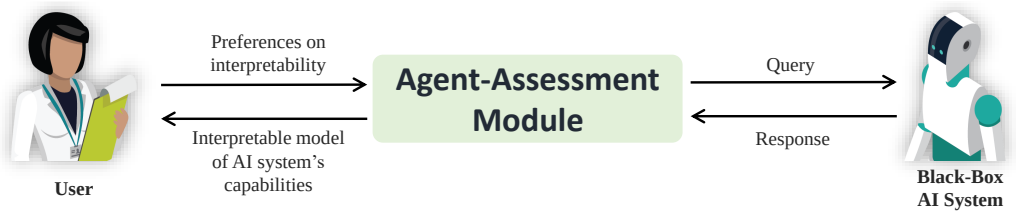


Figure 1. The agent-assessment module uses its user's preferred vocabulary, queries the AI system, and delivers a user-interpretable correct causal model of the AI system's capabilities. The AI system does not need to know the user's vocabulary or modeling language.

need to specify their instruction sets in order to be usable. The user provides the predicates or concepts that they can understand and these concepts can be defined as functions on simulator states.

This fundamental framework (Sec. 3.4) 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 the assessment module to pose queries such as "what do you think would happen if you did $\langle$*query plan*$\rangle$", and the learned 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 learned models and their accuracy. Using the assessment module 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. 3.4) 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. 3.4 presents our overall framework with algorithms and theoretical results about their convergence properties.

Our empirical evaluation (Sec. 3.6) shows that this method can efficiently learn correct models for black-box versions of agents using hidden models from the International Planning Competition (IPC)[2]. It also shows that the agent assessment module 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.

3.2 Preliminaries

Models learned by the agent assessment module are in the form of PDDL models defined in chapter 2. In the rest of the chapter, we use the term "model" to refer to planning models. Given a model M and a set of objects O, let $S_{M,O}$ be the space of all states defined as maximally consistent sets of literals over the predicate vocabulary of M with O as the set of objects. We omit the subscript when it is clear from the context. An action $a \in A$ is applicable in a state $s \in S$ if $s \models \mathit{pre}(a)$. The result of executing a is a state $a(s) = s' \in S$ such that $s' \models \mathit{eff}(a)$, and all atoms not in $\mathit{eff}(a)$ have literal forms as in s. We extend this notation to express the result of executing a plan $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ in a state s, i.e., $a_n(\ldots a_2(a_1(s))\ldots) = s_n$ as $\pi(s) = s_n$.

Lifted instantiated predicate Each predicate can be instantiated using the parameters of an action. The number of action parameters is

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

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. The predicate `(at ?x ?y)` 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 lifted instantiated predicates P^*.

3.3 Formal Framework

As noted in introduction, the agent assessment module uses the following information as input: (i) the instruction set from the agent in the form of $header(a)$ for each $a \in A$; and (ii) a predicate vocabulary P from the user with functional definitions of each predicate. This gives the assessment module sufficient information to perform a dialog with $\mathcal{A}$ about the outcomes of hypothetical action sequences. This dialog is performed in terms of queries and responses.

3.3.1 Form of Agent Queries

As mentioned earlier, the assessment module poses queries to the agent and based on $\mathcal{A}$'s responses θ it infers $\mathcal{A}$'s agent model. We express queries as functions that map models to answers.

Definition 5. Given a set of predicates P and a set A of actions, let $\mathcal{U}$ be the set of all possible planning models (ref. Def. 1) expressible using P and A. Let Θ be the set of possible responses. A ***query*** q is a function $q : \mathcal{U} \to \Theta$.

In this chapter, we utilize only one class of queries: *plan outcome queries* (Q_{PO}), which are parameterized by a state s_I and a plan π. Let P^* be the set of predicates P instantiated with objects O in an environment. 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 $s_I \subseteq P^*$ as well as the final state $s_F \subseteq P^*$ 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 $\langle \ell, s_F \rangle$, where ℓ is the number of steps for which the plan π could be executed, and $s_F \subseteq P^*$ is the final state after executing ℓ steps of the plan. If the plan π cannot be executed fully according to the agent model $M^{\mathcal{A}}$ then $\ell < len(\pi)$, otherwise $\ell = len(\pi)$. The final state $s_F \subseteq P^*$ is such that $M^{\mathcal{A}} \models \pi[1:\ell](s_I) = s_F$, i.e., starting with a state s_I, $M^{\mathcal{A}}$ successfully executed first ℓ steps of the plan π. Thus, $Q_{PO} : \mathcal{U} \rightarrow \mathbb{N} \times 2^P$, where $\mathbb{N}$ is the set of natural numbers.

3.3.2 Requirements for Independent Assessment

The requirements in an AI agent might change depending on the type of queries the agent is capable of answering. This is because we define the set of requirements as a function of the agent. Formally, we define the requirements on an agent as:

Definition 6. Given a query class Q, with an associated response set Θ, the ***assessment requirement*** $\rho_{\mathcal{A}}$ on an autonomous agent $\mathcal{A}$ is a relation between Q and Θ, and is represented as $\rho_{\mathcal{A}}\langle Q, \Theta \rangle$.

Given a plan-outcome query $q = \langle s_0, \pi \rangle$, where $\pi = \langle a_1, \ldots, a_n \rangle$, an autonomous agent $\mathcal{A}$ is said to support the set of requirements $\rho_{\mathcal{A}}$ if its response $\theta = \langle \ell, s_\ell \rangle$ satisfies:

$$\begin{aligned} \rho_{\mathcal{A}}(\langle s_0, \langle a_1, \ldots, a_n \rangle \rangle, \langle \ell, s_\ell \rangle) \triangleq \ell < n \wedge \\ \forall i \in 1, \ldots, \ell - 1, \exists s_i \ \mathcal{A} \models a_i(s_{i-1}) = s_i \wedge \mathcal{A} \models \neg(pre(a_{\ell+1}) \wedge s_\ell) \end{aligned} \tag{3.1}$$

We now define the overall problem of agent interrogation as follows. Given a class of queries and an agent with an unknown model supports the plan outcome query requirement above (1), determine the model of the agent. This can be formally defined as:

Definition 7. An ***agent interrogation task*** is defined as a quadruple $\langle \mathcal{A}, Q, P, A_H, \rho_{\mathcal{A}} \rangle$, where $\mathcal{A}$ is the agent being interrogated, Q is the class of queries that can be posed to the agent by the assessment module,P and A_H are the sets of predicates and action headers that the assessment module uses based on inputs from $\mathcal{H}$ and $\mathcal{A}$, and $\rho_{\mathcal{A}}$ is the assessment requirement that $\mathcal{A}$ must satisfy.

The objective of our solution to the the agent interrogation task is to derive $\mathcal{A}$'s agent model $M^{\mathcal{A}}$ using Q, P, and A_H. We now introduce a running example which we'll use throughout the chapter.

Running Example Consider a driving robot with a single action `drive(?t ?s ?d)`, parameterized by the truck it drives, source location, and destination location. Assume that all locations are connected, hence the robot can drive between any two locations. The predicates available are `(at ?t ?loc)`, representing the location of a truck; and `(src_blue ?loc)`, representing the color of the source location. Instantiating `at` and `src_blue` with parameters of the action *drive* gives four instantiated predicates `(at ?t ?s)`, `(at ?t ?d)`, `(src_blue ?s)`, and `(src_blue ?d)`.

3.3.3 Distinguishability and Prunability

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 q is said to *distinguish* a pair of models M_i and M_j, denoted as $M_i \mathrm{l}^q M_j$, iff $q(M_i) \neq q(M_j)$.

Definition 8. Two models M_i and M_j are said to be ***distinguishable***, denoted as $M_i \mathrm{l}\, M_j$, iff there exists a query that can distinguish between them, i.e., $\exists \mathcal{Q}\ M_i \mathrm{l}^{\mathcal{Q}} M_j$.

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 won't be possible 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 M_i's response to be consistent with that of agent model $M^{\mathcal{A}}$:

Definition 9. Let $\mathcal{Q}$ be a query such that $M_i \mathrm{l}^{\mathcal{Q}} M_j$; $\mathcal{Q}(M_i) = \langle \ell^i, s^i \rangle$, $\mathcal{Q}(M_j) = \langle \ell^j, s^j \rangle$,

and $\mathcal{Q}(M^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, s^{\mathcal{A}} \rangle$. M_i's response to $\mathcal{Q}$ is said to be ***consistent*** with that of $M^{\mathcal{A}}$, i.e., $\mathcal{Q}(M^{\mathcal{A}}) \models \mathcal{Q}(M_i)$ iff $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $len(\pi^{\mathcal{Q}}) = \ell^i$ and $s^i \subseteq s^{\mathcal{A}}$.

Using this notion of consistency, we can now reason that given a set of distinguishable models M_i and 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 10. Given an agent-interrogation task $\langle M^{\mathcal{A}}, Q, P, A_H \rangle$, two models M_i and M_j are ***prunable***, denoted as $M_i \Diamond M_j$, iff $\exists \mathcal{Q} \in Q : M_i \wr^{\mathcal{Q}} M_j \wedge (\mathcal{Q}(M^{\mathcal{A}}) \models q(M_i) \wedge \mathcal{Q}(M^{\mathcal{A}}) \not\models \mathcal{Q}(M_j)) \vee (\mathcal{Q}(M^{\mathcal{A}}) \not\models \mathcal{Q}(M_i) \wedge \mathcal{Q}(M^{\mathcal{A}}) \models \mathcal{Q}(M_j))$.

3.3.4 Components of Agent Models

In order to formulate our solution approach, we consider a model 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 P^*; a is an action from the set of parameterized actions A, $l \in \{pre, eff\}$, 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 mode 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., $\lambda_1 \sim \lambda_2 \Leftrightarrow (\lambda_{1_p} = \lambda_{2_p}) \wedge (\lambda_{1_a} = \lambda_{2_a}) \wedge (\lambda_{1_l} = \lambda_{2_l}) \wedge (\lambda_{1_m} \neq \lambda_{2_m})$.

We also define the notion of *pal* tuples which are represented a 3-tuple $\langle p, a, l \rangle$. Each pal tuple $\gamma = \langle p, a, l \rangle$ corresponds to three pal tuples λ_m, $m \in \{+, -, \emptyset\}$, such that $\gamma_p = \lambda_{m_p}$, $\gamma_a = \lambda_{m_a}$, and $\gamma_l = \lambda_{m_l}$. A 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 $\gamma^\emptyset = \langle p, a, l, \emptyset \rangle$.

For a model $M = \langle P, A \rangle$, the set of all possible palm tuples and pal tuples that can be generates using $p \in P$ and $a \in A$ are represented as Λ and Γ, respectively.

Example 1. Based on the running example, possible pal tuples are:

- $\langle$`(at ?t ?s)`, `drive(?t ?s ?d)`, `pre`$\rangle$
- $\langle$`(at ?t ?s)`, `drive(?t ?s ?d)`, `eff`$\rangle$
- $\langle$`(at ?t ?d)`, `drive(?t ?s ?d)`, `pre`$\rangle$
- $\langle$`(at ?t ?d)`, `drive(?t ?s ?d)`, `eff`$\rangle$
- $\langle$`(src_blue ?s)`, `drive(?t ?s ?d)`, `pre`$\rangle$
- $\langle$`(src_blue ?s)`, `drive(?t ?s ?d)`, `eff`$\rangle$
- $\langle$`(src_blue ?d)`, `drive(?t ?s ?d)`, `pre`$\rangle$
- $\langle$`(src_blue ?d)`, `drive(?t ?s ?d)`, `eff`$\rangle$

3.3.5 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); Helmert *et al.* (2007); Bäckström and Jonsson (2013); Srivastava *et al.* (2016). The definition of abstraction used in this work extends the concept of predicate and propositional domain abstractions Srivastava *et al.* (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. We define abstraction of a model as:

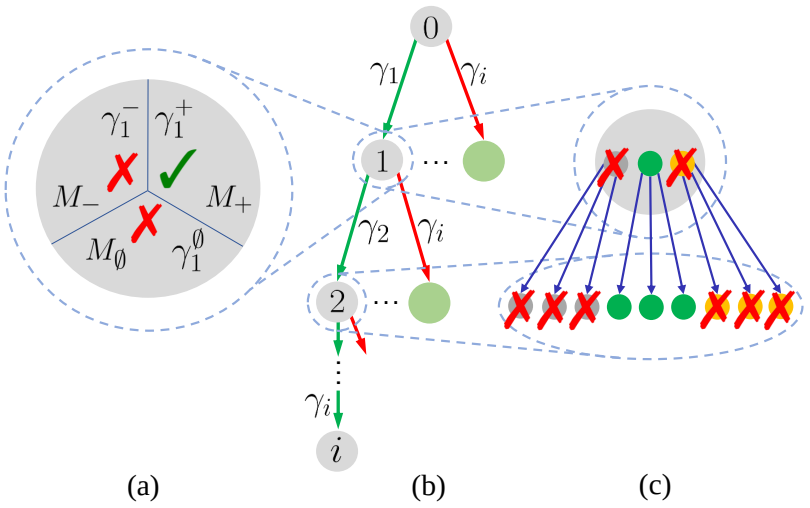


Figure 2. (b) Lattice segment explored in random order of $\gamma_i \in \Gamma$; (a) At each node, 3 abstract models are generated and 2 of them are discarded based on query responses; (c) An abstract model rejected at any level is equivalent to rejecting 3 models at the level below, 9 models two levels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

Definition 11. Let Λ be the set of all possible palm tuples which can be generated using a predicate vocabulary P^* and an action header set A_H. Let 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. The ***abstraction of a model*** M with respect to a palm tuple $\lambda \in \Lambda$, is defined by $f_\lambda : U \to U$ as $f_\lambda(M) = M \setminus \lambda$.

We extend this notation to define the abstraction of a set of models M with respect to a palm tuple λ as $X = \{f_\lambda(m) : m \in M\}$. 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 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 12. Let $\Lambda = \Gamma \times \{+, -, \emptyset\}$ be the set of all palm tuples. 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 mapping nodes to sets of abstract models, E is the set of lattice edges, and $\ell_E : E \to \Gamma$ is a labeling function mapping edges to pal tuples such that for each edge $n_i \to n_j$, $\ell_N(n_j) = \{\xi \cup \{\gamma^k\} | \, \xi \in \ell_N(n_i), \gamma = \ell_E(n_i \to n_j), k \in \{+, -, \emptyset\}\}$, 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 $\ell_N : N \to 2^{2^\Lambda}$, only three requirements of parent abstract models are stored. 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 models consistent with the agent's responses up to that point. At every step, our algorithm creates queries that help us determine the next descending edge to take from a lattice node; corresponding to the path $0, 1, 2, \ldots, i$ in Fig. 2(b). This also avoids generating and storing the complete lattice, which can be doubly exponential in number of predicates and actions.

3.4 Solving the Agent Interrogation Task

Let Θ be the set of possible answers to queries. Thus, strings $\theta^* \in \Theta^*$ denote the information received by the assessment module at any point in the query process. Query policies for the agent interrogation task are functions $\Theta^* \to Q \cup \{Stop\}$ 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.

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 2. Consider the case of a delivery agent. Assume that AAM is considering two abstract models M_1 and M_2 having only one action `load_truck(?package ?truck ?loc)` and the predicates `(at ?package ?loc)`, `(at ?truck ?loc)`, `(in ?package ?truck)`, and that the agent's model is $M^{\mathcal{A}}$ (Fig. 1). 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)`.

3.4.1 Agent Interrogation Algorithm

Algorithm 1 shows AAM's overall algorithm. It takes the agent $\mathcal{A}$, the set of instantiated predicates P^*, the set of all action headers A_H, and a set of random states

Model	Precondition		Effect
$M^{\mathcal{A}}$	`(at ?truck ?loc)`	$\wedge \rightarrow$	`(in ?package ?truck)`
	`(at ?package ?loc)`		$\wedge \neg$`(at ?package ?loc)`
M_1	`(at ?truck ?loc)`	$\rightharpoonup$	`(in ?package ?truck)`
	`(at ?package ?loc)`		
M_2	`(at ?truck ?loc)`	$\rightarrow$	`(in ?package ?truck)`
M_3	`(at ?truck ?loc)`	$\rightharpoonup$	`()`

Table 1. `load_truck(?package ?truck ?loc)` actions of the agent model $M^{\mathcal{A}}$ (unknown to $\mathcal{H}$) and three abstracted models M_1, M_2, and M_3.

S as input, and gives the set of functionally equivalent estimated models represented by *poss_models* as output. S can be generated in a preprocessing step given 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 initializes an empty set at each node to store the pruned models (line 6).

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. Sec. 3.4.1.1 describes this process in detail.

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

Algorithm 1: Agent Interrogation Algorithm (AIA)

Input : $\mathcal{A}, A_H, P^*, S$
Output : $poss\_models$

1 Initialize $poss\_models = \{\phi\}$
2 **for** γ *in some input pal ordering* Γ **do**
3 $new\_models \leftarrow poss\_models$
4 $pruned\_models = \{\}$
5 **for** *each* M' *in* new_models **do**
6 **for** *each pair* $\{i, j\}$ *in* $\{+, -, \emptyset\}$ **do**
7 $\mathcal{Q}$, M_i, $M_j \leftarrow generate\_query(M', i, j, \gamma, S)$
8 $M_{prune} \leftarrow filter\_models(\mathcal{Q}, M^{\mathcal{A}}, M_i, M_j)$
9 $pruned\_models \leftarrow pruned\_models \cup M_{prune}$
10 **if** $pruned\_models$ *is* $\emptyset$ **then**
11 $update\_pal\_ordering(\Gamma, S)$
12 continue
13 $poss\_models \leftarrow new\_models \times \{\gamma^+, \gamma^-, \gamma^\emptyset\} \setminus pruned\_models$
14 **return** $poss\_models$

consistent with that of the agent (line 11). Then it updates the estimated set of possible models represented by $poss\_models$ (line 18). This process is explained in Sec. 3.4.1.2 in detail.

If AIA is unable to prune any model at a node (line 14), it modifies the pal tuple ordering (line 15). Sec. 3.4.1.3 explains this modification in detail. 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 the accuracy of the derived models. AIA requires $O(|P^*| \times |A|)$ queries as there are $2 \times |P^*| \times |A|$ pal tuples. However, our empirical studies show that we never generate so many queries. The rest of the section describes each of the submodules used in AIA.

3.4.1.1 Query Generation

The query generation process corresponds to the *generate_ query()* module in AIA which takes a model M', the pal tuple γ, and 2 modes $i, j \in \{+, -, \emptyset\}$ as input; and returns the models $M_i = M' \cup \{\gamma^i\}$ and $M_j = M' \cup \{\gamma^j\}$, and a plan-outcome query $\mathcal{Q}$ distinguishing them, i.e., $M_i \mathrm{l}^{\mathcal{Q}} M_j$.

Plan-outcome queries have two components, an initial state s_I, and a plan π. AIA gets s_I from the input set of random states $S \in \mathcal{S}$ (line 4). Using s_I as the initial state, the idea is to find a plan, which when executed by M_i and 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 M_i and M_j.

Since the models M_i and M_j are abstract models, we need to ensure that we make accurate inferences based on their response to a query and compare it with $\mathcal{A}$'s response. Hence we cannot directly use M_i and M_j to ask questions. We illustrate this using the example below:

Example 3. Consider an empty model ϕ with two actions `unload(?package ?truck ?loc)` and `load(?package ?truck ?loc)`. Now consider that it is being concretized with the pal tuple $\gamma = \langle$`(in ?package ?truck)`, `unload(?package ?truck ?loc)`, $pre\rangle$. The two models possible in this case are M_1 and M_2 shown in Tab. 2. It can be clearly seen that the model M_2 (without p_u) is an abstraction of $M^{\mathcal{A}}$. Now consider a query $q = \langle s_I, \pi\rangle$ where $s_I = \{$`(at p1 loc1)`, `(at t1 loc1)`$\}$, and $\pi = \langle$`load(p1 t1 loc1)`, `unload(p1 t1 loc1)`$\rangle$. Now the response to this query by the agent $\mathcal{A}$ with model $M^{\mathcal{A}}$ in Tab. 2 will be $q(M^{\mathcal{A}}) = \langle 2, \{$`(at p1 loc1)`, `(at t1 loc1)`$\}\rangle$. On the same query, the response of the abstract model M_1 will

Model	Action	Precondition		Effect
$M^{\mathcal{A}}$	unload(?package ?truck ?loc)	(at ?truck ?loc) ∧ (in ?package ?truck)	→	(at ?package ?loc) ∧ ¬(in ?package ?truck)
	load(?package ?truck ?loc)	(at ?truck ?loc) ∧ (at ?package ?loc)	→	¬(at ?package ?loc) ∧ (in ?package ?truck)
M_1	unload(?package ?truck ?loc)	¬(in ?package ?truck)	→	()
	load(?package ?truck ?loc)	()	→	()
M_2	unload(?package ?truck ?loc)	(in ?package ?truck)	→	()
	load(?package ?truck ?loc)	()	→	()
M_1+$\texttt{in}_u$	unload(?package ?truck ?loc)	(($\texttt{in}_u$?package ?truck)∨ ¬(in ?package ?truck))	→	($\texttt{in}_u$?package ?truck)
	load(?package ?truck ?loc)	($\texttt{in}_u$?package ?truck)	→	($\texttt{in}_u$?package ?truck)
M_2+$\texttt{in}_u$	unload(?package ?truck ?loc)	(($\texttt{in}_u$?package ?truck)∨ (in ?package ?truck))	→	($\texttt{in}_u$?package ?truck)
	load(?package ?truck ?loc)	($\texttt{in}_u$?package ?truck)	→	($\texttt{in}_u$?package ?truck)

Table 2. load(?package ?truck ?loc) and unload(?package ?truck ?loc) actions of the agent model $M^{\mathcal{A}}$ (unknown to $\mathcal{H}$) and two abstracted models M_1 and M_2, with and without the dummy predicate $\texttt{in}_u$.

be $q(M_1) = \langle 2, \{\texttt{(at p1 loc1)}, \texttt{(at t1 loc1)}\}\rangle$, whereas M_2's response will be $q(M_2) = \langle 1, \{\texttt{(at p1 loc1)}, \texttt{(at t1 loc1)}\}\rangle$. Hence according to these responses, the M_2 will be discarded which is actually the correct model.

This inconsistency happens because the model ϕ is only partially concretized in terms of the predicate (in ?package ?truck), and this information is not captured in M_1 and M_2. To alleviate this issue, we add a predicate p_u to the models M_1 and M_2 as shown in Tab. 2. So p_u in any location (precondition or effect) helps capture the information that it is not known how a predicate p appears in that action's precondition (or effect). Without p_u, the planning problem can generate a plan as a query such that a model's response on it may be consistent with the agent even though the model is not an abstraction of $M^{\mathcal{A}}$. Now in the example above, with p_u added to the models M_1 and M_2, the response to the query for both the

Algorithm 2: Query Generation Algorithm

Input : M', i, j, γ, S
Output : $\mathcal{Q}, M_i, M_j$

$M_i, M_j \leftarrow add\_palm(M', i, j, \gamma)$
for s_I *in* S **do**
  dom, $prob \leftarrow get\_planning\_prob\ (s_I, M_i, M_j)$
  $\pi \leftarrow planner(dom, prob)$
  $\mathcal{Q} \leftarrow \langle s_I, \pi \rangle$
  if π **then** break **end if**
return $\mathcal{Q}$, $M' \cup \{\gamma^i\}$, $M' \cup \{\gamma^j\}$

models will be $q(M_1) = q(M_2) = \langle 1, \{$`(at p1 loc1)`, `(at t1 loc1)`$\}\rangle$, and hence q will no longer be a distinguishing query for these models. A possible distinguishing query in this case will be $q' = \langle s_I, \pi \rangle$ where $s_I = \{$`(in p1 t1)`, `(at t1 loc1)`$\}$, and $\pi = \langle$`unload(p1 t1 loc1)`$\rangle$.

Generating plan outcome queries by reduction to planning We reduce the problem of generating a plan-outcome query from M_i and M_j to a planning problem. We add the pal tuple $\gamma = \langle p, a, l \rangle$ in modes i and j to M' to get M'_i and M'_j, respectively. If the location $l = \mathit{eff}$, we add the palm tuple normally to M', i.e., $M'_m = M' \cup \langle p, a, l, m \rangle$, where $m \in \{i, j\}$. We modify these concretized models M'_j and M'_j further to reflect unknown effects on p as follows. Intuitively, an action makes an auxiliary predicate p_u (representing an unknown effect on p) true iff it does not have p in any mode as a precondition. To achieve this, we add the tuple $\langle p_u, a, \mathit{eff}, + \rangle$ for all actions a that don't have p in any mode as an effect, and the tuple $\langle p_u, a, \mathit{eff}, - \rangle$ for all other actions. Similarly, we further modify the model to reflect the unknown form of the preconditions in terms of p by adding p_u to the preconditions of all actions that do not have p in any mode in their precondition.

Note that p_u is added only for generating a distinguishing query and is not part of the models M_i and M_j returned by the query generation process.

We now show how to reduce plan-outcome query generation into a planning problem $\mathcal{P}(M_i'', M_j'')$ (line 5). $\mathcal{P}(M_i'', M_j'')$ uses conditional effects in its actions (in accordance with PDDL (McDermott *et al.*, 1998)). The model used to define $\mathcal{P}(M_i'', M_j'')$ has predicates from both models M_i'' and M_j'' represented as $P^{M_i''}$ and $P^{M_j''}$ respectively, in addition to a new auxiliary 0-ary predicate p_ψ. The action headers are the same as A_H. Each action's precondition is a disjunction of the preconditions of M_i'' and M_j''. This makes an action applicable in a state s if either M_i'' or M_j'' can execute it in s. The effect of each action has two conditional effects; the first applies the effects of both M_i'' and M_j'''s action if preconditions of both M_i'' and M_j'' are true, whereas the second makes the auxiliary predicate p_ψ true if precondition of only one of M_i'' and M_j'' is true. Note that p_ψ is initially false. Adding p_ψ helps identify the goal state $s_\mathcal{P}$.

Reason for adding $\mathbf{p}_\psi$ Formally, we express the planning problem $P_{PO}(M_i'', M_j'')$ as a 3-tuple $\langle M^{PO}, s_I, G\rangle$, where M^{PO} is a model with predicates $P^{PO} = \mathcal{P}^{M_i''} \cup \mathcal{P}^{M_j''} \cup \{p_\psi\}$, and actions A^{PO} where for each action $a \in A^{PO}$, $pre(a) = pre(a^{M_i''}) \vee pre(a^{M_j''})$, and

$$\begin{aligned} \mathit{eff}(a) = &(\mathit{when}\,(pre(a^{M_i''}) \wedge pre(a^{M_j''}))(\mathit{eff}(a^{M_i''}) \wedge \mathit{eff}(a^{M_j''}))) \\ &(\mathit{when}\,((pre(a^{M_i''}) \wedge \neg pre(a^{M_j''})) \vee (\neg pre(a^{M_i''}) \wedge pre(a^{M_j''})))\,(p_\psi)), \end{aligned}$$

The initial state $s_I = s_I^{M_i''} \wedge s_I^{M_j''}$, where $s_I^{M_i''}$ and $s_I^{M_j''}$ are copies of all predicates in s_I, and G is the goal formula expressed as $\exists p\ (p^{M_i''} \wedge \neg p^{M_j''}) \vee (\neg p^{M_i''} \wedge p^{M_j''}) \vee p_\psi$.

With this formulation, the goal is reached when an action in M_i'' and 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. 1. From the state `(at t1 l1)` $\wedge$ `¬(at p1 l1)`, M_2 can execute `load_truck(p1 t1 l1)` but M_1 cannot. Similarly, in state `(at t1 l1)` $\wedge$ `(at p1 l1)`, executing `load_truck(p1 t1 l1)` will cause

$M^{\mathcal{A}}$ and 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.

We will formally prove in Sec. 3.5 that (i) this planning problem will always generate a solution query when using two models M_i'' and M_j'' that differ only in one palm tuple (Thm. 1), and (ii) any inferences about the consistency of models wrt. the agent model $M^{\mathcal{A}}$ based on responses to these queries are correct (Thm. 2).

3.4.1.2 Filtering Possible Models

This section describes the *filter_models()* module in Algorithm 1 which takes as input $M^{\mathcal{A}}$, M_i, M_j, and the query $\mathcal{Q}$ (Sec. 3.4.1.1), and returns the subset M_{prune} which is not consistent with $M^{\mathcal{A}}$.

First, AAM *poses the query* $\mathcal{Q}$ to M_i, M_j, and the agent $\mathcal{A}$. Based on the responses of all three, it determines if the two models are prunable, i.e., $M_i \Diamond M_j$. As mentioned in Def. 10, checking for prunability involves checking if the response to the query $\mathcal{Q}$ by one of the models M_i or M_j is consistent with that of the agent or not.

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.

3.4.1.3 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 M_2 and M_3 being considered by AAM (Fig. 1). 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 M_2 and 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 $a_F = \pi[\ell{+}1]$ and the final state s_F. Since the query plan π is generated using M_i and M_j (which differ only in the newly added palm tuple), they both would reach the same state $\overline{s}_F$ after executing first ℓ steps of π. Thus, we search S for a state $s \supset \overline{s}_F$ where $\mathcal{A}$ can execute a_F. Similar to Stern and Juba (2017), we infer that any predicate which is false in s will not appear in a_F's precondition in the positive mode. Next, we iterate through the set of predicates $p' \subseteq s \setminus \overline{s}_F$ and add them to $\overline{s}_F$ to check if $\mathcal{A}$ can still execute a_F. Thus, on adding a predicate $p \in p'$ to the state $\overline{s}_F$, if $\mathcal{A}$ cannot execute a_F, we add p in negative mode in a_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.

3.5 Formal Analysis of the AIA

In this section, we present a comprehensive theoretical analysis of AIA (Alg. 1). We show that AIA will always terminate and that the models returned by AIA are consistent with the agent's model and any model that is discarded by AIA is not consistent with that of the agent model.

To show this we prove the main theorem of this chapter (Thm. 3) which shows that the algorithm will terminate and the models returned by the algorithm are consistent with the agent's model. To prove it, we will prove that the approach prunes away models that are not consistent with the agent's model, and that the models returned by the algorithm are consistent with the agent's model. We will also show that the algorithm will terminate.

We will first show that a model no pal tuples is consistent with the agent model according to Def. 9. When starting with an empty model, i.e., in which the correct mode is not known for any pal tuple, the only way we can concretize the model is when a predicate is added to the precondition. This is because we cannot reason about the effects of an action on a predicate if we are not sure about the cases under which that action is applicable in terms of that predicate. Proving this is important because we will use this property as an important step in our proof by induction later on.

Lemma 1. Consider an empty model M having 0 palm tuples, and that Alg. 1 concretizes M by adding a new pal tuple $\gamma = \langle p, a, l \rangle$ to M to generate models M'_i and M'_j, for $i, j \in \{+, -, \emptyset\}$. The planning problem $\mathcal{P}(M'_i, M'_j)$ has a solution if and only if $l = pre$.

Proof. We prove this in two parts. First, we show that if $l = pre$ then $\mathcal{P}(M'_i, M'_j)$ has

a solution. Then, we show that if $l \neq pre$ ($l = eff$) then $\mathcal{P}(M'_i, M'_j)$ does not have any solution.

We prove the first part by construction. Recall that we add the pal tuple $\gamma = \langle p, a, l = pre \rangle$ in modes $i, j \in \{\emptyset, -, +\}$ to M to get M_i and M_j. Here $M = \{\}$, and pal tuple being added is $\gamma = \langle p, a, l = pre \rangle$ (line 9 of Alg. 1). Hence in the model for $\mathcal{P}(M'_i, M'_j)$, (i) precondition of all actions except a will be p_u; (ii) precondition of a will be $p \vee p_u$ in M_+, $\neg p \vee p_u$ in M_-, and $M_\emptyset$ will have empty precondition; and (iii) effects of all actions will be p_u. Now if $\{i, j\} \in \{\{+, \emptyset\}, \{+, -\}\}$ then $\mathcal{P}(M'_i, M'_j)$ has a solution if the initial state does not have p. Similarly if $\{i, j\} \in \{\{-, \emptyset\}, \{+, -\}\}$ then $\mathcal{P}(M'_i, M'_j)$ has a solution if the initial state has p. In both these cases, the solution plan π will be $\langle a \rangle$. Hence if $l = pre$ then $\mathcal{P}(M'_i, M'_j)$ has a solution.

Now we show that if $l \neq pre$ then $\mathcal{P}(M'_i, M'_j)$ does not have any solution. Here $M = \{\}$, and pal tuple being added is $\gamma = \langle p, a, l = eff \rangle$. Hence in the model for $\mathcal{P}(M'_i, M'_j)$, (i) effect of all actions except a will be p_u; (ii) effect of a will be p in M_+, $\neg p$ in M_-, and $M_\emptyset$ will have empty effect; and (iii) precondition of all actions will be p_u. Since p_u is not present in the initial state, no action is executable. So there is no plan possible; hence there is no solution for $\mathcal{P}(M'_i, M'_j)$ in this case. □

We will now formalize and show that the solution to the planning problem $P_{PO}(M_i, M_j)$ we just created is possible if the two models M_i and M_j have a distinguishing query. To formulate this, we will first define some additional notation. Consider the example 3 shown earlier. The initial state s_I used in P_{PO} will be $\texttt{(at}_i\ \texttt{t1 l1)} \wedge \texttt{(at}_j\ \texttt{t1 l1)}$, where $\texttt{(at}_x\ \texttt{t1 l1)}$ corresponds to `(at t1 l1)` predicate in the model M_x. We also represent the projection of a state s in planning problem P^{ij}_{PO} according to models M_i and M_j as $[s]_{M_i}$ and $[s]_{M_j}$, respectively. Here projection of a state s in planning problem P^{ij}_{PO} according to models M_i is the set of predicates with the

subscript i, without their subscripts. E.g., consider if $s = \{$`(at`$_i$ `t1 l1)`, `(at`$_j$ `p1 l1)`, p_ψ, `(at`$_u$ `t1 l1)`$\}$, then $[s]_{M_i} = \{$`(at t1 l1)`$\}$, and $[s]_{M_j} = \{$`(at p1 l1)`$\}$.

We now formalize an important property of the planning problem $P_{PO}(M_i, M_j)$'s solution. For brevity, we will represent $P_{PO}(M_i, M_j)$ as P^{ij} and the set of actions in P^{ij} as A^{ij}. We will show that the result of executing any action according to P^{ij} will be such that if $a^{ij}(s) = s'$ for any $a^{ij} \in A^{ij}$, then $a^x([s]_{M_x}) = [s']_{M_x}$, for all $x \in \{i, j\}$, where $header(a^{ij}) = header(a^x)$. This property is important to ensure that the responses to the queries by the models M_i and M_j are consistent with what was expected from the query generation process. Note that this *will not hold* for the final action in the plan if the action was executable according to only one of the models. This is because the final action will make the predicate p_ψ true in this case, whereas the model according to which the action was not executable will fail to execute the action, and the other model will make the correct effects true or false. We formalize this property as follows:

Lemma 2. Consider a model M' that is an abstraction of the agent model $M^{\mathcal{A}}$, and M_i and M_j are two models concretized from M' such that they differ in mode of a single pal tuple $\langle p, a, l\rangle$. Consider $P^{ij}_{PO}(M_i, M_j)$ be a planning problem used to distinguish M_i and M_j with an initial state s_I, such that its solution is π, and $|\pi| = k$. If the result of executing any action $a^{ij} \in \pi$ according to P^{ij} will be such that if $a^{ij}_b(s) = s'$ for any $a^{ij}_b \in A^{ij}$ and $b \in [1, k-2]$, then $a^x([s]_{M_x}) = [s']_{M_x}$, for all $x \in \{i, j\}$, where $header(a^{ij}) = header(a^x)$.

Proof. Recall that the goal to the planning problem $P^{ij}_{PO}(M_i, M_j)$ contains p_ψ in disjunction. So we will never execute an action in a state where p_ψ is true. Also recall that in the problem P^{ij}, $pre(a^{ij}) = pre(a^{M''_i}) \vee pre(a^{M''_j})$, where M''_x, $x \in \{i, j\}$ were the intermediate models used to create the planning problem P^{ij}. Now, $pre(a^{M''_x})$

cannot be false here, as $b \in [1, k-2]$, so we are considering all the actions other than the last action. So if $pre(a^{M''_x})$ was false for any a_b such that $b < k-1$, then that would've been the last action, which is not the case.

Now we will consider two cases: first, where we execute an action in a state without p_u, and second, in a state containing p_u. We analyze them one by one.

Case 1: Executing an action a in a state s, that does not have p_u predicates. This is the trivial case where the action a is executed when the precondition is satisfied according to at least one of the two models. Now when $pre(a^{M''_x})$ is true (the precondition corresponding to model M_x is satisfied), then the same precondition $pre(a)$ (projected version) is true in the actual model M_x too. This is because by construction the precondition can only have p_u predicates in addition to the normal predicates. Since projection removes p_u, if we project $pre(a^{M''_x})$, we'll get $pre(a)$ according to M_x. Hence Similar argument holds for the effect, hence the projection of effect $\mathit{eff}(a^{M''_x})$ will be same as $\mathit{eff}(a)$ according to M_x. Additionally, This implies that if $a_b(s) = s'$ for the case where p_u is false in s, the projection of states s and s' will be such that $a^x([s]_{M_x}) = [s']_{M_x}$.

Case 2: Executing an action a in a state s, that has p_u predicates. This condition means that the action is being executed in the state s where it is not known if the predicate p is true or false. Now, projecting s according to M_i (or M_j) would result in $[s]_{M_i}$ (or $[s]_{M_j}$) where p is not true. Similarly, p would also be absent from the precondition of a^i (or a^j). Hence if a was executable in s, a^i (or a^j) would be executable in $[s]_{M_i}$ (or $[s]_{M_j}$).

In both these cases, the effects of the action will change the states in an exact manner except for the p_u predicates. And since the projection of states according to a model anyway removes the p_u predicates, the resulting state s' according to the

planning problem's action and $[s']_{M_i}$ ($[s']_{M_j}$) according to the model M_i (or M_j) will be the same. □

We will next show that the solution plan to $\mathcal{P}(M_i', M_j')$ always ends up with the action that is part of the pal tuple being concretized at that time. This will help us in limiting our analysis to, at most, the last two actions in the plan.

Lemma 3. Let $M_i, M_j \in \{M_+, M_-, M_\emptyset\}$ be the models generated by adding pal tuple $\gamma = \langle p, a, l\rangle$ to M'. Suppose starting in a state s_I, π is a solution to $\mathcal{P}(M_i, M_j)$. The last action in the plan π will be a.

Proof. We prove this by contradiction. Consider that the last action of the solution plan π is $a_F \neq a$, where a is the action in pal tuple $\gamma = \langle p, a, l\rangle$. Now, by construction, $\mathcal{P}(M_i, M_j)$ has a solution if after executing a_F (i) p_ψ is true, or (ii) $\exists p\ (p^{M_i''} \wedge \neg p^{M_j''}) \vee (\neg p^{M_i''} \wedge p^{M_j''})$ is true. Here M_x'' are the intermediate models used in creating the planning problem $\mathcal{P}(M_i, M_j)$. We will now consider both the cases.

Case 1: p_ψ is true after executing a_F. As mentioned earlier, p_ψ becomes true when $((pre(a_F^{M_i''}) \wedge \neg pre(a_F^{M_j''})) \vee (\neg pre(a_F^{M_i''}) \wedge pre(a_F^{M_j''}))$. This means that when a_F is executed in a state s such that either $[s']_{M_i} \not\models pre(a_F^{M_i}) \wedge [s']_{M_j} \models pre(a_F^{M_j})$ or $[s']_{M_i} \models pre(a_F^{M_i}) \wedge [s']_{M_j} \not\models pre(a_F^{M_j})$. This means that $pre(a_F^{M_i}) \neq pre(a_F^{M_j})$. This is not possible as M_i and M_j are constructed from same model M' by making changes to action a. So $pre(a_F^{M_i})$ must be equal to $pre(a_F^{M_j})$. This means that our assumption that the last action of the plan must be $a_F \neq a$, must be false.

Case 2: $\exists p\ (p^{M_i''} \wedge \neg p^{M_j''}) \vee (\neg p^{M_i''} \wedge p^{M_j''})$ is true. This means that when a_F is executed in a state s such that $[s']_{M_i} \models pre(a_F^{M_i}) \wedge [s']_{M_j} \models pre(a_F^{M_j})$, their effects are not the same, i.e., $\mathit{eff}(a_F^{M_i}) \neq \mathit{eff}(a_F^{M_j})$. This is not possible as M_i and M_j are constructed from same model M' by making changes to action a. So $\mathit{eff}(a_F^{M_i})$ must be equal to

$\mathit{eff}(a_F^{M_j})$. This means that our assumption that the last action of the plan must be $a_F \neq a$, must be false.

Since for both the cases, $a_F \neq a$ was false, hence $a_F = a$, i.e., the last action of the plan π will be a. □

We will next show that if the agent can execute the query successfully, then each intermediate state that each of the models generate on executing the plan in the query will be an abstraction of the states that the agent will generate as part of executing the same query. Additionally, all the intermediate states generated by both models will be identical.

Suppose $\mathcal{Q} = \langle s_I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models M_i, M_j, i.e., $M_i \mathtt{l}^{\mathcal{Q}} M_j$. Let $\mathcal{Q}_{1\ldots z}$ represent the query with same initial state $s_I^{\mathcal{Q}}$, and plan $\pi_{1\ldots z}^{\mathcal{Q}}$, i.e., first z ($z < len(\pi^{\mathbb{Q}})$) actions from plan $\pi^{\mathcal{Q}}$.

Lemma 4. Let $M_i, M_j \in \{M_+, M_-, M_\emptyset\}$ be the models generated by adding pal tuple γ to M' which is an abstraction of $M^{\mathcal{A}}$. Suppose $\mathcal{Q} = \langle s_I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models M_i, M_j, i.e., $M_i \mathtt{l}^{\mathcal{Q}} M_j$. Let the agent's response to the query $\mathcal{Q}(M^{\mathcal{A}})$ be $\langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$, where $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$. Let the response of models M_i, M_j, and $M^{\mathcal{A}}$ to the query $\mathcal{Q}_{1\ldots z}$ be: $\mathcal{Q}_{1\ldots z}(M_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}_{1\ldots z}(M_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}_{1\ldots z}(M^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_h^{\mathcal{A}} \rangle \rangle$. If $z \leq len(\pi^{\mathcal{Q}}) - 1$, then $m = n$, $\{p_1^i, \ldots, p_m^i\} = \{p_1^j, \ldots, p_n^j\}$ and $\{p_1^i, \ldots, p_m^i\} \setminus \{p_u\} \subseteq \{p_1^{\mathcal{A}}, \ldots, p_h^{\mathcal{A}}\}$.

Proof. We first show that if the agent is able to execute the distinguishing query q successfully (i.e., $\ell^{\mathcal{A}} = len(\pi^q)$), then all the intermediate states generated by both the models while on executing the plan $\pi^{\mathcal{Q}}$ starting in the initial state $s^{\mathcal{Q}}$ are the same, and then show that the projection of these states are an abstraction of the corresponding intermediate state generated by the agent on executing the same query.

Note that since q is a distinguishing query, $|\ell^i - \ell^j| \in \{0, 1\}$, also $z \leq \ell^i$ and $z \leq \ell^j$.

We now prove the first part by contradiction. The query $\mathcal{Q}$ used to distinguish between M_i and M_j is generated using the planning problem P^{ij}. Suppose there exists $z' < len(\pi^{\mathcal{Q}})$ for which the intermediate states generated after executing the plan $q_{1\ldots z'}$ are not same according to M_i and M_j. Recall that P^{ij} has a solution if M_i and M_j have different preconditions or different effects for the same action (by definition of goal of P^{ij}). According to this, the plan $\pi^q_{1\ldots z'}$ should also be a solution to P^{ij}. But this is not possible as any subsequence of $\pi^{\mathcal{Q}}$ cannot be a solution otherwise $\pi^{\mathcal{Q}}$ would never have been returned as the solution to P^{ij}. Hence all the intermediate states generated by both the models while on executing the plan $\pi^{\mathcal{Q}}$ starting in the initial state $s^{\mathcal{Q}}$ are the same.

Now we prove the second part that the intermediate states generated by both the models on executing the plan $\pi^{\mathcal{Q}}$ starting in the initial state $s^{\mathcal{Q}}$ are a subset of the corresponding projection of the intermediate state generated by the agent on executing the same query. Since M' is an abstraction of $M^{\mathcal{A}}$, all the palm tuples already present in it are also present in $M^{\mathcal{A}}$. The only new palm tuples in M_i and M_j are the ones involving pal tuple γ or involving predicate p_u. Now, as shown above, only the last action in the plan $\pi^{\mathcal{Q}}$ will be a (corresponding to γ), and all the actions prior to that will not be a. Now, all the actions other than a, have the same preconditions and effects as they were in M' (which is an abstraction of $M^{\mathcal{A}}$), and since those actions in M' were consistent with that of $M^{\mathcal{A}}$, the effect of executing them will also be consistent with the agent $\mathcal{A}$. Thus, if s^q is the starting state, $z \leq len(\pi^{\mathcal{Q}}) - 1$, and $\pi^{\mathcal{Q}}_{1\ldots z}(s)$ $= s'_{ij}$ according to P^{ij}, and $\pi^{\mathcal{Q}}_{1\ldots z}(s) = s'_{\mathcal{A}}$ according to $M^{\mathcal{A}}$, then $[s'_{ij}]_{M_i} \subseteq s'_{\mathcal{A}}$ and $[s'_{ij}]_{M_j} \subseteq s'_{\mathcal{A}}$. Note that in this notation, $[s'_{ij}]_{M_i} = \{p^i_1, \ldots, p^i_m\}$, $[s'_{ij}]_{M_j} = \{p^j_1, \ldots, p^j_n\}$, and $s'_{\mathcal{A}} = \{p^{\mathcal{A}}_1, \ldots, p^{\mathcal{A}}_h\}$. □

Now we will see how these lemmas we proved so far combines to show that the planning problem created as part of Alg. 1 is guaranteed to generate a distinguishing query, if exists.

Theorem 1. Consider a model M' that is an abstraction of the agent model $M^{\mathcal{A}}$, and M_i and M_j are two models concretized from M' such that they differ in mode of a single pal tuple. Given such pair of models M_i and M_j and an initial state s_I, the planning problem P^{ij}_{PO} has a solution iff M_i and M_j have a distinguishing plan-outcome query $\mathcal{Q}_{PO}$.

Proof. We first show that if the planning problem P_{PO} has a solution, then M_i and M_j have a distinguishing plan-outcome query $\mathcal{Q}_{PO}$ comprising of an initial state s_I and plan π^{PO}. By construction, the initial state s_I in $\mathcal{Q}_{PO}$ and P_{PO} is same. Suppose $P(M_i, M_j)$ has a solution plan π, which is a sequence of actions $a_1, a_2, \ldots, a_k$ where $a_x \in A_{P^{ij}}$, such that on executing this plan, the goal condition is met. Consider the trace of this execution be $\langle s_I, a_1, s_1, a_2, s_2, \ldots, s_{k-1}, a_k, s_G \rangle$. Recall that the goal is either the predicate p_ψ is true, or the final state according to the two models on executing the plan does not match. We will consider these cases individually.

Case 1: The predicate p_ψ is true on executing the plan, i.e., $s_G \models p_\psi$. By the construction of P^{ij}, it means that a_k made p_ψ true. It implies that only one of M_i's or M_j's preconditions were met in s_{k-1}. Consider that model whose preconditions were not met to be M_i. Now using lemma 4, we know that $[s_{k-1}]_{M_i} = [s_{k-1}]_{M_j}$, hence we will refer it as $[s_{k-1}]_{M_x}$ for brevity. Using Lemma 2, this also means that in the original models M_i and M_j, when executing the corresponding action a^i_k (or a^j_k) in the state $[s_{k-1}]_{M_x}$, $[s_{k-1}]_{M_x} \not\models pre(a^i_k)$, and $[s_{k-1}]_{M_x} \models pre(a^j_k)$. Hence, if P^{ij} has a solution such that $s_G \models p_\psi$, then $M_i \wr M_j$.

Case 2: One of the predicates is true according to one of the model, and false according to another in the goal state, i.e., $s_G \models \exists p^i, p^j \in P^{ij}(p^i \wedge \neg p^j) \vee (\neg p^i \wedge p^j)$. By the construction of P^{ij}, it means that a_k made this condition true. It implies that both of M_i's or M_j's preconditions were met in s_{k-1}. Now using lemma 4, we know that $[s_{k-1}]_{M_i} = [s_{k-1}]_{M_j}$, hence we will refer it as $[s_{k-1}]_{M_x}$ for brevity. Using Lemma 2, this also means that in the original models M_i and M_j, when executing the corresponding action a_k^i (or a_k^j) in the state $[s_{k-1}]_{M_x}$, $[s_{k-1}]_{M_x} \models pre(a_k^i) \wedge pre(a_k^j)$. Since, in the predicate after executing a_k differs, $[s_k]_{M_i} \neq [s_k]_{M_j}$. Hence, if P^{ij} has a solution such that $s_G \models \exists p^i, p^j \in P^{ij}(p^i \wedge \neg p^j) \vee (\neg p^i \wedge p^j)$, then $M_i \wr M_j$.

□

Equivalent Models It is possible for AIA to encounter a pair of models M_i and M_j that are not prunable. In such cases, the models M_i and M_j are functionally equivalent and cannot be discarded. Hence, both the models end up in the set *poss_models* in line 18 of AIA.

We will next prove that a model is not an abstraction of the agent model if it is not consistent with that of the agent. But to prove that, we will use a couple of smaller results. We first start by showing that we can only prune an abstract model based on a query's responses if the agent can execute all actions in the query plan successfully.

We will next show that if an agent's response is not consistent with such a model, then that model (without the p_us) is not an abstraction of the agent. But to prove that we first show that if the agent cannot successfully execute all the actions in the query plan, then we cannot decide if an abstracted model is consistent with the agent.

Lemma 5. Let $M_i \in \{M_+, M_-, M_\emptyset\}$ be the model generated by adding the pal tuple γ to M' which is an abstraction of the true agent model $M^{\mathcal{A}}$. Suppose $\mathcal{Q}$ is

a distinguishing query for two distinct models M_i and $M_j \in \{M_+, M_-, M_\emptyset\} \setminus M_i$. If $M^{\mathcal{A}}$ cannot execute all the actions in the query successfully, then we cannot decide consistency of the M_i (or M_j) response with that of the agent.

Proof. Suppose $\mathcal{Q} = \langle s_I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models M_i, M_j, i.e. $M_i \wr^{\mathcal{Q}} M_j$, and the response of models M_i, M_j, and $M^{\mathcal{A}}$ to the query $\mathcal{Q}$ are $\mathcal{Q}(M_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(M_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(M^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. We show that when $\ell^{\mathcal{A}} \neq len(\pi^{\mathcal{Q}})$, i.e., $M^{\mathcal{A}}$ cannot execute all the actions in the query successfully, then we can make incorrect inferences about the consistency of the M_i (or M_j) response with that of the agent.

We prove this by counterexample. When $\ell^{\mathcal{A}} \neq len(\pi^{\mathcal{Q}})$, consider the models M_1 with p_u, and M_2 with p_u in Tab. 2. Consider the initial state to be $\{$`(in package1 truck1)`$\}$, and the plan be $\langle$`unload(package1 truck1 location1)`$\rangle$. Now the responses of $M^{\mathcal{A}}$ and M_1 to this query will be $\langle 0, \{$`(in package1 truck1)`$\}\rangle$, whereas that of M_2 will be $\langle 1, \{p_u\}\rangle$. This happens because the first action in the plan failed for $M^{\mathcal{A}}$ because of the precondition `(at truck1 location1)` that is not satisfied in the initial state. On the other hand, the first action for M_1 failed because of the precondition $\neg$`(in package1 truck1)`. This happens because an action may execute successfully in an abstract model (M_2 here), but fail in the model which is an accurate concretization of it ($M^{\mathcal{A}}$ here) because of some predicate (`(at truck1 location1)` here) that the abstract models haven't added to their model. Hence M_i (or M_j) cannot be pruned if $\ell^{\mathcal{A}} \neq len(\pi^{\mathcal{Q}})$. □

Lemma 6. Let $M_i \in \{M_+, M_-, M_\emptyset\}$ be the model generated by adding the pal tuple γ to M' which is an abstraction of the true agent model $M^{\mathcal{A}}$. Suppose $\mathcal{Q}$ is a distinguishing query for two distinct models M_i and $M_j \in \{M_+, M_-, M_\emptyset\} \setminus M_i$. If

M_i's (or M_j's) response is not consistent with that of the agent, then it is not an abstraction of $M^{\mathcal{A}}$.

Proof. Suppose $\mathcal{Q} = \langle s_I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models M_i, M_j, i.e. $M_i \wr^{\mathcal{Q}} M_j$, and the response of models M_i, M_j, and $M^{\mathcal{A}}$ to the query $\mathcal{Q}$ are $\mathcal{Q}(M_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(M_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(M^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. Now the model M_i's response to $\mathcal{Q}$ is said to be *consistent* with that of $M^{\mathcal{A}}$ when $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, $len(\pi^{\mathcal{Q}}) = \ell^i$ and $s^i \subseteq s^{\mathcal{A}}$, where $s^i = \{p_1^i, \ldots, p_m^i\} \backslash p_u$ and $s^{\mathcal{A}} = \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$. We prove this in multiple parts. We have already shown in Lemma 4 that $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$ is a necessary condition for consistency. We will now show that if either $len(\pi^{\mathcal{Q}}) = \ell^i$ or $s^i \subseteq s^{\mathcal{A}}$ are not true, then M_i is not an abstraction of $M^{\mathcal{A}}$.

We first show that if $len(\pi^{\mathcal{Q}}) \neq \ell^i$, then M_i is not an abstraction of $M^{\mathcal{A}}$. Since $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})taa$, the agent can execute each of the actions in the plan. Now if M_i is also able to execute an action whereas M_j can (since q is a distinguishing query), then using Lemma 2 it can only be the last action. Now the set of preconditions for an abstracted model will never be larger than its corresponding concretized model, hence if an action is executable in the concretized model, it should also be executable in the abstracted model. Now since the other actions in the plan are correct as M' is an abstraction of $M^{\mathcal{A}}$, hence if $len(\pi^{\mathcal{Q}}) \neq \ell^i$, then M_i is not an abstraction of $M^{\mathcal{A}}$.

We now show that if $s^i \not\subseteq s^{\mathcal{A}}$, then M_i is not an abstraction of $M^{\mathcal{A}}$. Since $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, the agent can execute each of the actions in the plan. Now if the states that M_i and M_j do not end up in the same state, then using Lemma 2 it can only be after the last action. Now the set of effects for an abstracted model will never be larger than its corresponding concretized model, hence if an action is executable in the concretized model and ends up in a state, then the abstracted model should also

reach a state where only the subset of its effects are true. Hence if $s^i \not\subseteq s^{\mathcal{A}}$, then M_i is not an abstraction of $M^{\mathcal{A}}$. □

Theorem 2. Let $M_i \in \{M_+, M_-, M_\emptyset\}$ be the model generated by adding the pal tuple γ to M' which is an abstraction of the true agent model $M^{\mathcal{A}}$. Suppose $\mathcal{Q}$ is a distinguishing query for two distinct models M_i and $M_j \in \{M_+, M_-, M_\emptyset\} \setminus M_i$. If M_i (or M_j) is pruned out by Alg. 1, then it is not an abstraction of $M^{\mathcal{A}}$.

Proof. We prove this by mathematical induction. Suppose $\mathcal{Q} = \langle s_I^{\mathcal{Q}}, \pi^{\mathcal{Q}} \rangle$ is a distinguishing query for two distinct models M_i, M_j, i.e. $M_i \wr^{\mathcal{Q}} M_j$, and the response of models M_i, M_j, and $M^{\mathcal{A}}$ to the query $\mathcal{Q}$ are $\mathcal{Q}(M_i) = \langle \ell^i, \langle p_1^i, \ldots, p_m^i \rangle \rangle$, $\mathcal{Q}(M_j) = \langle \ell^j, \langle p_1^j, \ldots, p_n^j \rangle \rangle$, and $\mathcal{Q}(M^{\mathcal{A}}) = \langle \ell^{\mathcal{A}}, \langle p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}} \rangle \rangle$. Now when $\ell^{\mathcal{A}} \neq len(\pi^{\mathcal{Q}})$, none of M_i or M_j can be discarded as shown in Lemma 4.

When $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, and M_i (or M_j) is pruned then it means that either $len(\pi^{\mathcal{Q}}) \neq \ell^i$ or $\{p_1^i, \ldots, p_m^i\} \not\subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$. So we will now prove that if $len(\pi^{\mathcal{Q}}) \neq \ell^i$ or $\{p_1^i, \ldots, p_m^i\} \setminus p_u \not\subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$ then M_i is not an abstraction of $M^{\mathcal{A}}$.

Let P(n) be the proposition that for every model with n pal tuples, which is consistent with $M^{\mathcal{A}}$, refining it with a pal tuple with the correct mode according to Def. 3 will prune out the models that are not an abstraction of $M^{\mathcal{A}}$.

Base Case: The proof for P(0) being true is by case analysis. Assume the model $M' = \{\}$, which is consistent with $M^{\mathcal{A}}$, is concretized with pal tuple $\gamma = \langle p, a, l \rangle$. There are only two cases possible as the location can only be a precondition or an effect.

Case 1: Consider $l = pre$. This case splits into 2 subcases, based on if the predicate will be true in the initial state or not. Note that the plan will have only one action as the models are completely empty except the only action that is being concretized.

Case 1.1: If $p \in s_I^{\mathcal{Q}}$, $\pi^{\mathcal{Q}} = \langle a \rangle$, and $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}}) = 1$, then $\langle p, a, pre, - \rangle \notin M^{\mathcal{A}}$. Also $M_j \mathfrak{l}^{\mathcal{Q}} M_-$, where $j \in \{+, \emptyset\}$, as $\ell^j = 1$, and $\ell^- = 0$. Hence P(0) is true.

Case 1.2: If $\neg p \in s_I^{\mathcal{Q}}$, $\pi^{\mathcal{Q}} = \langle a \rangle$, and $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}}) = 1$, then $\langle p, a, pre, + \rangle \notin M^{\mathcal{A}}$. Also $M_j \mathfrak{l}^{\mathcal{Q}} M_+$, where $j \in \{-, \emptyset\}$, as $\ell^j = 1$, and $\ell^+ = 0$. Hence P(0) is true.

Case 2: Consider $l = eff$. If $M = \{\}$ and $l = eff$, $\forall i, j \in \{+, -, \emptyset\}$, $\nexists \mathcal{Q}\ M_i \mathfrak{l}^{\mathcal{Q}} M_j$ as shown in Lemma 1. Hence P(0) is true.

Inductive Step: Assume that P(n) is true for some $n \geq 0$; that is we have a model M', with n palm tuples, which is an abstraction of $M^{\mathcal{A}}$, and refining it with a pal tuple $\gamma = \langle p, a, l \rangle$ will generate models with $n + 1$ tuples. From Lemma 4, we know that before executing the last action, the state reached by both the abstracted models ($\overline{s}_{F-1}$) will be a subset of the state reached by $M^{\mathcal{A}}$ (s_{F-1}). There are two cases:

Case 1: Consider $l = pre$. Since $l = pre$, $p_u \notin \overline{s}_{F-1}$. This case splits into 2 subcases:

Case 1.1: If $p \in \overline{s}_{F-1}$, and $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, then $\langle p, a, pre, - \rangle \notin M^{\mathcal{A}}$. Also $M_j \mathfrak{l}^{\mathcal{Q}} M_-$, where $j \in \{+, \emptyset\}$, as $\ell^j = len(\pi^{\mathcal{Q}})$, and $\ell^- = len(\pi^{\mathcal{Q}}) - 1$. Thus, M_- is not an abstraction of $M^{\mathcal{A}}$. Hence P(n) is true.

Case 1.2: If $\neg p \in \overline{s}_{F-1}$, and $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$, then $\langle p, a, pre, + \rangle \notin M^{\mathcal{A}}$. Also $M_j \mathfrak{l}^{\mathcal{Q}} M_+$, where $j \in \{-, \emptyset\}$, as $\ell^j = len(\pi^{\mathcal{Q}})$, $\ell^+ = len(\pi^{\mathcal{Q}}) - 1$. Thus, M_+ is not an abstraction of $M^{\mathcal{A}}$. Hence P(n) is true.

Case 2: Consider $l = \mathit{eff}$. Since $l = \mathit{eff}$, p_u may or may not be in $\overline{s}_{F-1}$. In either case, the full plan is executed in M_i, M_j and $M^{\mathcal{A}}$. Hence we can compare the states reached after executing the complete plan. Let $\overline{s}_F^{M_i} = \{p_1^i, \ldots, p_m^i\}$, $\overline{s}_F^{M_j} = \{p_1^i, \ldots, p_n^i\}$, and $s_F = \{p_1^i, \ldots, p_k^i\}$ be the final states reached upon executing $\pi^{\mathcal{Q}}$ in M_i, M_j and $M^{\mathcal{A}}$ respectively and $\overline{s}_{F-1}$ is the state reached in M_i and M_j before executing action a. This case splits into 2 subcases:

Case 2.1: If $p \in \overline{s}_{F-1}$. If $p \in s_F$, $\langle p, a, \mathit{eff}, - \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_-} \not\subseteq s_F$. Similarly if $\neg p \in s_F$, $\langle p, a, \mathit{eff}, + \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_+} \not\subseteq s_F$, and $\langle p, a, \mathit{eff}, \emptyset \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_\emptyset} \not\subseteq s_F$. Hence P(n) is true.

Case 2.2: If $\neg p \in \overline{s}_{F-1}$. If $\neg p \in s_F$, $\langle p, a, \mathit{eff}, + \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_+} \not\subseteq s_F$. Similarly if $p \in s_F$, $\langle p, a, \mathit{eff}, - \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_-} \not\subseteq s_F$, and $\langle p, a, \mathit{eff}, \emptyset \rangle \notin M^{\mathcal{A}}$ and $\overline{s}_F^{M_\emptyset} \not\subseteq s_F$. Hence P(n) is true.

This proves that if we add a pal tuple to a model that is an abstraction of $M^{\mathcal{A}}$, then we prune only inconsistent models M_i whenever $len(\pi^{\mathcal{Q}}) \neq \ell^i$ or $\{p_1^i, \ldots, p_m^i\} \not\subseteq \{p_1^{\mathcal{A}}, \ldots, p_k^{\mathcal{A}}\}$ when $\ell^{\mathcal{A}} = len(\pi^{\mathcal{Q}})$. □

We will now 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. We will henceforth refer to Alg. 1 as AIA (Agent Interrogation Algorithm). To prove our next theorem we'll need some additional lemmas that we prove below. The first one mentions that AIA never prunes away a model whose possible concretization is an abstraction of the agent model, and the second one shows that AIA always terminates.

Lemma 7. Given an agent $\mathcal{A}$ with a model $M^{\mathcal{A}}$, and an abstract model M^{abs}, if AIA prunes away an abstract model M^{abs}, then no possible concretization of M^{abs} will be an abstraction of the agent model $M^{\mathcal{A}}$.

Proof. We prove this using simple inference. At each node in the lattice, we always prune away some of the models. If we discard an inconsistent model, it is because some palm tuple in the model has a different mode m, than that of $M^{\mathcal{A}}$. This incorrect palm tuple will also be present in all its concretizations, making all of them inconsistent with $M^{\mathcal{A}}$. Theorem 2 proves that at each node the models pruned away by AIA are not an abstraction of the agent model. This means that they have at least one of the pal tuples in a mode that does not match that of the agent model. Now concretizing such a model will only add other pal tuples in one of the three modes as explained in section 3.5, hence the incorrect mode of the pal tuple will remain unchanged thereby keeping making all possible concretizations of such a model an incorrect abstraction of the agent model. □

With the guarantee that we are not pruning away any correct possible model, we now prove that the agent interrogation algorithm will terminate, hence giving a solution.

Lemma 8. The Agent Interrogation Algorithm (Alg. 1) will always terminate.

Proof. As mentioned in Def. 16, in our subset lattice, "level" is equivalent to the number of refined pal tuples. At each step of the algorithm, when we consider a refinement in terms of pal tuples, we are left with one or more variants of the pal tuple. This ensures that we never refine the models more than once at a single level in the lattice. Since we refine at least one pal tuple in every iteration of the algorithm,

the algorithm is bound to terminate as the number of pal tuples is finite for a finite number of propositions and actions under consideration. □

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

Proof. Theorem 1 and Theorem 2 prove that whenever we get a prunable query, AIA discards only the models that are not abstractions of the agent model, thereby ensuring that no model that is an abstraction of the agent model is discarded. When we do not get a prunable query, AIA infers the correct precondition(s) of the failed action using *update_pal_ordering()*, hence the number of refined palm tuples always increase with the number of iterations of AIA (line 4 of AIA), thereby ensuring its termination in finite time. And when the algorithm terminates, the models that remain have all their responses consistent with that of the agent's model and hence are functionally equivalent to that of the agent model. □

3.6 Empirical Evaluation

We implemented AIA in Python to evaluate the efficacy of our approach.[3] In this implementation, initial states ($\mathcal{S}$, 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

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

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 evaluated three hypotheses using the experiments:

Hypothesis 1: The number of queries grows as we increase the number of *pal* tuples in the domains.

Hypothesis 2: The number of queries is lower than the observational learners to learn the complete model.

Hypothesis 3: The approach always learns the correct set of equivalent models.

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

3.6.1 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.

Domain	$\lvert P^* \rvert$	$\lvert A \rvert$	$\lvert \hat{\mathcal{Q}} \rvert$	$\mathbf{t}_\mu$ **(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 3. The number of queries ($\lvert \hat{\mathcal{Q}} \rvert$), 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.

Table 3 shows that the number of queries required increases with the number of predicates and actions in the domain, hence proving Hypothesis 1. 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.

3.6.2 Comparison with Observational Learner

We compare the performance of AIA with that of FAMA, state of the art observational learner, in terms of stability of the models learned 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 chapter ($\langle s_I, a_1, a_2, a_3, s_G \rangle$).

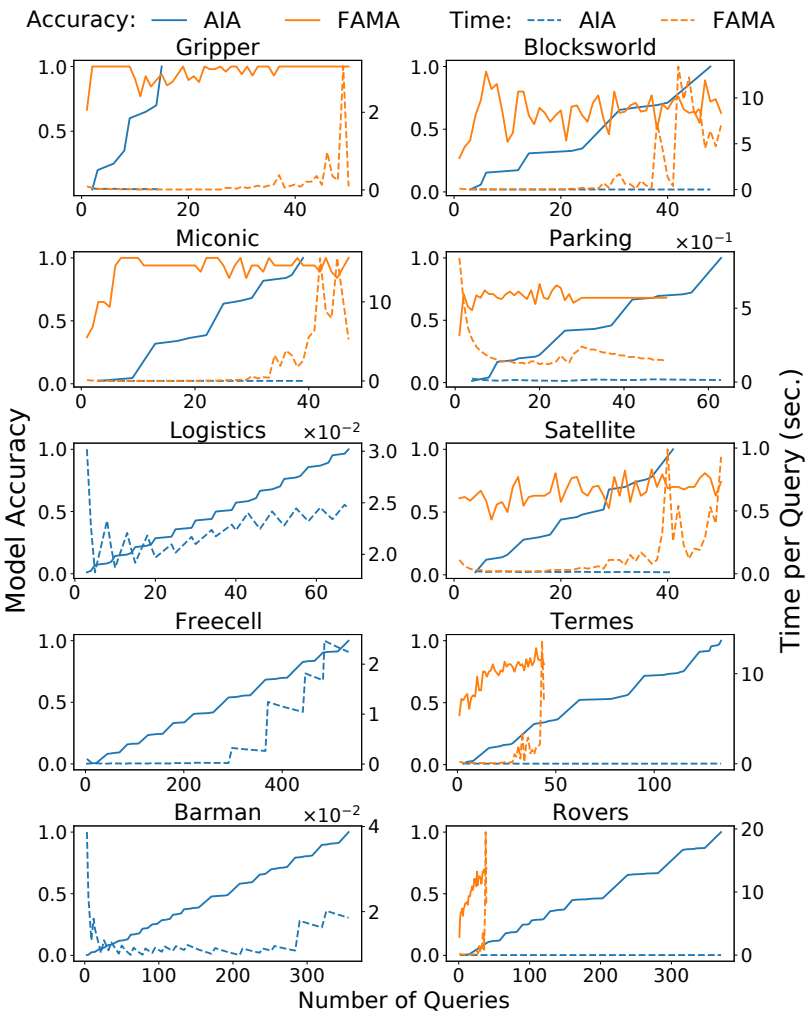


Figure 3. Performance comparison of AIA and FAMA in terms of model accuracy and time taken per query with an increasing number of queries.

Fig. 3 summarizes our findings. AIA takes lesser time per query and shows better convergence to the correct model, hence proving Hypothesis 2. 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,

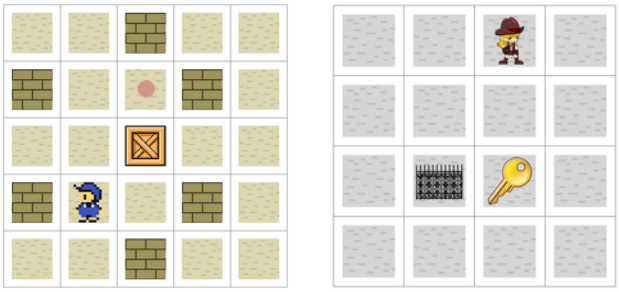


Figure 4. PDDLGym's simulated Sokoban (left) and Doors (right) environments used for the experiments.

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.

Also, not that AIA is able to learn the correct model for all the instances, hence proving Hypothesis 3.

3.6.3 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.

The classifiers are based on detecting objects in an image using colors (Duffy *et al.*, 2000; Khan *et al.*, 2012).

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. 4. 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.

3.7 Related Work

A number of researchers have explored the problem of learning agent models from observations of its behavior (Gil, 1994; Wang, 1994; Benson, 1995; Wu *et al.*, 2007; Yang *et al.*, 2007; Cresswell *et al.*, 2009; Zhuo and Kambhampati, 2013). Such action-model learning approaches have also found practical applications in robot navigation (Balac *et al.*, 2000), web-service description learning (Walsh and Littman, 2008), player behavior modeling (Krishnan *et al.*, 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. We now present a detailed comparison of our work with the related works.

3.7.1 Passive Observations Based Learners

Amir and Chang (2008) use logical filtering (Amir and Russell, 2003) to learn partially observable action models from the observation traces. Shahaf and Amir (2007); Zettlemoyer *et al.* (2008) and Shirazi and Amir (2011) also use logical filtering to acquire action models. 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, but it is not clear how to acquire such an oracle. Roy *et al.* (2020) learns the models in Property Specification Language (PSL) with very little overhead as compared to learning LTL formulas. All these approaches learn models at the propositional level.

Genetic programming-based techniques like EvoCK (Aler *et al.*, 1998), L2Plan Levine and Humphreys (2003), and LOUGA (Kučera and Barták, 2018) learn the domain rules by searching through a set of rules using genetic programming. LOUGA (Kučera and Barták, 2018) combines a genetic algorithm with an ad-hoc method to learn planning operators from observed plan traces. LOCM (Cresswell *et al.*, 2009), LOCM2 Cresswell and Gregory (2011), LOP (Gregory and Cresswell, 2015), and NLOCM (Gregory and Lindsay, 2016) present a class of algorithms that use finite-state machines to create action models from observed plan traces.

ARMS (Yang *et al.*, 2007; Wu *et al.*, 2007), AMAN (Zhuo and Kambhampati, 2013) and ML-CBP (Zhuo *et al.*, 2013; Zhuo and Kambhampati, 2017) leverage MAX-SAT to learn action models with partial or noisy traces. AMDN (Zhuo *et al.*, 2019) learn action models using noisy observations with sets of disordered and parallel actions. LAMP (Zhuo *et al.*, 2010) uses Markov Logic Networks (MLNs) (Richardson and Domingos, 2006) to learn complex action models involving quantifiers and logical implications from observed traces. LAWS (Zhuo *et al.*, 2011) use MLNs and knowledge from similar domains to learn action models for new domains. TRAMP (Zhuo and Yang, 2014) tries to minimize the observations it uses by leveraging knowledge from similar domains using MAX-SAT to learn a domain model for a new domain but still depends on passively collected observations.

FAMA (Aineto *et al.*, 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 some other approaches like LOUGA (Kučera and Barták, 2018) require a post-processing step to update the learned 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. Stern and Juba (2017), Juba *et al.* (2021), and Juba and Stern (2022) learn safe action models for various settings leveraging intermediate states in execution traces. 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.

Bonet and Geffner (2020) and Rodriguez *et al.* (2021) present approaches for learning relational models using a SAT-based method when the action schema, predicates, etc. are not available. These approaches take 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.

Online Learning Xu and Laird (2010) and Lamanna *et al.* (2021a) use online learning to learn an action model incrementally. The idea is to incorporate new observations to improve the action model. These approaches even though incremental, do not focus on acquiring directed observations that will help it learn faster, but rater work with already available observations.

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.

3.7.2 Non-Passive Observation Based Learners

Unlike the approaches that learn the action models using passively collected observations, there are some approaches that try to generate observations that help them direct the learning with lesser observations in general.

Active Learning 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 information measures. IRALe (Rodrigues *et al.*, 2011b) is one method that learns lifted transition modules by exploring actions in states where its partially learned preconditions almost hold. However, the effective feature set in active learning 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.

RL based Model Learning: 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 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.

Automata Learning: There is a large body of work on the active learning of automata of various types, namely DFA (Angluin, 1987; Rivest and Schapire, 1993; Kearns and Vazirani, 1994; Parekh and Honavar, 1996; Denis *et al.*, 2001; Bongard and Lipson, 2005; Isberner *et al.*, 2014; Volpato and Tretmans, 2015), NFA (Oncina and García, 1992; Dupont, 1996), Moore machine (Giantamidis and Tripakis, 2016; Moerman, 2018), Mealy machine (Shahbaz and Groz, 2009; Aarts and Vaandrager, 2010; Steffen *et al.*, 2011), Register Automata (Howar *et al.*, 2012; Aarts *et al.*, 2015; Cassel *et al.*, 2016), Büchi Automata (Maler and Pnueli, 1995; Farzan *et al.*, 2008; Angluin and Fisman, 2016; Li *et al.*, 2021), Symbolic Automata (Botinčan and Babić, 2013; Maler and Mens, 2014), etc. Angluin (1987) proposed the earliest approaches for actively learning DFAs using the L* algorithm, which leveraged membership queries and equivalence queries. A significant limitation of L* is that these machines use grounded states as inputs, limiting their application to small state spaces. There have been multiple optimizations, including replacing exhaustive observation tables with decision trees (Kearns and Vazirani, 1994); carefully choosing suffixes as columns instead of adding all the prefixes of the counterexamples in the observation table (Rivest and Schapire, 1993); reorganizing the decision trees by combining the previous two approaches (Isberner *et al.*, 2014); and using parameterized states in register automata (Cassel *et al.*, 2015). Even with these optimizations, the number of membership queries required to learn the automata is quadratic in the input size. In contrast, the number of equivalence queries required is linear in the size of the input (Isberner *et al.*, 2014). The abovementioned approaches use the minimal

adequate teacher (MAT) framework proposed by Angluin (1987). However, this has several drawbacks, which we cover below.

The first drawback of the MAT framework is that it needs a teacher who knows the correct model or approximates it to answer the equivalence queries correctly. Angluin (1988) showed that (i) using only membership queries, it is not possible to infer the correct DFA using a polynomial number of membership queries if the number of states of the target DFA is unknown; and (ii) even if the target number of states are known, an exponential number of membership queries are required. This requirement of using equivalence queries is a significant assumption that these theoretical approaches make. Implementing equivalence queries is difficult as functional equivalence is an undecidable problem. In practical settings, various systems use approximations to alleviate this. E.g., LearnLib (Raffelt *et al.*, 2009), an automata learning tool, uses basic test suite generation algorithms to approximate the answers to equivalence queries; Aarts *et al.* (2013) use the actual bank cards with a card reader as a teacher and "a random test suite with 1000 test traces of length 10 to 50 as equivalence oracle."; etc. Fiterau-Brostean *et al.* (2016) show that using such a method by LearnLib can incorrectly identify an incorrect model as a correct one. In our case even if we treat the simulator as a teacher, we don't need equivalence queries to guarantee that the model we learn is correct.

The second drawback of the MAT framework is that it needs knowledge of the input and output alphabet for working with membership and equivalence queries. In our framework, this information might correspond to the knowledge of the simulator's internal vocabulary (output alphabet) and the agent's primitive actions (input alphabet). Hence such automata learning would not work for the black-box agents that we work with. Note that the term "black box" in automata learning literature

corresponds to the systems where the "internal structure of the checked system is not revealed" (Peled *et al.*, 2001). In our setting, in addition to not knowing the internal working of the black-box agent, we also do not assume knowledge of the agent's vocabulary.

The third drawback of the MAT framework is that it can only learn the automata in terms of the primitive actions of the black-box system. This drawback also relates to the second drawback discussed earlier. In our approach, we can also learn the model in terms of high-level capabilities.

The final drawback of the MAT framework is that the learned automata have control states that might not be readily interpretable as they may not map to actual environment states but some property of the environment. Our assessment approach alleviates these concerns as the final model is easily interpretable as the preconditions and effects are defined in terms the user understands.

3.8 Concluding Remarks

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.

In this chapter, we saw the case when the stationary setting where neither the agent's nor the environment's model was changing. So once a model is learned using this approach it can be used as is whenever needed. But there are settings where the environment or the agent's model can get updated. In the next chapter, we will see how to design the assessment module to handle such cases.

Chapter 4

DIFFERENTIAL ASSESSMENT

In the last chapter we saw the method for assessment that learns symbolic models of AI agents with stationary models. This assumption fails to hold in settings where the agent's capabilities may change as a result of learning, adaptation, or other post-deployment modifications. Efficient assessment of agents in such settings is critical for learning the true capabilities of an AI system and for ensuring its safe usage. In this chapter, we propose a novel approach to *differentially* assess black-box AI agents that have drifted from their previously known models. As a starting point, we consider the fully observable and deterministic setting. We leverage sparse observations of the drifted agent's current behavior and knowledge of its initial model to generate an active querying policy that selectively queries the agent and computes an updated model of its functionality. Empirical evaluation shows that our approach is much more efficient than re-learning the agent model from scratch. We also show that the cost of differential assessment using our method is proportional to the amount of drift in the agent's functionality.

The primary contribution of this chapter is an algorithm for *differential assessment* of black-box AI systems (Fig. 5). This algorithm utilizes an initially known interpretable model of the agent as it was in the past, and a small set of observations of agent execution. It uses these observations to develop an incremental querying strategy that avoids the full cost of assessment from scratch and outputs a revised model of the agent's new functionality. One of the challenges in learning agent models from observational data is that reductions in agent functionality often do not correspond

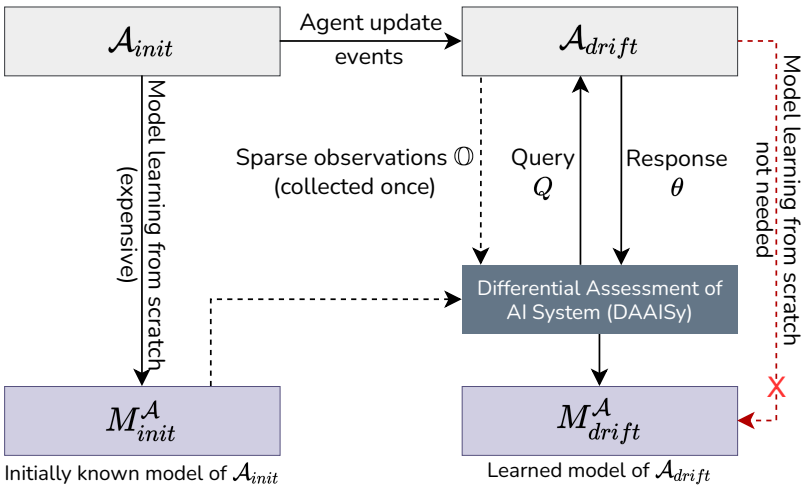


Figure 5. The Differential Assessment of AI System (DAAISy) takes as input the initially known model of the agent prior to model drift, available observations of the updated agent's behavior, and performs a selective dialog with the black-box AI agent to output its updated model through efficient model learning.

to specific "evidence" in behavioral observations, as the agent may not visit states where certain useful actions are no longer applicable. Our analysis shows that if the agent can be placed in an "optimal" planning mode, differential assessment can indeed be used to query the agent and recover information about reduction in functionality. This "optimal" planning mode is not necessarily needed for learning about increase in functionality. Empirical evaluations on a range of problems clearly demonstrate that our method is much more efficient than re-learning the agent's model from scratch. They also exhibit the desirable property that the computational cost of differential assessment is proportional to the amount of drift in the agent's functionality.

Running Example Consider a battery-powered rover with limited storage capacity that collects soil samples and takes pictures. Assume that its planning model is similar to IPC domain Rovers (Long and Fox, 2003). It has an action that collects a rock sample at a waypoint and stores it in a storage iff it has at least half of the battery capacity remaining. Suppose there was an update to the rover's system and as a

result of this update, the rover can now collect the rock sample only when its battery is full, as opposed to at least half-charged battery that it needed before. Mission planners familiar with the earlier system and unaware about the exact updates in the functionality of the rover would struggle to collect sufficient samples. This could jeopardise multiple missions if it is not detected in time.

This example illustrates how our system could be of value by differentially detecting such a drift in the functionality of a black-box AI system and deriving its true functionality.

The rest of this chapter is organized as follows: The next section presents background terminology. This is followed by a formalization of the differential model assessment problem in Section 3. Section 4 presents our approach for differential assessment by first identifying aspects of the agent's functionality that may be affected (Section 4.1) followed by the process for selectively querying the agent using a primitive set of queries. We present empirical evaluation of the efficiency of our approach on randomly generated benchmark planning domains in Section 5. Finally, we discuss relevant related work in Section 6 and conclude in Section 7.

4.1 Preliminaries

In this chapter, we learn the agent's representation as the PDDL models defined in chapter 2, that express an agent's functionalities in the form of STRIPS-like planning models (Fikes and Nilsson, 1971; McDermott *et al.*, 1998; Fox and Long, 2003). We represent them slightly differently, which is explained below.

4.1.1 Representing Models

We represent a model M using the set of all possible *pal-tuples* Γ_M of the form $\gamma = \langle p, a, \ell \rangle$, where a is a parameterized action header for an action in A, $p \in P^*$ is a possible lifted instantiation of a predicate in P, and $\ell \in \{pre,\ eff\}$ denotes a location in a, precondition or effect, where p can appear. A model M is thus a function $\mu_M : \Gamma_M \rightarrow \{+, -, \emptyset\}$ that maps each element in Γ_M to a *mode* in the set $\{+, -, \emptyset\}$. The assigned mode for a *pal-tuple* $\gamma \in \Gamma_M$ denotes whether p is present as a positive literal $(+)$, as a negative literal $(-)$, or absent $(\emptyset)$ in the precondition $(\ell = pre)$ or effect $(\ell = eff)$ of the action header a.

This formulation of models as *pal-tuples* allows us to view the modes for any predicate in an action's precondition and effect independently. However, at times it is useful to consider a model at a granularity of relationship between a predicate and an action. We address this by representing a model M as a set of *pa-tuples* Λ_M of the form $\langle p, a \rangle$ where a is a parameterized action header for an action in A, and $p \in P^*$ is a possible lifted instantiation of a predicate in P. Each *pa-tuple* can take a value of the form $\langle m_{pre}, m_{eff} \rangle$, where m_{pre} and m_{eff} represents the mode in which p appears in the precondition and effect of a, respectively. Since a predicate cannot appear as a positive (or negative) literal in both the precondition and effect of an action, $\langle +, + \rangle$ and $\langle -, - \rangle$ are not in the range of values that *pa-tuples* can take. Henceforth, in the context of a *pal-tuple* or a *pa-tuple*, we refer to a as an action instead of an action header.

4.1.2 Measure of Model Difference

Given two models $M_1 = \langle P, A_1 \rangle$ and $M_2 = \langle P, A_2 \rangle$, defined over the same sets of predicates P and action headers A, the difference between the two models $\Delta(M_1, M_2)$ is defined as the number of *pal-tuples* that differ in their modes in M_1 and M_2, i.e., $\Delta(M_1, M_2) = |\{\gamma \in P \times A \times \{+, -, \emptyset\} | \mu_{M_1}(\gamma) \neq \mu_{M_2}(\gamma)\}|$.

4.1.3 Abstracting Models

Several authors have explored the use of abstraction in planning (Sacerdoti, 1974; Giunchiglia and Walsh, 1992; Helmert *et al.*, 2007; Bäckström and Jonsson, 2013; Srivastava *et al.*, 2016). We define an abstract model as a model that does not have a mode assigned for at least one of the *pal-tuples*. Let Γ_M be the set of all possible *pal-tuples*, and ⍰ be an additional possible value that a *pal-tuple* can take. Assigning ⍰ mode to a *pal-tuple* denotes that its mode is unknown. An abstract model M is thus a function $\mu_M : \Gamma_M \rightarrow \{+, -, \emptyset, ⍰\}$ that maps each element in Γ_M to a *mode* in the set $\{+, -, \emptyset, ⍰\}$. Let $\mathcal{U}$ be the set of all abstract and concrete models that can possibly be expressed by assigning modes in $\{+, -, \emptyset, ⍰\}$ to each *pal-tuple* $\gamma \in \Gamma_M$. We now formally define model abstraction as follows:

Definition 13. Given models M_1 and M_2, M_2 is an abstraction of M_1 over the set of all possible *pal-tuples* Γ iff $\exists \Gamma_2 \subseteq \Gamma$ s.t. $\forall \gamma \in \Gamma_2$, $\mu_{M_2}(\gamma) =$ ⍰ and $\forall \gamma \in \Gamma \setminus \Gamma_2$, $\mu_{M_2}(\gamma) = \mu_{M_1}(\gamma)$.

4.1.4 Queries

We use queries to actively gain information about the functionality of an agent to learn its updated model. We assume that the agent can respond to a query using a simulator. The availability of such agents with simulators is a common assumption as most AI systems already use simulators for design, testing, and verification.

We use a notion of queries similar to chapter 3, to perform a dialog with an autonomous agent. These queries use an agent to determine what happens if it executes a sequence of actions in a given initial state. E.g., in the rovers domain, the rover could be asked: what happens when the action `sample_rock (rover1 storage1 waypoint1)` is executed in an initial state {`(equipped_rock_analysis rover1)`, `(battery_half rover1)`, `(at rover1 waypoint1)`}?

Formally, a *query* is a function that maps an agent to a response, defined as:

Definition 14. Given a set of predicates P, a set of actions A, and a set of objects O, a *query* $Q\langle s, \pi\rangle : \mathcal{A} \to \mathbb{N} \times S$ is parameterized by a start state $s_I \in S$ and a plan $\pi = \langle a_1, \ldots, a_N\rangle$, where S is the state space over P and O, and $\{a_1, \ldots, a_N\}$ is a subset of action space over A and O. It maps agents to responses $\theta = \langle n_F, s_F\rangle$ such that n_F is the length of the longest prefix of π that $\mathcal{A}$ can successfully execute and $s_F \in S$ is the result of that execution.

Responses to such queries can be used to gain useful information about the model drift. E.g., consider an agent with an internal model $M^{\mathcal{A}}_{drift}$ as shown in Tab. 4. If a query is posed asking what happens when the action `sample_rock (rover1 storage1 waypoint1)` is executed in an initial state {`(equipped_rock_analysis rover1)`, `(battery_half rover1)`, `(at rover1 waypoint1)`}, the agent would

respond $\langle 0, \{$`(equipped_rock_analysis rover1)`, `(battery_half rover1)`, `(at rover1 waypoint1)`$\}\rangle$, representing that it was not able to execute the plan, and the resulting state was {`(equipped_rock_analysis rover1)`, `(battery_half rover1)`, `(at rover1 waypoint1)`} (same as the initial state in this case). Note that this response is inconsistent with the model $M^{\mathcal{A}}_{init}$, and it can help in identifying that the precondition of action `sample_rock (?r ?s ?w)` has changed.

4.2 Formal Framework

Our objective is to address the problem of differential assessment of black-box AI agents whose functionality may have changed from the last known model. Without loss of generality, we consider situations where the set of action headers is same because the problem of differential assessment with changing action headers can be reduced to that with uniform action headers. This is because if the set of actions has increased, new actions can be added with empty preconditions and effects to $M^{\mathcal{A}}_{init}$, and if it has decreased, $M^{\mathcal{A}}_{init}$ can be reduced similarly. We assume that the predicate vocabulary used in the two models is the same; extension to situations where the vocabulary changes can be used to model open-world scenarios. However, that extension is beyond the scope of this chapter.

Suppose an agent $\mathcal{A}$'s functionality was known as a model $M^{\mathcal{A}}_{init} = \langle P, \mathcal{A}_{init} \rangle$, and we wish to assess its current functionality as the model $M^{\mathcal{A}}_{drift} = \langle P, \mathcal{A}_{drift} \rangle$. The drift in the functionality of the agent can be measured by changes in the preconditions and/or effects of all the actions in $\mathcal{A}_{init}$. The extent of the drift between $M^{\mathcal{A}}_{init}$ and $M^{\mathcal{A}}_{drift}$ is represented as the model difference $\Delta(M^{\mathcal{A}}_{init}, M^{\mathcal{A}}_{drift})$.

We formally define the problem of differential assessment of an AI agent below.

Model	Precondition	Effect
$M^{\mathcal{A}}_{init}$	(equipped_rock_analysis ?r) → (battery_half ?r) (at ?r ?w)	(rock_sample_taken ?r) (store_full ?r ?s) ¬(battery_half ?r) (battery_reserve ?r)
$M^{\mathcal{A}}_{init}$	(equipped_rock_analysis ?r) → (battery_full ?r) (at ?r ?w)	(rock_sample_taken ?r) (store_full ?r ?s) ¬(battery_full ?r) (battery_half ?r)

Table 4. `sample_rock (?r ?s ?w)` action of the agent $\mathcal{A}$ in $M^{\mathcal{A}}_{init}$ and a possible drifted model $M^{\mathcal{A}}_{drift}$.

Definition 15. Given an agent $\mathcal{A}$ with a functionality model $M^{\mathcal{A}}_{init}$, and a set of observations $\mathbb{O}$ collected using its current version of $\mathcal{A}_{drift}$ with unknown functionality $M^{\mathcal{A}}_{drift}$, the *differential model assessment* problem $\langle M^{\mathcal{A}}_{init}, M^{\mathcal{A}}_{drift}, \mathbb{O}, \mathcal{A}\rangle$ is defined as the problem of inferring $\mathcal{A}$ in form of $M^{\mathcal{A}}_{drift}$ using the inputs $M^{\mathcal{A}}_{init}$, $\mathbb{O}$, and $\mathcal{A}$.

We wish to develop solutions to the problem of differential assessment of AI agents that are more efficient than re-assessment from scratch.

4.2.1 Correctness of Assessed Model

We now discuss the properties that a model, which is a solution to the differential model assessment problem, should satisfy. A critical property of such models is that they should be consistent with the observation traces. We formally define consistency of a model w.r.t. an observation trace as follows:

Definition 16. Let o be an observation trace $\langle s_0, a_1, s_1, a_2, \ldots, s_{n-1}, a_n, s_n\rangle$. A model $M = \langle P, A\rangle$ is *consistent with the observation trace* o iff $\forall i \in \{1, .., n\}$ $\exists a \in A$ and a_i is a grounding of action a *s.t.* $s_{i-1} \models \mathit{pre}(a_i) \wedge \forall\, l \in \mathit{eff}(a_i)\ s_i \models l$.

In addition to being consistent with observation traces, a model should also be

consistent with the queries that are asked and the responses that are received while actively inferring the model of the agent's new functionality. We formally define consistency of a model with respect to a query and a response as:

Definition 17. Let $M = \langle P, A \rangle$ be a model; O be a set of objects; $Q = \langle s_I, \pi = \langle a_1, \ldots a_n \rangle \rangle$ be a query defined using P, A, and O, and let $\theta = \langle n_F, s_F \rangle$, $(n_F \leq n)$ be a response to Q. M is *consistent with the query-response* $\langle Q, \theta \rangle$ iff there exists an observation trace $\langle s_I, a_1, s_1, \ldots, a_{n_F}, s_{n_F} \rangle$ that M is consistent with and $s_{n_F} \not\models pre(a_{n_F+1})$ where $pre(a_{n_F+1})$ is the precondition of a_{n_F+1} in M.

We now discuss our methodology for solving the problem of differential assessment of AI systems.

4.3 Differential Assessment of AI Systems

Differential **A**ssessment of **AI Sy**stems (Alg. 19) -- DAAISy -- takes as input an agent $\mathcal{A}$ whose functionality has drifted, the model $M^{\mathcal{A}}_{init} = \langle P, A \rangle$ representing the previously known functionality of $\mathcal{A}$, a set of arbitrary observation traces $\mathbb{O}$, and a set of random states $\mathcal{S} \subseteq S$. Alg. 19 returns a set of updated models $\mathcal{M}^{\mathcal{A}}_{drift}$, where each model $M^{\mathcal{A}}_{drift} \in \mathcal{M}^{\mathcal{A}}_{drift}$ represents $\mathcal{A}$'s updated functionality and is consistent with all observation traces $o \in \mathbb{O}$.

A major contribution of this work is to introduce an approach to make inferences about not just the expanded functionality of an agent but also its reduced functionality using a limited set of observation traces. Situations where the scope of applicability of an action reduces, i.e., the agent can no longer use an action a to reach state s' from state s while it could before (e.g., due to addition of a precondition literal), are particularly difficult to identify because observing its behavior does not readily reveal

what it cannot do in a given state. Most observation based action-model learners, even when given access to an incomplete model to start with, fail to make inferences about reduced functionality. DAAISy uses two principles to identify such a functionality reduction. First, it uses active querying so that the agent can be made to reveal failure of reachability, and second, we show that if the agent can be placed in optimal planning mode, plan length differences can be used to infer a reduction in functionality.

DAAISy performs two major functions; it first identifies a salient set of *pal-tuples* whose modes were likely affected (line 1 of Alg. 19), and then infers the mode of such affected *pal-tuples* accurately through focused dialog with the agent (line 2 onwards of Alg. 19). In Sec. 4.3.1, we present our method for identifying a salient set of potentially affected *pal-tuples* that contribute towards expansion in the functionality of the agent through inference from available arbitrary observations. We then discuss the problem of identification of *pal-tuples* that contribute towards reduction in the functionality of the agent and argue that it cannot be performed using successful executions in observations of satisficing behavior. We show that *pal-tuples* corresponding to reduced functionality can be identified if observations of optimal behavior of the agent are available (Sec. 4.3.1). Finally, we present how we infer the nature of changes in all affected *pal-tuples* through a query-based interaction with the agent (Sec. 4.3.2) by building upon the Agent Interrogation Algorithm (AIA) (Verma *et al.*, 2021a). Identifying affected *pal-tuples* helps reduce the computational cost of querying as opposed to the exhaustive querying strategy used by AIA. We now discuss the two major functions of Alg. 19 in detail.

Algorithm 3: Differential Assessment of AI Systems

Input : $M^{\mathcal{A}}_{init}$, $\mathbb{O}$, $\mathcal{A}$, $\mathcal{S}$
Output : $\mathcal{M}^{\mathcal{A}}_{drift}$

$\Gamma_\delta \leftarrow$ *identify_affected_pals()*
$M_{abs} \leftarrow$ set *pal-tuples* in $M^{\mathcal{A}}_{init}$ corresponding to Γ_δ to ?
$\mathcal{M}^{\mathcal{A}}_{drift} \leftarrow \{M_{abs}\}$
for *each* γ *in* Γ_δ **do**
  for *each* M_{abs} *in* $\mathcal{M}^{\mathcal{A}}_{\text{drift}}$ **do**
    $\mathcal{M}_{abs} \leftarrow M_{abs} \times \{\gamma^+, \gamma^-, \gamma^\emptyset\}$
    $\mathcal{M}_{sieved} \leftarrow \{\}$ **if** *action corresponding to* γ*:* γ_a *in* $\mathbb{O}$ **then**
      $s_{pre} \leftarrow$ *states_where_*γ_a*_applicable*$(\mathbb{O}, \gamma_a)$
      $Q \leftarrow \langle s_{pre} \setminus \{\gamma_p \cup \neg\gamma_p\},\ \gamma_a \rangle$
      $\theta \leftarrow$ *ask_query*$(\mathcal{A}, Q)$
      $\mathcal{M}_{sieved} \leftarrow$ *sieve_models*$(\mathcal{M}_{abs}, Q, \theta)$
    else
      for *each pair* $\langle M_i, M_j \rangle$ *in* $\mathcal{M}_{\text{abs}}$ **do**
        $Q \leftarrow$ *generate_query*(M_i, M_j, γ, S)
        $\theta \leftarrow$ *ask_query*$(\mathcal{A}, Q)$
        $\mathcal{M}_{sieved} \leftarrow$ *sieve_models*$(\{M_i, M_j\}, Q, \theta)$
    $\mathcal{M}_{abs} \leftarrow \mathcal{M}_{abs} \setminus \mathcal{M}_{sieved}$
  $\mathcal{M}^{\mathcal{A}}_{drift} \leftarrow \mathcal{M}_{abs}$
return $\mathcal{M}^{\mathcal{A}}_{\text{drift}}$

4.3.1 Identifying Potentially Affected *pal-tuples*

We identify a reduced set of *pal-tuples* whose modes were potentially affected during the model drift, denoted by Γ_δ, using a small set of available observation traces $\mathbb{O}$. We draw two kinds of inferences from these observation traces: inferences about expanded functionality, and inferences about reduced functionality. We discuss our method for inferring Γ_δ for both types of changes in the functionality below.

$\langle m_{pre}, m_{eff} \rangle$	*(pos,pos)*	*(pos,neg)*	*(neg,pos)*	*(neg,neg)*
$\langle +, - \rangle$	✗	✓	✗	✗
$\langle +, \emptyset \rangle$	✓	✗	✗	✗
$\langle -, + \rangle$	✗	✗	✓	✗
$\langle -, \emptyset \rangle$	✗	✗	✗	✓
$\langle \emptyset, + \rangle$	✓	✗	✓	✗
$\langle \emptyset, - \rangle$	✗	✓	✗	✓
$\langle \emptyset, \emptyset \rangle$	✓	✗	✗	✓

Table 5. Each row represents a possible value $\langle m_{pre}, m_{eff} \rangle$ for a *pa-tuple* $\langle p, a \rangle$. Each column represents a possible tuple representing presence of predicate p in the pre- and post-states of an action triplet $\langle s_i, a, s_{i+1} \rangle$ (discussed in Sec.4.3.1). The cells represent whether a value for *pa-tuple* is consistent with an action triplet in observation traces.

4.3.1.1 Expanded Functionality

To infer expanded functionality of the agent, we use the previously known model of the agent's functionality and identify its differences with the possible behaviors of the agent that are consistent with $\mathbb{O}$. To identify the *pal-tuples* that directly contribute to an expansion in the agent's functionality, we perform an analysis similar to (Stern and Juba, 2017), but instead of bounding the predicates that can appear in each action's precondition and effect, we bound the range of possible values that each *pa-tuple* in $M^{\mathcal{A}}_{drift}$ can take using Tab. 5. For any *pa-tuple*, a direct comparison between its value in $M^{\mathcal{A}}_{init}$ and possible inferred values in $M^{\mathcal{A}}_{drift}$ provides an indication of whether it was affected.

To identify possible values for a *pa-tuple* $\langle p, a \rangle$, we first collect a set of all the action-triplets from $\mathbb{O}$ that contain the action a. For a given predicate p and state s, if $s \models p$ then the presence of predicate p is represented as *pos*, similarly, if $s \models \neg p$ then the presence of predicate p is represented as *neg*. Using this representation, a tuple

of predicate presence $\in$ {*(pos,pos)*, *(pos,neg)*, *(neg,pos)*, *(neg,neg)*} is determined for the *pa-tuple* $\langle p, a\rangle$ for each action triplet $\langle s, a, s'\rangle \in \mathbb{O}$ by analyzing the presence of predicate p in the pre- and post-states of the action triplets. Possible values of the *pa-tuple* that are consistent with $\mathbb{O}$ are directly inferred from the Tab. 5 using the inferred tuples of predicate presence. E.g., for a *pa-tuple*, the values $\langle +, -\rangle$ and $\langle \emptyset, -\rangle$ are consistent with (pos, neg), whereas, only $\langle \emptyset, +\rangle$ is consistent with (pos, pos) and (neg, pos) tuples of predicate presence that are inferred from $\mathbb{O}$.

Once all the possible values for each *pa-tuple* in $M^{\mathcal{A}}_{drift}$ are inferred, we identify *pa-tuples* whose previously known value in $M^{\mathcal{A}}_{init}$ is no longer possible due to inconsistency with $\mathbb{O}$. The *pal-tuples* corresponding to such *pa-tuples* are added to the set of potentially affected *pal-tuples* Γ_δ. Our method also infers the correct modes of a subset of *pal-tuples*. E.g., consider a predicate p and two actions triplets in $\mathbb{O}$ of the form $\langle s_1, a, s'_1\rangle$ and $\langle s_2, a, s'_2\rangle$ that satisfy $s_1 \models p$ and $s_2 \models \neg p$. Such an observation clearly indicates that p is not in the precondition of action a, i.e., mode for $\langle p, a\rangle$ in the precondition is $\emptyset$. Such inferences of modes are used to update the known functionality of the agent. We remove such *pal-tuples*, whose modes are already inferred, from Γ_δ.

A shortcoming of direct inference from successful executions in available observation traces is that it cannot learn any reduction in the functionality of the agent, as discussed in the beginning of Sec. 4.3. We now discuss our method to address this limitation and identify a larger set of potentially affected *pal-tuples*.

4.3.1.2 Reduced Functionality

We conceptualize reduction in functionality as an increase in the optimal cost of going from one state to another. More precisely, reduction in functionality represents

situations where there exist states s_i, s_j such that the minimum cost of going from s_i to s_j is higher in $M^{\mathcal{A}}_{drift}$ than in $M^{\mathcal{A}}_{init}$. In this chapter, this cost refers to the number of steps between the pair of states as we consider unit action costs. This notion encompasses situations with reductions in reachability as a special case. In practice, a reduction in functionality may occur if the precondition of at least one action in $M^{\mathcal{A}}_{drift}$ has new *pal-tuples*, or the effect of at least one of its actions has new *pal-tuples* that conflict with other actions required for reaching certain states.

Our notion of reduced functionality captures all the variants of reduction in functionality. However, for clarity, we illustrate an example that focuses on situations where precondition of an action has increased. Consider the case from Tab. 4 where $\mathcal{A}$'s model gets updated from $M^{\mathcal{A}}_{init}$ to $M^{\mathcal{A}}_{drift}$. The action *sample_rock*'s applicability in $M^{\mathcal{A}}_{drift}$ has reduced from that in $M^{\mathcal{A}}_{init}$ as $\mathcal{A}$ can no longer sample rocks in situations where the battery is half charged but needs a fully charged battery to be able to execute the action. In such scenarios, instead of relying on observation traces, our method identifies traces containing indications of actions that were affected either in their precondition or effect, discovers additional salient *pal-tuples* that were potentially affected, and adds them to the set of potentially affected *pal-tuples* Γ_δ. To find *pal-tuples* corresponding to reduced functionality of the agent, we place the agent in an optimal planning mode and assume limited availability of observation traces $\mathbb{O}$ in the form of optimal unit-cost state-action trajectories $\langle s_0, a_1, s_1, a_2, \ldots, s_{n-1}, a_n, s_n \rangle$. We generate optimal plans using $M^{\mathcal{A}}_{init}$ for all pairs of states in $\mathbb{O}$. We hypothesize that, if for a pair of states, the plan generated using $M^{\mathcal{A}}_{init}$ is shorter than the plan observed in $\mathbb{O}$, then some functionality of the agent has reduced.

Our method performs comparative analysis of optimality of the observation traces against the optimal solutions generated using $M^{\mathcal{A}}_{init}$ for same pairs of initial and final

states. To begin with, we extract all the continuous state sub-sequences from $\mathbb{O}$ of the form $\langle s_0, s_1, \ldots, s_n \rangle$ denoted by $\mathbb{O}_{drift}$ as they are all optimal. We then generate a set of planning problems $\mathcal{P}$ using the initial and final states of trajectories in $\mathbb{O}_{drift}$. Then, we provide the problems in $\mathcal{P}$ to $M^{\mathcal{A}}_{init}$ to get a set of optimal trajectories $\mathbb{O}_{init}$. We select all the pairs of optimal trajectories of the form $\langle o_{init}, o_{drift} \rangle$ for further analysis such that the length of $o_{init} \in \mathbb{O}_{init}$ for a problem is shorter than the length of $o_{drift} \in \mathbb{O}_{drift}$ for the same problem. For all such pairs of optimal trajectories, a subset of actions in each $o_{init} \in \mathbb{O}_{init}$ were likely affected due to the model drift. We focus on identifying the first action in each $o_{init} \in \mathbb{O}_{init}$ that was definitely affected.

To identify the affected actions, we traverse each pair of optimal trajectories $\langle o_{init}, o_{drift} \rangle$ simultaneously starting from the initial states. We add all the *pal-tuples* corresponding to the first differing action in o_{init} to Γ_δ. We do this because there are only two possible explanations for why the action differs: (i) either the action in o_{init} was applicable in a state using $M^{\mathcal{A}}_{init}$ but has become inapplicable in the same state in $M^{\mathcal{A}}_{drift}$, or (ii) it can no longer achieve the same effects in $M^{\mathcal{A}}_{drift}$ as $M^{\mathcal{A}}_{init}$. We also discover the first actions that are applicable in the same states in both the trajectories but result in different states. The effect of such actions has certainly changed in $M^{\mathcal{A}}_{drift}$. We add all the *pal-tuples* corresponding to such actions to Γ_δ. In the next section, we describe our approach to infer the correct modes of *pal-tuples* in Γ_δ.

4.3.2 Investigating Affected *pal-tuples*

This section explains how the correct modes of *pal-tuples* in Γ_δ are inferred (line 2 onwards of Alg.1). Alg. 19 creates an abstract model in which all the *pal-tuples* that

are predicted to have been affected are set to ⍰ (line 2). It then iterates over all *pal-tuples* with mode ⍰ (line 4).

4.3.2.1 Removing Inconsistent Models

Our method generates candidate abstract models and then removes the abstract models that are not consistent with the agent (lines 7-18 of Alg. 19). For each *pal-tuple* $\gamma \in \Gamma$, the algorithm computes a set of possible abstract models $\mathcal{M}_{abs}$ by assigning the three mode variants $+$, $-$, and $\emptyset$ to the current *pal-tuple* γ in model M_{abs} (line 6). Only one model in $\mathcal{M}_{abs}$ corresponds to the agent's updated functionality.

If the action γ_a in the *pal-tuple* γ is present in the set of action triplets generated using $\mathbb{O}$, then the pre-state of that action s_{pre} is used to create a state s_I (lines 9-10). s_I is created by removing the literals corresponding to predicate γ_p from s_{pre}. We then create a query $Q = \langle s_I, \langle \gamma_a \rangle \rangle$ (line 10), and pose it to the agent $\mathcal{A}$ (line 11). The three models are then sieved based on the comparison of their responses to the query Q with that of $\mathcal{A}$'s response θ to Q (line 12). We use the same mechanism as AIA for sieving the abstract models.

If the action corresponding to the current *pal-tuple* γ being considered is not present in any of the observed action triplets, then for every pair of abstract models in $\mathcal{M}_{abs}$ (line 14), we generate a query Q using a planning problem (line 15). We then pose the query Q to the agent (line 16) and receive its response θ. We then sieve the abstract models by asking them the same query and discarding the models whose responses are not consistent with that of the agent (line 17). The planning problem that is used to generate the query and the method that checks for consistency of abstract models' responses with that of the agent are used from AIA.

Finally, all the models that are not consistent with the agent's updated functionality are removed from the possible set of models M_{abs}. The remaining models are returned by the algorithm. Empirically, we find that only one model is always returned by the algorithm.

4.3.3 Correctness

We now show that the learned drifted model representing the agent's updated functionality is consistent as defined in Def. 8 and Def. 9. We will first need to prove that Tab. 5 is constructed correctly. We do this by using the following result:

Lemma 9. Given an action triplet $\langle s, a, s' \rangle \in \mathbb{O}$ and a predicate $p \in P$, Tab. 2 correctly represents the set of values for the pair of modes $\langle m_{pre}, m_{\mathit{eff}} \rangle$, where m_{pre} and m_{eff} are the modes of predicate p in the precondition and effect of action a respectively, that are *consistent* with the action triplet.

Proof. Given an action triplet $\langle s, a, s' \rangle$, if a predicate $p \in P$ is true (or false) in s (or s'), then it cannot be false (or true) in the precondition (or effect) of a. Hence, if p is true in both s and s', its value for $\langle m_{pre}, m_{\mathit{eff}} \rangle$ can only be $\langle +, \emptyset \rangle$, $\langle \emptyset, + \rangle$, or $\langle \emptyset, \emptyset \rangle$. If p is true in s but false in s', its value for $\langle m_{pre}, m_{\mathit{eff}} \rangle$ can only be $\langle +, - \rangle$ or $\langle \emptyset, - \rangle$. If p is false in s but true in s', its value for $\langle m_{pre}, m_{\mathit{eff}} \rangle$ can only be $\langle -, + \rangle$ or $\langle \emptyset, + \rangle$. Finally, if p is false in both s and s', its value for $\langle m_{pre}, m_{\mathit{eff}} \rangle$ can only be $\langle -, \emptyset \rangle$, $\langle \emptyset, - \rangle$, or $\langle \emptyset, \emptyset \rangle$. Thus, for an observed action triplet $\langle s, a, s' \rangle$, Tab. 2 shows all the possible values for a $p \in P$ in the precondition and effect of a that do not conflict with the presence (or absence) of p in s and s' respectively. □

We now prove the two smaller results that combine to form the theorem we are trying to build.

Lemma 10. Given a set of observation traces $\mathbb{O}$ generated by the drifted agent $\mathcal{A}_{drift}$, each of the models $M = \langle P, A \rangle$ in $\mathcal{M}^{\mathcal{A}}_{drift}$ learned by Alg. 1 are *consistent* with respect to all the observation traces $o \in \mathbb{O}$.

Proof. Given that the action triplets in the set of observations $\mathbb{O}$ are generated using the same functionality of the deterministic agent after the drift $\mathcal{A}_{drift}$ (i.e., all the observations correspond to the same drifted model), any two different action triplets in $\mathbb{O}$ containing groundings of the same action a_i must have pre- and post-states that do not contradict each other. Now, for any action triplet $\langle s_{i-1}, a_i, s_i \rangle$ that is part of an observation trace $o \in \mathbb{O}$, when we consider the correct values for $\langle m_{pre}, m_{eff} \rangle$ for a *pa-tuple* $\langle p, a_i \rangle$ such that $p \in P$, we only consider the values for $\langle m_{pre}, m_{eff} \rangle$ that are shown in Tab. 2. For multiple actions triplets, possible values for $\langle m_{pre}, m_{eff} \rangle$ can be found by taking an intersection of the sets of values for $\langle m_{pre}, m_{eff} \rangle$ for each action triplet found using Tab. 2. Using Lemma 1, this ensures that the learned model M is consistent with all the action triplets in an observation trace $o \in \mathbb{O}$. Since an observation trace is a sequence of action triplets, the learned model $M \in \mathcal{M}^{\mathcal{A}}_{drift}$ is consistent with all the observation traces in the set of observation traces $\mathbb{O}$. □

Lemma 11. Given a set of queries Q posed to $\mathcal{A}_{drift}$ by Alg. 1, and the model $M^{\mathcal{A}}_{init}$ representing the agent's functionality prior to the drift, each of the models $M = \langle P, A \rangle$ in $\mathcal{M}^{\mathcal{A}}_{drift}$ learned by Alg. 1 are *consistent* with respect to all query-responses $\langle q, \theta \rangle$ for all the queries $q \in Q$.

Proof. The agent responds to the query $q = \langle s_I, \pi = \langle a_1, \ldots a_n \rangle \rangle$ using the drifted model with a response $\theta = \langle n_F, s_F \rangle$. This response can only be generated if there

exists an observation trace $\langle s_I, a_1, s_1, \ldots, a_{n_F}, s_{n_F} \rangle$ of length n_F that can take the agent starting from state s_I to the state s_{n_F}. Now, pruning models based on responses of the agent follows the criteria shown in Tab. 5. Hence, the only modes we consider for p in the precondition and effect of a_{n_F} are the ones that do not conflict with the presence (or absence) of p in s_{n_F-1} and s_{n_F} respectively. The modes for any predicate in other actions are not fixed using responses to queries. Hence, the learned model $M \in \mathcal{M}^{\mathcal{A}}_{drift}$ is consistent with all the query-responses $\langle q, \theta \rangle$ for all the queries $q \in Q$. □

Theorem 4. Given a set of observation traces $\mathbb{O}$ generated by the drifted agent $\mathcal{A}_{drift}$, a set of queries Q posed to $\mathcal{A}_{drift}$ by Alg. 1, and the model $M^{\mathcal{A}}_{init}$ representing the agent's functionality prior to the drift, each of the models $M = \langle P, A \rangle$ in $\mathcal{M}^{\mathcal{A}}_{drift}$ learned by Alg. 1 are *consistent* with respect to all the observation traces $o \in \mathbb{O}$ and query-responses $\langle q, \theta \rangle$ for all the queries $q \in Q$.

Proof. This theorem is conjunction of Lemma 10 and Lemma 11. Since both of the lemmas are proven to be true, this theorem is also true. □

.

There exists a finite set of observations that if collected will allow Alg. 19 to achieve 100% correctness with any amount of drift: this set corresponds to observations that allow line 1 of Alg. 19 to detect a change in the functionality. This includes an action triplet in an observation trace hinting at increased functionality, or a shorter plan using the previously known model hinting at reduced functionality. Thus, models learned by DAAISy are guaranteed to be completely correct irrespective of the amount of the drift if such a finite set of observations is available. While using queries significantly

reduces the number of observations required, asymptotic guarantees subsume those of passive model learners while ensuring convergence to the true model.

4.4 Empirical Evaluation

In this section, we evaluate our approach for assessing a black-box agent to learn its model using information about its previous model and available observations. We implemented the algorithm for DAAISy in Python[4] and tested it on six planning benchmark domains from the International Planning Competition (IPC) [5]. We used the IPC domains as the unknown drifted models and generated six initial domains at random for each domain in our experiments.

To assess the performance of our approach with increasing drift, we employed two methods for generating the initial domains: (a) dropping the *pal-tuples* already present, and (b) adding new *pal-tuples*. For each experiment, we used both types of domain generation. We generated different initial models by randomly changing modes of random *pal-tuples* in the IPC domains. Thus, in all our experiments an IPC domain plays the role of ground truth M^*_{drift} and a randomized model is used as $M^{\mathcal{A}}_{init}$.

We use a very small set of observation traces $\mathbb{O}$ (single observation trace containing 10 action triplets) in all the experiments for each domain. To generate this set, we gave the agent a random problem instance from the IPC corresponding to the domain used by the agent. The agent then used Fast Downward (Helmert, 2006) with LM-Cut heuristic (Helmert and Domshlak, 2009) to produce an optimal solution for the given problem. The generated observation trace is provided to DAAISy as input in addition

[4]Code available at https://github.com/AAIR-lab/DAAISy

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

to a random $M^{\mathcal{A}}_{init}$ as discussed in Alg. 19. The exact same observation trace is used in all experiments of the same domain, without the knowledge of the drifted model of the agent, and irrespective of the amount of drift.

We measure the final accuracy of the learned model $M^{\mathcal{A}}_{drift}$ against the ground truth model M^{*}_{drift} using the measure of model difference $\Delta(M^{\mathcal{A}}_{drift}, M^{*}_{drift})$. We also measure the number of queries required to learn a model with significantly high accuracy. We compare the efficiency of DAAISy (our approach) with the Agent Interrogation Algorithm (AIA) (Verma *et al.*, 2021a) as it is the most closely related querying-based system. All of our experiments were executed on 5.0 GHz Intel i9 CPUs with 64 GB RAM running Ubuntu 18.04. We now discuss our results in detail below.

4.4.1 Results

We evaluated the performance of DAAISy along 2 directions; the number of queries it takes to learn the updated model $M^{\mathcal{A}}_{drift}$ with increasing amount of drift, and the correctness of the model $M^{\mathcal{A}}_{drift}$ it learns compared to M^{*}_{drift}.

4.4.1.1 Efficiency in Number of Queries

As seen in Fig. 6, the computational cost of assessing each agent, measured in terms of the number of queries used by DAAISy, increases as the amount of drift in the model M^{*}_{drift} increases. This is expected as the amount of drift is directly proportional to the number of *pal-tuples* affected in the domain. This increases the number of *pal-tuples* that DAAISy identifies as affected as well as the number of

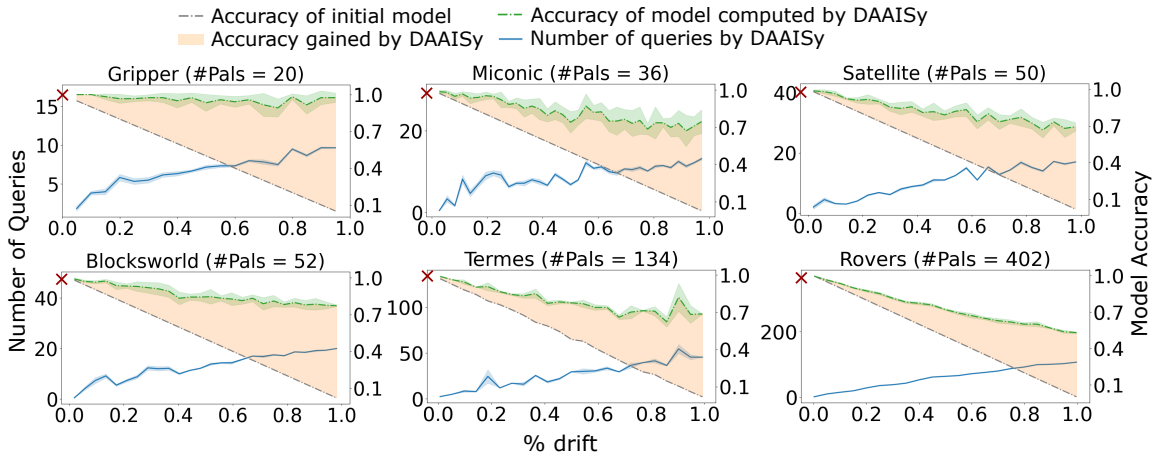


Figure 6. The number of queries used by DAAISy (our approach) and AIA (marked × on y-axis), as well as accuracy of model computed by DAAISy with increasing amount of drift. Amount of drift equals the ratio of drifted *pal-tuples* and the total number of *pal-tuples* in the domains (nPals). The number of action triplets in the observation trace used for each domain is 10.

queries as a result. As demonstrated in the plots, the standard deviation for number of queries remains low even when we increase the amount of drift, showing the stability of DAAISy.

4.4.1.2 Comparison with AIA

Tab. 6 shows the average number of queries that AIA took to achieve the same level of accuracy as our approach for 50% drifted models, and DAAISy requires significantly fewer queries to reach the same levels of accuracy compared to AIA. Fig. 6 also demonstrates that DAAISy always takes fewer queries as compared to AIA to reach reasonably high levels of accuracy.

This is because AIA does not use information about the previously known model of the agent and thus ends up querying for all possible *pal-tuples*. DAAISy, on the other

Domain	#Pals	AIA	DAAISy
Gripper	20	15.0	6.5
Miconic	36	32.0	7.7
Satellite	50	34.0	9.0
Blocksworld	52	40.0	11.4
Termes	134	115.0	27.0
Rovers	402	316.0	61.0

Table 6. The average number of queries taken by AIA to achieve the same level of accuracy as DAAISy (our approach) for 50% drifted models.

hand, predicts the set of *pal-tuples* that might have changed based on the observations collected from the agent and thus requires significantly fewer queries.

4.4.1.3 Correctness of Learned Model

DAAISy computes models with at least 50% accuracy in all six domains even when they have completely drifted from their initial model, i.e., $\Delta(M^{\mathcal{A}}_{drift}, M^{*}_{drift}) = nPals$. It attains nearly accurate models for Gripper and Blocksworld for upto 40% drift. Even in scenarios where the agent's model drift is more than 50%, DAAISy achieves at least 70% accuracy in five domains. Note that DAAISy is guaranteed to find the correct mode for an identified affected *pal-tuple*. The reason for less than 100% accuracy when using DAAISy is that it does not predict a *pal-tuple* to be affected unless it encounters an observation trace conflicting with $M^{\mathcal{A}}_{init}$. Thus, the learned model $M^{\mathcal{A}}_{drift}$, even though consistent with all the observation traces, may end up being inaccurate when compared to M^{*}_{drift}.

4.4.1.4 Discussion

AIA always ends up learning completely accurate models, but as noted above, this is because AIA queries exhaustively for all the *pal-tuples* in the model. There is a clear trade-off between the number of queries that DAAISy takes to learn the model as compared to AIA and the correctness of the learned model. As evident from the results, if the model has not drifted much, DAAISy can serve as a better approach to efficiently learn the updated functionality of the agent with less overhead as compared to AIA. Deciding the amount of drift after which it would make sense to switch to querying the model from scratch is a useful analysis not addressed in this chapter.

4.5 Related Work

White-Box Model Drift Bryce *et al.* (2016) address the problem of learning the updated mental model of a user using particle filtering given prior knowledge about the user's mental model. However, they assume that the entity being modeled can tell the learning system about flaws in the learned model if needed. Eiter *et al.* (2005, 2010) propose a framework for updating action laws depicted in the form of graphs representing the state space. They assume that changes can only happen in effects, and that knowledge about the state space and what effects might change is available beforehand. Our work does not make such assumptions to learn the correct model of the agent's functionalities.

Action Model Learning The problem of learning agent models from observations of its behavior is an active area of research (Gil, 1994; Yang *et al.*, 2007; Cresswell *et al.*, 2009; Zhuo and Kambhampati, 2013; Arora *et al.*, 2018; Aineto *et al.*, 2019). Recent

work addresses active querying to learn the action model of an agent (Rodrigues *et al.*, 2011b; Verma *et al.*, 2021a). However, these methods do not address the problem of reducing the computational cost of differential model assessment, which is crucial in non-stationary settings.

Online action model learning approaches learn the model of an agent while incorporating new observations of the agent behavior (Čertický, 2014; Lamanna *et al.*, 2021a,b). Unlike our approach, they do not handle cases where (i) the new observations are not consistent with the older ones due to changes in the agent's behavior; and/or (ii) there is reduction in functionality of the agent. Lindsay (2021) solve the problem of learning all static predicates in a domain. They start with a correct partial model that captures the dynamic part of the model accurately and generate negative examples by assuming access to all possible positive examples. Our method is different in that it does not make such assumptions and leverages a small set of available observations to infer about increased and reduced functionality of an agent's model.

Model Reconciliation Model reconciliation literature (Chakraborti *et al.*, 2017; Sreedharan *et al.*, 2019, 2021) deals with inferring the differences between the user and the agent models and removing them using explanations. These methods consider white-box known models whereas our approach works with black-box models of the agent.

4.6 Concluding Remarks

We presented a novel method for *differential assessment* of black-box AI systems to learn models of true functionality of agents that have drifted from their previously known functionality. Our approach provides guarantees of correctness w.r.t. obser-

vations. Our evaluation demonstrates that our system, DAAISy, efficiently learns a highly accurate model of agent's functionality issuing a significantly lower number of queries as opposed to relearning from scratch. In the future, we plan to extend the framework to more general classes, stochastic settings, and models. Analyzing and predicting switching points from selective querying in DAAISy to relearning from scratch without compromising the correctness of the learned models is also a promising direction for future work.

In this chapter and the previous one, in addition to the assumption of determinism and full observability, we assumed that (i) the user's vocabulary in terms of which the model is being learned is same as that of the agent's vocabulary, and (ii) the model is learned in terms of low level actions of the agent. We will remove these two assumptions in the next chapter.

Chapter 5

CAPABILITY DISCOVERY

AI systems are rapidly developing to an extent where their users may not understand what they can and cannot do safely. In fact, the limits and capabilities of many AI systems are not always immediately clear even to the experts, as they may use black box policies, e.g., text summarization tools (Paulus *et al.*, 2018), game-playing agents (Greydanus *et al.*, 2018), mobile manipulators (Popov *et al.*, 2017), etc.

In the last two chapters, we have seen two approaches that assesses the agent whose capability names are part of the input. These approaches, along with a several others, are helpful for answering users' specific questions about AI behavior and for assessing their core functionality in terms of primitive executable actions. However, the problem of summarizing an AI agent's broad capabilities for a user is comparatively new.

Ongoing research on the topic focuses on the significant problem of how to answer users' questions about the system's behavior while assuming that the user and AI share a common action vocabulary (Chakraborti *et al.*, 2017; Dhurandhar *et al.*, 2018; Anjomshoae *et al.*, 2019; Barredo Arrieta *et al.*, 2020). Furthermore, most non-experts hesitate to ask questions about new AI tools (Mou and Xu, 2017) and often do not know which questions to ask for assessing the safe limits and capabilities of an AI system. This problem is aggravated in situations where an AI system can carry out planning or sequential decision making. Lack of understanding about the limits of an imperfect system can result in unproductive usage or, in the worst-case, serious accidents (Randazzo, 2018). This, in turn, limits the adoption and productivity of AI systems.

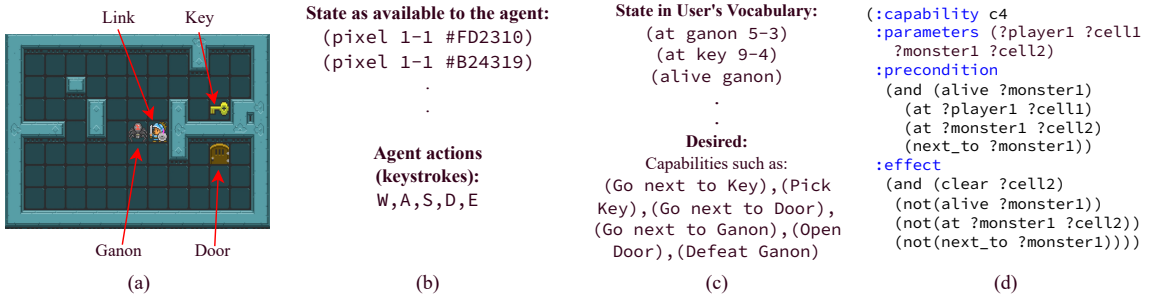


Figure 7. From pixels to interpretable capabilities. (a) A Zelda-like game; (b) States available to the agent and its actions; (c) States represented in user vocabulary, and possible set of desired capabilities; (d) A parameterized capability description learned by our method.

To alleviate these issues, this chapter presents a new approach for discovering from scratch the suite of high-level "capabilities" that an AI system with arbitrary internal planning algorithms/policies can perform. It computes conditions describing the applicability and effects of these capabilities in user-interpretable terms. Starting from a set of user-interpretable state properties, an AI agent, and a simulator that the agent can interact with, our algorithm returns a set of high-level capabilities with their parameterized descriptions. Prior work on the topic addresses complementary problems of deriving symbolic descriptions for pre-defined skills (Konidaris *et al.*, 2018) and of learning users' conceptual vocabularies (Kim *et al.*, 2018; Sreedharan *et al.*, 2022). However, they do not address the problem of *discovering* high-level user-interpretable capabilities that an agent can perform using arbitrary, internal behavior synthesis algorithms (see Sec. 5.4 for a greater discussion).

As a starting point, in this chapter, we assume determinism and full observability on part of the AI system. Since there are no solution approaches for solving the problem even in this foundational setting, our framework can serve as a foundation for solutions to the more general setting in the future.

Running example Consider a game based on "The Legend of Zelda" (Fig. 7)

featuring a protagonist player *Link* who must defeat the antagonist monster *Ganon*, and escape through the door using a key. (Fig. 7)(a) shows the game state as the agent sees it; its primitive actions are keystrokes as shown in (b). These keystrokes do not help convey the agent's capabilities because (i) they are too fine-grained, and (ii) they show the set of actions available to the AI system, although its true capabilities depend on its AI planning and learning algorithms. Fig. 7(c) shows common English terms that a user might understand (called the *user's vocabulary*), and the types of capabilities that they may want to know about. Fig. 7(d) shows a parameterized capability discovered by our method. Intuitively, Fig. 7(d) captures the "defeat Ganon" capability.

This chapter shows how we can discover and describe an agent's capabilities in the form of Fig. 7(d). This capability description can be readily transcribed as "If the *player1* is at *cell1*; the *monster1* is at *cell2*; the *monster1* is *alive* (not defeated); and the *monster1* is next to the *player1*; then the *player1* can act to reach a state where *cell2* is empty; the *monster1* is *not alive* (defeated); the *monster1* is *not at cell2*; and the *player1* is not next to the *monster1*." Our empirical evaluation shows that our system effectively discovers such high-level capabilities; our user study shows that the discovered capabilities help users effectively estimate black-box agent capabilities.

The rest of this chapter is organized as follows. The next section presents a formal framework for capabilities as well as notions of correctness for discovered agent capabilities. Sec. 5.2 describes our main algorithms and their formal properties and Sec. 5.3 presents empirical results and results from user studies. Sec. 5.4 discusses the relationship of the presented methods with prior work. Finally, Sec. 5.5 presents our conclusion and future directions.

5.1 Formal Framework

We model an AI system ("agent" henceforth) as a 3-tuple $\langle S, A, T \rangle$, where S is the state space, A is the set of actions that the agent can execute, $T : S \times A \rightarrow S$ is a deterministic black-box transition function determining the effects of the agent's primitive actions on the environment. For brevity of notation, we use $a(s)$ to represent $T(s, a)$, where $a \in A$, and $s \in S$. Given a goal set $G \subseteq S$, a black-box *deterministic policy* $\Pi : S \rightarrow A$ maps each state to the action that the agent should execute in that state to reach a $g \in G$.

In this chapter, we use "actions" to refer to the core *functionality* of the agent, denoting the agent's decision choices, or primitive actions that the agent could execute (e.g., keystrokes in our running example). In contrast, we use the term "capabilities" to refer to the *high-level behaviors* that the agent can perform using its AI algorithms for behavior synthesis, including planning and learning (e.g., defeating Ganon or picking up the key). Thus, actions refer to the set of choices that a tabular-rasa agent may possess, while capabilities are a result of its agent function (Russell, 1997) and can change as a result of algorithmic updates even as the agent uses the same actions.

5.1.1 Abstraction

We now define the notion of abstraction used in this work. 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 *et al.*, 2016). We refer to $\tilde{S}$ as the set of *high-level* or *abstract* states, and S as the set of *low-level* or *concrete* states. We define abstraction as in (Srivastava *et al.*, 2016):

Definition 18. Let S and $\tilde{S}$ be sets such that $|\tilde{S}| \leq |S|$. An *abstraction* from S to $\tilde{S}$ is defined by a surjective function $f : S \to \tilde{S}$. For any $\tilde{s} \in \tilde{S}$, the concretization function $f^{-1}(\tilde{s}) = \{s \in S : f(s) = \tilde{s}\}$ denotes the set of states represented by the abstract state $\tilde{s}$.

Following this, we use $\tilde{}$ whenever we refer to a state, a predicate, or an action pertaining to the abstract state space.

5.1.2 Capability Descriptions

We express capability descriptions using a STRIPS-like representation (Fikes and Nilsson, 1971; McDermott *et al.*, 1998). In our running example, such a description could indicate that if Link is next to Ganon then Link can defeat it. We now formally define a capability.

Definition 19. Given a set of objects $\tilde{O}$; and a finite set of predicates $\tilde{P} = \{\tilde{p}_1^{k_1}, \ldots, \tilde{p}_n^{k_n}\}$ with arities k_i; a *grounded capability* $\tilde{c}^*$ is defined as a tuple $\langle \mathit{pre}(\tilde{c}^*), \mathit{eff}(\tilde{c}^*) \rangle$ where precondition $\mathit{pre}(\tilde{c}^*)$ and effect $\mathit{eff}(\tilde{c}^*)$ are conjunctions of literals over $\tilde{P}$ and $\tilde{O}$.

We also refer to the tuple $\langle \tilde{c}^*, \mathit{pre}(\tilde{c}^*), \mathit{eff}(\tilde{c}^*) \rangle$ as the *capability description* for a capability $\tilde{c}^*$. Here each atom could be absent, positive, or negative (henceforth referred to as the *mode*) in the precondition and the effect of an action. However, we disallow atoms to be positive (or negative) in both the preconditions and the effects of an action simultaneously to avoid redundancy. Semantics of capabilities are close to those of STRIPS actions, but they address vocabulary disparity: an agent can execute a capability $\tilde{c}^*$ in any concrete state s where $\tilde{s} \models \mathit{pre}(\tilde{c}^*)$; as a result, the

system reaches a concrete state s' (a member of an abstract state $\tilde{s}'$). Atoms that don't appear in $\mathit{eff}(\tilde{c}^*)$ retain their truth values from $\tilde{s}$ in $\tilde{s}'$ while others are set to their modes (positive, negative, or absent) in $\mathit{eff}(\tilde{c}^*)$, i.e., $\forall \ell \in \mathit{eff}(\tilde{c}^*)$, $\tilde{s}' \models \ell$. For brevity, we represent this as $\tilde{s}' = \tilde{c}^*(\tilde{s})$.

We refer to the capabilities defined in Def. 19 as grounded capabilities as they are instantiated with a specific set of objects in $\tilde{O}$. We use * whenever we refer to a grounded capability. We define a lifted form of capabilities as parameterized capabilities.

Definition 20. Given a set of objects $\tilde{O}$; a finite set of predicates $\tilde{P} = \{\tilde{p}_1^{k_1}, \ldots, \tilde{p}_n^{k_n}\}$ with arities k_i; a *parameterized capability* $\tilde{c}$ is defined a 3-tuple $\langle \mathit{args}(\tilde{c}), \mathit{pre}(\tilde{c}), \mathit{eff}(\tilde{c}) \rangle$ where $\mathit{args}(\tilde{c})$ is the set of arguments that can be initialized with a set of objects $\tilde{o} \subseteq \tilde{O}$; and $\mathit{pre}(\tilde{c})$ and $\mathit{eff}(\tilde{c})$ are sets of literals over $\tilde{P}$ and $\mathit{args}(\tilde{c})$.

A set of parameterized capabilities constitutes a parameterized capability model. Formally, a *parameterized capability model* is a tuple $\tilde{M} = \langle \tilde{P}, \tilde{C}, \tilde{O} \rangle$, where $\tilde{P}$ is a finite set of predicates, $\tilde{C}$ is a finite set of parameterized capabilities, and $\tilde{O}$ is the set of objects that can be used to ground the capabilities.

Our objective is to develop a capability discovery algorithm that learns a parameterized capability model of a black-box AI agent using as input (i) the agent, (ii) a compatible simulator using which the agent can simulate its primitive action sequences, and (iii) the user's concept vocabulary, which may be insufficient to express the simulator's state representation. Such assumptions on the agent are common. In fact, the use of third-party simulators for development and testing is the bedrock of most of the research on taskable AI systems today (including game-playing AI, autonomous cars, and factory robots). Providing simulator access for assessment is reasonable as it would allow AI developers to retain freedom and proprietary controls on internal

software while supporting calls for assessment and regulation using approaches such as ours.

Our user studies show the efficacy of this approach using spoken English terms for concepts without an explicit process for vocabulary synchronization. Several threads of ongoing research address the problem of identifying user-specific concept vocabularies (e.g., Kim *et al.* (2018), Sreedharan *et al.* (2022)), and the field of intelligent tutoring systems develops methods for helping users understand a fixed concept vocabulary. These methods can be used to either elicit or impart a vocabulary for a given user and such systems can be used to complement the methods developed in this chapter.

However, since the problem of capability discovery is not well understood even in settings where user-concept definitions are readily available, we focus on capability discovery with a given vocabulary with known definitions and formalize our approach using them. Furthermore, our empirical evaluation and user studies don't place requirements on user concept vocabularies and show the efficacy of this representation. We formalize these concept definitions as follows:

Definition 21. Given a concrete state $s \in S$, a set of objects $\tilde{O}$ and their tuples $\tilde{O}^{\leq d}$ (of dimension at most d, where d is a positive integer), a set of *concepts/predicates* $\tilde{P} = \tilde{p}_1^{k_1}, \ldots, \tilde{p}_n^{k_n}$ with their arities k_i and an associated Boolean evaluation function $\psi_{\tilde{p}_i} : S \times \tilde{P} \times \tilde{O}^{\leq max(k_i)} \rightarrow \{T, F\}$, $j \leq max(k_i)$, we define $s \models_{\psi_{\tilde{p}_i}} \tilde{p}_i(\tilde{o}_1, \ldots, \tilde{o}_j)$ iff $\psi_{\tilde{p}_i}(s, \tilde{p}_i, \tilde{o}_1, \ldots, \tilde{o}_j) = T$. We define *the abstraction* $\tilde{s}_{\tilde{P},\tilde{O}}$ *of a state* $s \in S$ as the set of all literals over $\tilde{P}$ and $\tilde{O}$ that are true in s. $\tilde{S}_{\tilde{P},\tilde{O}}$ denotes the abstract state space $\{\tilde{s}_{\tilde{P},\tilde{O}} : s \in S\}$.

We omit subscripts $\tilde{P}$ and $\tilde{O}$ unless needed for clarity. As mentioned, we assume availability of an evaluation function $\psi_{\tilde{p}}$ associated with each predicate $\tilde{p} \in \tilde{P}$. E.g., for a 3-D Blocksworld simulator with objects a and b, and coordinates x, y, and z,

Algorithm 4: Interactive Capability Model Learning

Input : predicates $\tilde{P}$, agent $\mathcal{A}$
Output : $\tilde{M}$

$E \leftarrow$ generate_execution_traces($\mathcal{A}$)
$\tilde{C}^* \leftarrow$ generate_partial_capability_descriptions(E)
$\tilde{C}' \leftarrow$ parameterize_partial_capabilities($\tilde{C}^*$)
$\tilde{M} \leftarrow$ generate_parameterized_capability_model($\tilde{C}'$)
Set $\tilde{L} \leftarrow \{pre, eff\}$
for each $\langle \tilde{L}, \tilde{C}, \tilde{P} \rangle$ in $\tilde{M}$ **do**
    Generate $\tilde{M}_+, \tilde{M}_-, \tilde{M}_\emptyset$ by setting $\tilde{P}$ in $\tilde{C}$ at $\tilde{L}$ to $+, -, \emptyset$ in $\tilde{M}$
    for *each pair* $\tilde{M}_x, \tilde{M}_y$ *in* $\{\tilde{M}_+, \tilde{M}_-, \tilde{M}_\emptyset\}$ **do**
        $\tilde{q} \leftarrow$ generate_query($\tilde{M}_1, \tilde{M}_2$)
        $\tilde{\varrho} \leftarrow$ generate_waypoints($\tilde{q}$)
        $\varrho \leftarrow$ refine_waypoints($\tilde{\varrho}, \tilde{P}$)
        for i in range$[0, k-1]$ **do**
            $\theta \leftarrow$ ask_agent($\mathcal{A}, \langle s_i, s_{i+1} \rangle$)
            break if $\theta = \perp$
        $\tilde{M} \leftarrow$ consistent_description($i, \tilde{s}_i, \tilde{M}_x, \tilde{M}_y$)
return $\tilde{M}$

"$on(a, b)$ is true exactly for states where $z(a) > z(b)$, $x(a) = x(b)$, and $y(a) = y(b)$." As this example illustrates, such vocabularies can be inaccurate. The abstraction function f (Def. 21) can be modeled as a conjunction of these evaluation functions $\psi_{\tilde{p}}$. We now discuss how we discover capabilities and learn their descriptions.

5.2 Active Capability Discovery

Our overall approach consists of two main phases: (1) discovering candidate capabilities and their partial descriptions from a set of execution traces of the agent's behavior (Sec. 5.2.1); and (2) completing the descriptions of the candidate capabilities discovered in step (1) by asking the agent queries designed to assess under which

conditions it can execute those capabilities and what their effects are (Sec. 5.2.2). The interactive Capability Model Learning (iCaML) algorithm (Alg. 4) performs both these steps using user interpretable predicates $\tilde{P}$ and the agent $\mathcal{A}$ as inputs. We now explain these two phases in detail.

5.2.1 Discovering Candidate Partial Capabilities

In this section we explain the various steps for discovering the candidate capabilities and generating partial description for them.

5.2.1.1 Generating Execution Traces

As a first step, Alg. 4 collects a set of execution traces E from the agent (line 1). An *execution trace* e is a sequence of states of the form $\langle s_0, s_1, \ldots, s_{n-1}, s_n \rangle$, such that $\forall j \in [1, n] \quad \exists a_i \in A \; a_j(s_{j-1}) = s_j$. To obtain the traces $e \in E$, a set of random tasks of the form $\langle s_I, s_G \rangle$, where $s_I, s_G \in S$, are given to the agent $\mathcal{A}$, and the agent is asked to reach s_G from s_I. Intermediate states that the agent goes through form the set of execution traces E.

5.2.1.2 Discovering Candidate Capabilities

To discover candidate capabilities, we abstract the low-level execution traces E in terms of the user's vocabulary (line 2). This abstraction of a low-level execution trace $\langle s_0, s_1, \ldots, s_{n-1}, s_n \rangle$ gives a high-level execution trace $\langle \tilde{s}_0, \tilde{s}_1, \ldots, \tilde{s}_{n-1}, \tilde{s}_n \rangle$. Since we do not assume that the user's vocabulary is precise enough to discern all the states

available to the agent, more than one low-level state in an execution trace may be abstracted to a single high-level abstract state in $\tilde{S}$. Hence for some $j \in [0, n-1]$, it is possible that $\tilde{s}_j = \tilde{s}_{j+1}$. E.g., in Fig. 7(a), the state available to the agent in the simulator expresses pixel-level details of the game (Fig. 7(b)), whereas the user's vocabulary can express it only as an abstract state that represents multiple similar low-level states (Fig. 7(c)). Formally, an *abstract execution trace* is the longest subsequence of $\tilde{s}_1, \ldots, \tilde{s}_n$ such that no two subsequent elements are identical. We remove the repetitions from the high-level execution trace to get the abstract execution trace $\tilde{e} = \langle \tilde{s}_0, \ldots, \tilde{s}_m \rangle$, where $m \leq n$.

We store each transition $\tilde{s}_i, \tilde{s}_{i+1}$ in $\tilde{e}$ as a new grounded candidate capability $\tilde{c}^*_{\tilde{s}_i, \tilde{s}_{i+1}}$.

5.2.1.3 Generating Partial Capability Descriptions

For each candidate capability $\tilde{c}^*_{\tilde{s}_i, \tilde{s}_{i+1}}$, the set of predicates $\tilde{s}_{i+1} \setminus \tilde{s}_i$ is added to the effects of $\tilde{c}^*_{\tilde{s}_i, \tilde{s}_{i+1}}$ in positive form (add effects); whereas the set $\tilde{s}_i \setminus \tilde{s}_{i+1}$ is added to the same candidate capability's effects in negative form (delete effects). As an optimization, in a manner similar to Stern and Juba (2017), we also store that the predicates true in $\tilde{s}_i$ cannot be negative preconditions for this capability, whereas the predicates false in $\tilde{s}_i$ cannot be positive preconditions.

5.2.1.4 Lifting the Partial Capability Descriptions

After line 2 of Alg. 4, we get a set of candidate capabilities with their partial descriptions that are in terms of predicates $\tilde{P}$ instantiated with objects in $\tilde{O}$. For each such grounded partial capability description, the predicates in the preconditions

and effects are sorted in some lexicographic order. The choice of ordering is not important as long as it stays consistent throughout Alg. 4. The objects used in predicate arguments are assigned unique IDs corresponding to this capability in the order of their appearance in ordered predicates. These IDs are then used as variables representing capability parameters. E.g., suppose we have a grounded partial capability description with a precondition: `(alive ganon)`, `(at link cell6)`, `(at ganon cell5)`, `(next_to ganon)`. Traversing the predicates in this order, the objects used in these predicates are given IDs as follows: {`ganon`: 1, `link`: 2, `cell6`: 3, `cell5`: 4}. Note that there is only one assignment per object, hence `ganon` in `(at ganon cell5)` was not given a separate ID. This procedure is extended to effects while assigning new IDs for any unseen objects in the partial capability description. Finally, the parameterized partial capability description is constructed by replacing all occurrences of objects in the partial capability description with variables corresponding to their unique IDs.

5.2.1.5 Combining Candidate Capabilities

Multiple candidate partial capabilities can be combined if their precondition and effect conjunctions are unifiable. E.g., for any capability to match the capability discussed above, it's precondition should be in the form: `(alive ?1)`, `(at ?2 ?3)`, `(at ?1 ?4)`, `(next_to ?1)`. Its effects should also be unifiable in terms of these parameters. The algorithm also keeps track of which grounded partial candidate capabilities map to each parameterized partial capability description. These descriptions are partial as they are generated using limited execution traces and may not capture all the preconditions or effects of a capability. E.g., suppose a capability

adds a literal on its execution. If that literal is already present in the state where the capability was executed, it will not be captured in the effect of the capability's partial description. Hence, we next try to complete the partial capability descriptions. Note that all parameterized partial capability descriptions are collectively used as the parameterized capability model $\tilde{M}$ (line 4).

5.2.2 Completing Partial Capability Descriptions

To complete the partial capability descriptions $\tilde{M}$, Alg. 4 generates queries aimed to gain more information about the conditions under which the capability can be executed and the state properties that become true or false upon its execution. These queries give the agent a sequence of states, called waypoints, to traverse. Based on the agent's ability to traverse them, we derive the precondition and effect of each capability. Alg. 4 iterates through the combinations of predicates and capabilities generated earlier to determine how each predicate will appear in each capability's precondition and effect (line 6). For each combination, it generates a query as follows.

5.2.2.1 Active Query Generation

For each combination of predicate, capability, and precondition (or effect), three possible capability descriptions $M_+, M_-, M_\emptyset$ are possible, one each for the predicate appearing in the precondition (or effect) of the capability in positive, negative, or absent mode, respectively (line 7). As noted when generating partial capability descriptions in Sec. 3.1, some of the models will not be considered if we know that a form is not possible for a particular predicate. The algorithm iteratively picks

two such models M_x, M_y from $M_+, M_-, M_\emptyset$ (line 8) and generates a query $\tilde{q}$ in the form of a state $\tilde{s}_0$ and a capability sequence $\tilde{\pi}$ such that the result of executing the sequence $\tilde{\pi}$ on $\tilde{s}_0$ is different in M_x and M_y (line 9). We use the agent interrogation algorithm (AIA) (Alg. 1), and AIA reduces query generation to a planning problem. The resulting query $\tilde{q}$ is of the form $\langle \tilde{s}_0, \tilde{\pi} \rangle$, asking the model (or an agent) about the length of the plan $\tilde{\pi}$ that it can successfully execute when starting from state $\tilde{s}_0$. Here plan $\tilde{\pi}$ is a sequence of capabilities $\langle \tilde{c}^*_1, \ldots, \tilde{c}^*_n \rangle$ grounded with objects in $\tilde{O}$.

5.2.2.2 Generating Waypoints from Queries

The queries described above cannot be directly posed to an agent, as the plan $\tilde{\pi}$ is in terms of high-level capabilities $\tilde{c}^*_i \in \tilde{C}^*$, which the agent will not be able to comprehend. Additionally, these high-level capabilities cannot be converted directly to low-level actions, as each capability may correspond to a different sequence of low-level actions depending on the state in which it is executed. Hence, we pose the queries to the agent in the form of high-level state transitions induced by the capabilities in the query's capability sequence.

To accomplish this, Alg. 4 converts the query $\tilde{q}$ to a sequence of waypoints $\tilde{\varrho} = \langle \tilde{s}_0, \ldots, \tilde{s}_n \rangle$. Starting from the initial state $\tilde{s}_0$, these are generated by applying the capability $\tilde{c}^*_i$, for $i \in [1, n]$, in the state $\tilde{s}_{i-1}$ according to the partial capability description of $\tilde{c}^*_i$. Note that the waypoints $\tilde{\varrho}$ cannot be presented to the agent as the agent may not know the high-level vocabulary. Hence these high-level waypoints must be refined into the low-level waypoints $\varrho = \langle s_0, \ldots, s_n \rangle$ (with each s_i similar to state shown in Fig. 7(b)) that agent understands.

Alg. 4 first converts the high-level waypoints $\tilde{\varrho}$ to a sequence of low-level waypoints

$\varrho = \langle s_0, \ldots, s_n \rangle$ using the predicate definitions (line 11). Then each consecutive pair of states $\langle s_i, s_{i+1} \rangle$ is given sequentially to the agent as a *state reachability query* asking if it can reach from state s_i to s_{i+1} using its internal black-box policy (line 13).

5.2.2.3 Updating Partial Models based on Agent Responses

Using its internal planning mechanism and the simulator, the agent attempts to reach from state s_i to s_{i+1}. If it succeeds, the response to the query is recorded as true; if it fails, the response is recorded as false. The algorithm keeps track of the waypoints that were successfully traversed. Based on the waypoint pairs that the agent was able to traverse, we discard the capability descriptions among M_x and M_y that are not consistent with the agent's response (line 15).

E.g., suppose the algorithm is trying to determine how the predicate `(alive ?monster1)` should appear in the precondition of capability $c4$ shown in Fig. 7(d). Now the two possible capability descriptions M_1 and M_2 that Alg. 4 is considering in line 6 are M_+ and M_-, corresponding to `(alive ?monster1)` being in $c4$'s precondition in positive and negative form, respectively. The algorithm will generate query with its corresponding waypoints $\tilde{\varrho} = \langle \tilde{s}_0, \tilde{s}_1 \rangle$, where $\tilde{s}_0$ will correspond to the state shown in Fig 7(a), and $\tilde{s}_1$ will be $\tilde{s}_0$ without Ganon. Now the agent uses its own internal mechanism to try to reach $\tilde{s}_1$ from $\tilde{s}_0$ and succeeds. Since this is not possible according to M_-, M_- will be discarded.

We now define and prove the theoretical properties of iCaML algorithm. To do this, we use two key properties of Alg. 1 relevant to this work: (1) if there exists a distinguishing query for two models then it will be generated (Thm. 1); and (2) the algorithm will not discard any model that is consistent with the agent (Thm. 2).

5.2.3 Formal Analysis

Alg. 4 has two main desirable properties: (1) the partial capability model (that is maintained as $\tilde{M}$) is always maximally consistent, i.e, adding any more literals into it would be unsupported by the execution traces that we obtain; and (2) the final parameterized capability is complete in the limit of infinite execution traces given to Alg. 4. We first define these concepts and then formalize the results under Thm. 1 and Thm. 2.

Definition 22. Let $e = \langle s_0, \ldots, s_n \rangle$ be an execution trace with a corresponding abstract execution trace $\tilde{e} = \langle \tilde{s}_0, \ldots, \tilde{s}_m \rangle$, where $m \leq n$. A ***parameterized capability model*** $\tilde{M} = \langle \tilde{P}, \tilde{C}, \tilde{O} \rangle$ is consistent with E iff $\forall i \in [0, m-1]\ \exists \tilde{c}^* \in \tilde{C}^*\ \tilde{s}_{i+1} = \tilde{c}^*(\tilde{s}_i)$, where $\tilde{C}^*$ is a set of grounded capabilities that can be generated by instantiating the parameters of capabilities $\tilde{c} \in \tilde{C}$ with objects in $\tilde{O}$.

We extend this terminology to say that a capability model is consistent with a set of execution traces E iff it is consistent with every trace in E. This notion of consistency captures completeness as a parameterized capability model $\tilde{M}$ that is consistent with a set of execution traces E, is also complete w.r.t. E. We next define a stronger notion of completeness that our algorithm provides in the form of maximal consistency. This helps to assess the succinctness of a capability model with a set of execution traces E.

Definition 23. Let E be a set of execution traces, and Λ be the set of possible agents that can generate all execution traces in E. A *parameterized capability model* $\tilde{M} = \langle \tilde{P}, \tilde{C}, \tilde{O} \rangle$ *is maximally consistent with a set of execution traces* E iff (i) $\tilde{M}$ is consistent with E, and (ii) adding any predicate as positive or negative precondition

or effect of a capability in $\tilde{M}$ makes it inconsistent with at least one execution trace that can be generated by at least one agent $\mathcal{A}^{\#} \in \Lambda$.

An abstraction satisfies ***local connectivity*** iff $\forall \tilde{s}\ \forall s_i, s_j \in f^{-1}(\tilde{s})$ there exists a sequence of primitive actions $\langle a_i, \ldots, a_n \rangle$ such that $a_n(a_{n-1} \ldots (a_1(s_i)) \ldots) = s_j$. We use this to show that the parameterized capability model learned by Alg. 4 is maximally consistent.

Theorem 5. Let $\mathcal{A} = \langle S, A, T \rangle$ be an agent operating in a deterministic, fully observable, and stationary environment with a state space S using a set of primitive actions A. Given an input vocabulary $\tilde{P}$, and the set of execution traces E generated by $\mathcal{A}$, if local connectivity holds, then the capability model $\tilde{M}$ maintained by Alg. 1 is *consistent* with the set of execution traces E.

Proof. We show that given the set of all execution traces E, the parameterized capability model $\tilde{M}$ maintained by Alg. 4 is consistent with E, i.e., for every high-level transition $\tilde{s}, \tilde{s}'$ corresponding to a transition in E, there exists a capability $\tilde{c}$ which has a grounding $\tilde{c}^*$ such that $\tilde{c}^*(s) = \tilde{s}'$. We prove this by contradiction. The partial capability model $\tilde{M}$ is initially generated using observed transitions $\tilde{s}, \tilde{s}'$ corresponding to the transitions in E as grounded capabilities $\tilde{c}^*_{\tilde{s},\tilde{s}'}$ (lines 2 to 4 in Alg. 4). So the model $\tilde{M}$ is consistent with the set of traces to start with. At each step, Alg. 4 adds a new literal l to a capability $\tilde{c}$ in $\tilde{M}$ such that adding l keeps $\tilde{M}$ consistent with the agent $\mathcal{A}$ (Thm. 2 from VMS21). Now consider that adding l to $\tilde{M}$ makes it inconsistent with an execution trace in E, i.e., there must exist a transition $\tilde{s}_1, \tilde{s}_2$ such that no capability $\tilde{c}^* \in \tilde{C}^*$ corresponds to it.

Consider the version of $\tilde{c}_1$ corresponding to $c^*_{\tilde{s}_1,\tilde{s}_2}$ that was modified by Alg. 4. We show that modifications inconsistent with this transition are not possible under the

assumption that the agent's capabilities can be expressed using the input vocabulary.
Case 1: Suppose Alg. 4 added a literal l in the precondition of $\tilde{c}_1$ that was not true in $\tilde{s}_1$. Thm. 2 in VMS21 implies that absent and negated forms of l were inconsistent with executions of $\tilde{c}_1$ using the same agent that generated E. In other words, the agent sometimes requires l as a precondition to execute $\tilde{c}_1$, even though l was not a part of $\tilde{s}_1$. This contradicts the assumption that $\tilde{c}_1$ is expressible using the input vocabulary in the form of Def. 3.
Case 2: Suppose Alg. 4 added a literal l in the effect of $\tilde{c}_1$ that was not present in $\tilde{s}_2$. This implies that the negation and absence of l in the result of $\tilde{c}_1$ were inconsistent with the agent's execution of $\tilde{c}_1$ in query-responses generated by Alg. 4. A similar contradiction about the assumption of expressiveness follows.

Hence, the capability model $\tilde{M}$ maintained by Alg. 1 is *consistent* with the set of execution traces E. □

Theorem 6. Let $\mathcal{A} = \langle S, A, T \rangle$ be an agent operating in a deterministic, fully observable, and stationary environment with a state-space S using a set of primitive actions A. Given an input vocabulary $\tilde{P}$, and the set of execution traces E generated by $\mathcal{A}$, if local connectivity holds, then the capability model $\tilde{M}$ returned by Alg. 1 is *maximally consistent* with the set of execution traces E.

Proof. We will prove the two conditions for maximal consistency separately. The first condition is that the model $\tilde{M}$ returned by Alg. 4 is consistent with E follows directly from Thm. 1. Since the model maintained by Alg. 4 at each step is consistent with E, hence the same model returned after the last iteration is also consistent with E.

Next, we show that adding any predicate as a positive or negative precondition or effect of a capability in $\tilde{M}$ returned by Alg. 4 makes it inconsistent with at least one execution trace that can be generated by at least one agent $\mathcal{A}^{\#} \in \Lambda$, where Λ is the

set of possible agents that can generate all execution traces in E. We prove this by contradiction. Note that a literal is not added by Alg. 4 to an action's precondition (or effect) only if (1) in the observed traces, it was not present in the state where (immediately after) that action was executed; or (2) adding it in the precondition (or effect) of an action resulted in a response to a query that was inconsistent with that of the agent. Also, note that a predicate corresponding to a literal is always added to the model in some form in each precondition (or effect). Suppose a literal l that was not added by Alg. 4 is added to $\tilde{M}$ in precondition (or effect) of a capability $\tilde{c}$ without making it inconsistent with the agent. Since a predicate p corresponding to this literal l is already present in $\tilde{c}$, this implies that the form of the predicate p added by Alg. 4 is incorrect. But this is not possible as shown by Thm. 1 and Thm. 2 of VMS21. Hence this is not possible and adding an additional literal in any form to an action's precondition or effect would make it inconsistent with the agent. This means that it also makes the model inconsistent with at least one agent $\mathcal{A}^{\#} \in \Lambda$. □

Next, we formalize the notion of downward refinability, that the discovered capabilities are indeed within the agent's scope. In this work, refinability is similar to the notion of forall-exists abstractions (Srivastava *et al.*, 2016) for deterministic systems. Recall the notion of abstraction functions (Def. 21).

Definition 24. Let $\tilde{M} = \langle \tilde{P}, \tilde{C}, \tilde{O} \rangle$ be a capability model with $\tilde{S}$, the induced state space over $\tilde{P}, \tilde{O}$ using an abstraction function f, for an agent $\mathcal{A} = \langle S, A, T \rangle$; and $\tilde{C}^*$ be a set of grounded capabilities that can be generated by instantiating the arguments of capabilities $\tilde{c} \in \tilde{C}$ with objects in $\tilde{O}$. A capability $\tilde{c}^* \in \tilde{C}^*$ is ***realizable w.r.t.*** $\mathcal{A}$ iff $\forall \tilde{s} \in \tilde{S}$, if $\tilde{s} \models pre(\tilde{c}^*)$ then $\forall s \in f^{-1}(\tilde{s}) \quad \exists a_1, \ldots, a_n \in A : a_n(a_{n-1} \ldots (a_1(s)) \ldots) \in \tilde{c}^*(\tilde{s})$. The model M is *realizable w.r.t.* $\mathcal{A}$ iff all capabilities $\tilde{c}^* \in \tilde{C}^*$ are realizable.

In these terms, discovered capabilities are more likely to be useful if they are accurate in the sense that they are consistent with execution traces and realizable, i.e., true representations of what the agent can do. Realizability captures the soundness of the model wrt. the execution of the capabilities. We now show that the parameterized capability model that we learn is realizable.

Theorem 7. Let $\tilde{P}$ be a set of predicates $\tilde{P}$, $\mathcal{A} = \langle S, A, T \rangle$ be an agent with a deterministic transition system T. If a high-level model is expressible deterministically using the predicates $\tilde{P}$, and local connectivity is ensured, then the parameterized capability model $\tilde{M}$ learned by Alg. 4 is *realizable*.

Proof. We will prove that for all capabilities in $\tilde{C}$ learned as part of the parameterized capability model $\tilde{M}$, for all groundings $\tilde{C}^*$, if the capability is executed in an abstract state $\tilde{s}$ such that $\tilde{s} \models pre(\tilde{c}^*)$ then there exists a sequence of low-level states that the agent can traverse to reach a state $\tilde{s}' \in \tilde{c}^*(\tilde{s})$.

We prove this by cases. Consider a capability $\tilde{c} \in \tilde{C}$ whose description is learned using Alg. 4. Using Thm. 1, the precondition and effect of $\tilde{c}$ will be consistent with E generated by the agent. Now consider a grounded capability $\tilde{c}^*$ corresponding to the capability $\tilde{c}$. There are only two cases possible: (1) either $\tilde{c}^*$ appeared in the observed traces or was executed successfully by the agent in response to one of the queries posed to the agent; or (2) it was not present in either. We prove each case separately.
Case 1: There exists a set of low-level states s and s' such that $\tilde{c}^*(\tilde{s}) = \tilde{s}'$, where $\tilde{s} = f(s)$ and $\tilde{s}' = f(s')$. Now due to local connectivity, all states in $f^{-1}(s)$ are connected with each other and same is true for all states in $f^{-1}(s')$. Hence the agent can traverse from any state in $f^{-1}(s)$ to any state in $f^{-1}(s')$ on executing the capability $\tilde{c}^*$. This makes the capability $\tilde{c}^*$ realizable.
Case 2: Since $\tilde{c}^*$ was not observed directly and the only way capabilities are added

to $\tilde{M}$ is if they are lifted forms of capabilities identified from observation traces E, $\tilde{c}^*$ must be a grounding of the lifted form $\tilde{c}_1$ of a capability $\tilde{c}_1^*$ that is of the type considered in case 1. Alg. 4 constructs precondition and effect of $\tilde{c}_1$ while ensuring consistency with query responses and observations under the assumption that the capability model is expressible as in Def. 3. When this assumption holds, the effect or precondition of a capability can only depend on the vocabulary of available predicates, which are considered exhaustively (hierarchically) by Alg. 4. This implies that there must be a path from a concrete state s in the grounding corresponding to $\tilde{c}_1$'s precondition to a concrete state s' that satisfies the effects of grounding of $\tilde{c}_1$'s effects. By local connectivity, this extends to all concrete states in the same abstract state as $\tilde{s}$ corresponding to s.

Hence if a high-level model is expressible deterministically using the predicates $\tilde{P}$, and local connectivity is ensured, then the parameterized capability model $\tilde{M}$ learned by Alg. 4 is *realizable*. □

Note that here expressibility of a high-level model refers to the class of models of the form defined in Def. 3. Together, the notions of maximal consistency and realizability establish the completeness and soundness of our approach wrt a set of execution traces E. Note that this approach will also work when we have access to a stream of execution traces E being collected at random, independent of our active querying mechanism. We next show that in the limit of infinite randomly generated execution traces, our approach will capture all possible agent capabilities with probability 1. Here, capturing all possible agent capabilities in a learned model $\tilde{M} = \langle \tilde{P}, \tilde{C}, \tilde{O} \rangle$ means that if the agent can go from $\tilde{s}_i$ to $\tilde{s}_j$, then one of the capabilities in $\tilde{C}$ will be instantiable to result in $\tilde{s}_j$ when executed from $\tilde{s}_i$.

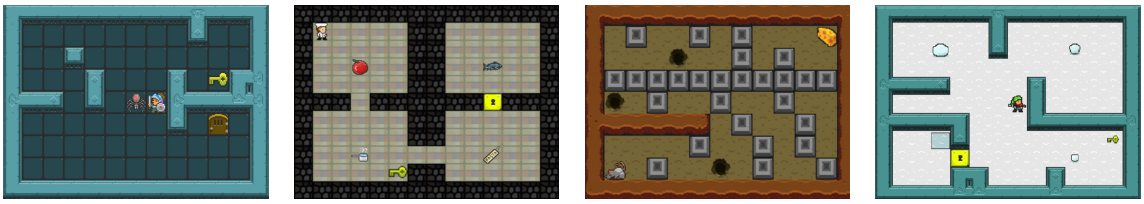
Theorem 8. Let $\tilde{P}$ be a set of predicates, $\mathcal{A} = \langle S, A, T \rangle$ be an agent with a deterministic transition system T. Suppose random samples of agent behavior in the form of execution traces E are coming from a distribution that assigns non-zero probability to at least one transition corresponding to each ground capability ($\tilde{c}^*_{\tilde{s}_i,\tilde{s}_j}$, $\tilde{s}_i, \tilde{s}_j \subseteq \tilde{P}$). If a high-level model is expressible deterministically using the predicates $\tilde{P}$ and local connectivity holds, then in the limit of infinite execution traces E, the probability of discovering all capabilities $\tilde{c} \in \tilde{C}$ expressible using the predicates $\tilde{P}$ is 1.

Proof. Consider every possible abstract transition that the agent can make. There are finite (let's consider L) such transitions possible given the predicate vocabulary $\tilde{P}$ and a fixed set of objects $\tilde{O}$. Now we are getting random execution traces E from a distribution that assigns non-zero probability to at least one transition corresponding to each ground capability ($\tilde{c}^*_{\tilde{s}_i,\tilde{s}_j}$). This means that the probability of not observing this finite set of cardinality L will reduce with each successive collection of L execution traces. Hence we will eventually observe at least one transition corresponding to each ground capability ($\tilde{c}^*_{\tilde{s}_i,\tilde{s}_j}$). Then as shown in Thm. 1, we will discover the capability $\tilde{c}$ corresponding to the ground transition $\tilde{c}^*_{\tilde{s}_i,\tilde{s}_j}$ with probability 1. □

5.3 Empirical Evaluation

We implemented Alg. 4 in Python to empirically verify its effectiveness. [6] To show that our approach can work with different internal agent implementations, we evaluated Alg. 4 with two broad categories of input test agents: *Policy agents* can use (possibly learned) black-box policies to plan and to respond to state reachability queries. We used policy agents with hand-coded policies for this evaluation. *Search*

[6] Code: https://github.com/AAIR-lab/iCaML



(a) (b) (c) (d)

Figure 8. GVGAI's domains; (a) Zelda, (b) Cook-Me-Pasta, (c) Escape, and (d) Snowman.

agents respond to the state reachability queries using arbitrary search algorithms. We used search agents that use A* search (Hart *et al.*, 1968). We now describe the setup of our experiments used for evaluation.

5.3.1 Experimental Setup

Our test agents use the General Video Game Artificial Intelligence framework (Perez-Liebana *et al.*, 2016, 2019). Domains in GVGAI are two-dimensional ATARI-like games defined using the Video Game Description Language PyVGDL (Schaul, 2013). We performed experiments on four such game domains – Zelda, Cook-Me-Pasta, Escape, and Snowman (Fig. 8). All these domains require the agent to navigate in a grid-based environment and complete a set of tasks (in some partial order) to complete the game. Since the complete list of an agent's capabilities may be irrelevant to a user's current needs, w.l.o.g, our implementation supports an input including sets of formulas representing the properties that may be of interest to the user. This set can be the set of all grounded predicates in the user's concept vocabulary. We also consider object types to be a subset of the unary predicates in the vocabulary and assume that each object has exactly one type. These types are

used and discovered in capability like any other predicate. In addition, they are used in creating parameterized capability parameters as shown in Fig. 7(d). More details about the four GVGAIdomains, and the associated user vocabulary for each domain is described below.

5.3.2 Domains and their Semantics

We now see the description of the four GVGAI game domains used in this evaluation and the semantics of the user interpretable predicates in these domains. Note that information like the orientation of the agent (player) in each of these domains is not captured by any of the predicates. This information is important for low-level policies as certain actions can only be executed in certain orientations.

Zelda The Zelda-like domain, as shown in Fig. 7a, consists of a key, a door that opens using that key, the antagonist player *Link*, and the protagonist monster *Ganon*. To win the game, Link must defeat Ganon, and then should use the key to open the door to escape. Link can move one cell at a time in the direction it is facing. If Link moves into the cell adjacent to the key, Link picks up the key by executing the keystroke `E` (special keystroke). The same keystroke is used to Defeat Ganon when Link is facing Ganon and is in a cell adjacent to Ganon, and to escape when Link is in a cell adjacent to the door and facing it. The user vocabulary for this domain is shown in Tab. 7.

Cook-Me-Pasta The Cook-Me-Pasta domain, as shown in Fig. 8b, consists of raw pasta, sauce, boiling water, tuna (fish), lock, and key. The objective is to cook tuna pasta using a three-step process. First, the pasta is cooked by adding boiling water to the raw pasta, this can be done by pressing `E` while holding both the ingredients.

Predicate	Meaning
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if Link has the key.
`escaped()`	True if Link has escaped (game is over).
`alive(?m)`	True if Ganon is still alive
`next_to(?ob2)`	True if Link is in a cell adjacent to ganon, door, or key.

Table 7. Predicates in the user vocabulary for Zelda

Similarly, tuna is cooked by mixing sauce and tuna. Finally, the cooked pasta and the cooked tuna are to be mixed together. One or more of the ingredients can be locked in a room which must be opened using a key. The user vocabulary for this domain is shown in Tab. 8.

Predicate	Meaning
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if the player has the key
`pasta_cooked()`	True if the pasta is cooked
`is_door(?loc)`	True if the location `?loc` has a door

Table 8. Predicates in the user vocabulary for Cook-Me-Pasta.

Escape The Escape domain, as shown in Fig. 8c, consists of movable blocks, fixed holes, and cheese. The blocks can be pushed into the holes to clear out a path. The game is finished when the player reaches the location with cheese. The user vocabulary for this domain is shown in Tab. 9.

Snowman The Snowman domain, as shown in Fig. 8d, consists of three pieces of a snowman: the top, middle, and bottom piece; a key that can be used to unlock a

Predicate	Meaning
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`is_hole(?loc)`	True if the location `?loc` has a hole
`is_goal(?loc)`	True if the location `?loc` is the goal location
`is_block(?loc)`	True if the location `?loc` has a movable block

Table 9. Predicates in the user vocabulary for Escape.

door (like other domains), and the goal cell. The objective of the game is to assemble the snowman in the goal location in order, constrained by the player being able to hold only one piece at any given time. The user vocabulary for this domain is shown in Tab. 10.

Predicate	Meaning
`at(?ob ?loc)`	True if an object `?ob` is at location `?loc`
`wall(?loc)`	True if there is a wall at location `?loc`
`clear(?loc)`	True if location `?loc` is empty, i.e., it has no object, wall, or player
`has_key()`	True if the player has the key
`player_has(?o)`	True if the player has object `?o`
`is_goal(?loc)`	True if the location `?loc` is the goal location
`placed(?part)`	True if part `?part` is placed at the goal location.
`is_door(?loc)`	True if the location `?loc` has a door

Table 10. Predicates in the user vocabulary for Snowman.

5.3.3 Evaluation Strategy

For each domain, and for each grid size in that domain, we create a random game instance with the goal of achieving one of the user's specified properties of interest. To generate these instances, the number of obstacles in all domains, except Escape,

is set to 20% of the total cells in the grid, whereas all other objects are generated randomly. We use the solution to that instance to generate the execution trace that is used in lines 1-2 of Alg. 4. These solutions are not always optimal. All experiments are run on 5.0 GHz Intel i9 CPUs with 64 GB RAM running Ubuntu 18.04.

As shown in Sec. 5.2.3, Alg. 4 is guaranteed to compute capability descriptions that are correct in the sense that they are consistent with the execution traces, and refinable and executable with respect to the true capabilities of the agent. We now present the main conclusions of our empirical analysis.

We evaluated our algorithm's performance along two aspects; (i) how the performance of our approach changes with respect to the size of the problem; and (ii) how its performance differs for search-based vs policy-based agents.

5.3.4 Empirical Results

5.3.4.1 Scalability Analysis

We increase the size of each domain to analyze its effect on the performance of the search and policy agents. Fig. 9 shows the graphs for the experimental runs on the four domains. In all four domains, for both kinds of agents, *the number of queries increases as we increase the grid size.* The increasing number of queries is an expected behavior and this is also clear in approaches that use passive observations of agent behavior (Yang *et al.*, 2007; Aineto *et al.*, 2019).

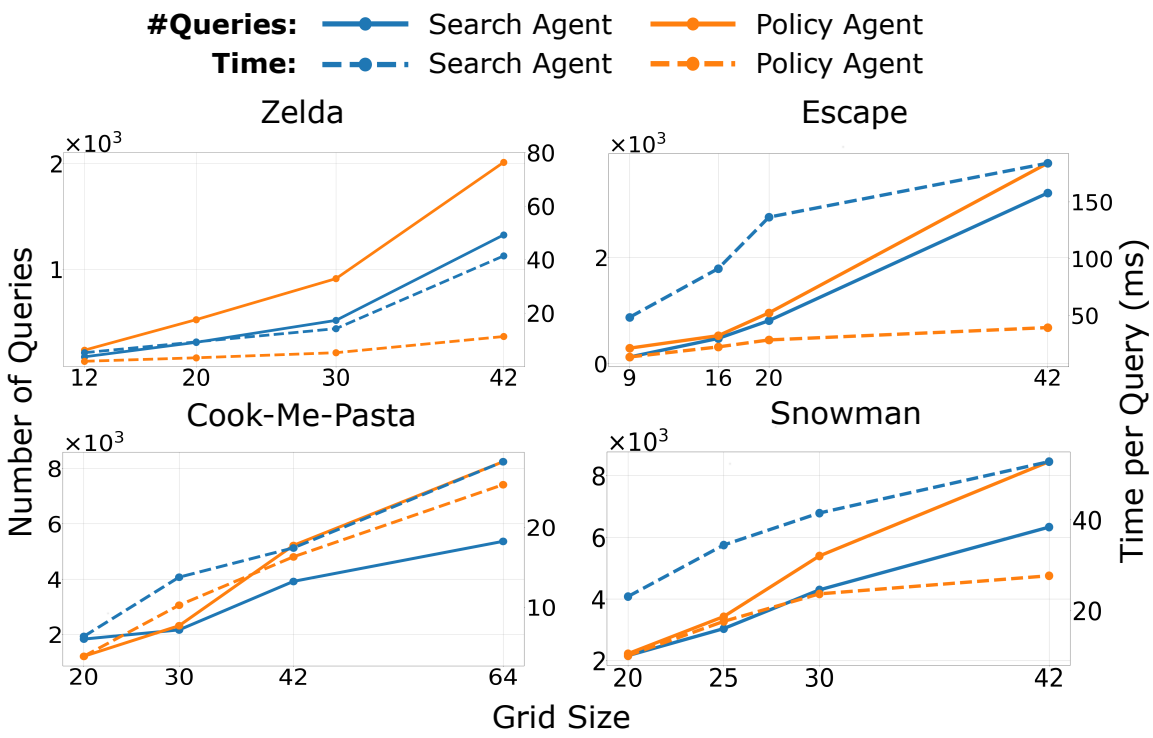


Figure 9. Performance comparison of search-based agents and policy-based agents in terms of the number of queries asked and time taken per query when increasing the grid size (number of cells in the grid) in the four GVGAI domains.

5.3.4.2 Agent Type Analysis

The *number of queries required by the policy agent is higher than that of the search agent* in almost all cases. This is because a large number of state reachability queries fail on the policy agent as the sequence of waypoints in these queries does not always align with the policy of the agent. However, *the time per query is lesser for the policy agents* as they can answer the state reachability queries by following their policy, whereas the search agents perform an exhaustive search of the state space for every such query.

5.3.5 User Study

We conducted a user study to evaluate the utility of the capability descriptions discovered and computed by Alg. 4. Intuitively, our notion of interpretability matches that of common English and its use in AI literature, e.g., as enunciated by Doshi-Velez and Kim (2018): *"the ability to explain or to present in understandable terms to a human."* We evaluate this through the following two operational hypothesis:

H1. The user can effectively summarize the learned capability descriptions.

H2. The discovered capabilities make it easier for users to analyze and predict the outcome of the agent's possible behaviors.

We performed the following study to evaluate H1:

Capability Summarization Study This study evaluates the interpretability of the discovered capability descriptions. The user is explained the rules of the Zelda-like game described earlier (shown in Fig. 26), and then presented with a text description of the six learned capabilities. Finally, as shown in Fig. 27, the user is asked to choose a short summarization for each description, out of the eight possible summarizations that we provide.

We designed the following study to evaluate H2.

Behavior Analysis Study This study compares the predictability and analyzability of agent behavior in terms of the agent's low-level actions and high-level capabilities. Each user is explained the rules of a Zelda-like game. One group of users – called the *primitive action group* – are presented with text descriptions of the agent's primitive actions, while the users in the other group – called the *capability group* – are presented with a text description of the six learned capabilities. The capability group users are asked to choose a short summarization for each capability description, out of the eight

possible summarizations that we provide, whereas the primitive action group users are asked to choose a short summarization for each primitive action description, out of the five possible summarizations that we provide. Then each user is given the same 5 questions in order. Each question contains two game-state images; start and end state. The user is asked what sequence of actions or capabilities that the agent should execute to reach the end state from the start state. Each question has 5 possible options for the user to choose from, and these options differ depending on their group. We then collect the data about the accuracy of the answers, and the time taken to answer each question.

5.3.5.1 Study Design

108 participants were recruited from Amazon Mechanical Turk and randomly divided into two groups of 54 each. Each user was provided with a survey on Qualtrics (Qualtrics, 2005) that explained the rules of GVGAI's Zelda game. We used screeners (Kennedy *et al.*, 2020; Arndt *et al.*, 2021) to ensure quality of the data collected, and discarded 23 responses. The results are based on the responses of 41 and 43 users in the primitive action and capability group, respectively. Additional details of the study are available in Appendix A.

5.3.5.2 Results

Capability Summarization Study There were a total of 54 participants in the capability group out of whom 43 got the sanity check question right. The results of the capability summarization study (Tab. 11) for these 43 participants demonstrate

	S1	S2	S3	S4	S5	S6	S7	S8
C1	1.0	0	0	0	0	0	0	0
C2	0	1.0	0	0	0	0	0	0
C3	0	0	0.91	0	0	0	0.09	0
C4	0	0	0	1.0	0	0	0	0
C5	0	0	0	0	0.84	0	0	0.16
C6	0	0	0	0	0	1.00	0	0

Table 11. Accuracy of capability summarization study for the Zelda-like game. An element in row Ci and column Sj represents the fraction of instances when capability Ci was summarized as Sj by the study participants. Correct summarization of Ci is Si (in green). C1,S1: *Go next to Ganon*; C2,S2: *Go next to Key*; C3,S3: *Go next to Door*; C4,S4: *Defeat Ganon*; C5,S5: *Pick Key*; C6,S6: *Open Door*; S7: *Go next to Wall*; S8: *Break Key*.

that the users are able to summarize the descriptions almost uniformly accurately except for C3 and C5. This verifies H1 that the users can effectively summarize the learned capability descriptions.

Behavior Analysis Study The results of the behavior analysis study are shown in (Fig. 10) To evaluate the statistical significance (p-value) of the difference in the mean of the time taken by the two groups, we used Student's t-test (Student, 1908). The results indicate that the test results were statistically significant with p-values less than 0.05 for all five questions. Also, the users took less time to answer questions and they got more responses correct when using the capabilities as compared to using primitive actions. This validates H2 that the discovered capabilities made it easier for the users to analyze and predict the agent's behavior correctly.

5.4 Related Work

High-level skills from input options Given a set of options encoding skills as

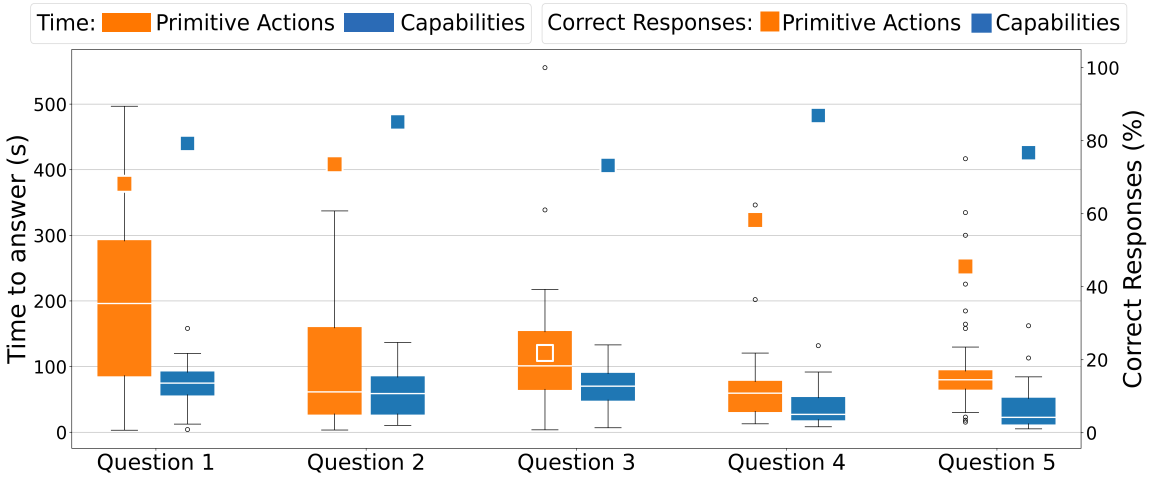


Figure 10. Data from behavior analysis shows that using computed capability descriptions took lesser time and yielded more accurate results. See Sec. 5.3.5 for details.

input, Konidaris *et al.* (2018) and James *et al.* (2020) propose methods for learning high-level propositional models of options representing various "skills." They assume access to predefined options and learn the high-level symbols that describe those options at the high-level. While they use options or skills as inputs to learn models defining when those skills will be useful in terms of auto-generated symbols (for which explanatory semantics could be derived in a post-hoc fashion), our approach uses user-provided interpretable concepts as apriori inputs to learn agent capabilities: high-level actions as well as their interpretable descriptions in terms of the input vocabulary.

Learning symbolic models using physics simulators Multiple approaches learn different kinds of symbolic models of the functionality of ATARI or physics-based simulators using methods like conjunctions of binary input features (Kansky *et al.*, 2017), graph neural networks (Battaglia *et al.*, 2016; Cranmer *et al.*, 2020), CNNs (Agrawal *et al.*, 2016; Fragkiadaki *et al.*, 2016), etc. Some methods create interpretable descriptions of reinforcement learning policies using trees (Liu *et al.*,

2018) or specialized programming languages (Verma *et al.*, 2018). These approaches solve the orthogonal problem of learning the functionality of an agent that could help a user understand how an agent would solve a problem, whereas we focus on learning capabilities of the agent that could help a user understand and answer what type of problems it could solve.

Action model learning The planning community has also worked on learning STRIPS-like action models of agent functionality from observations of its behavior (Gil, 1994; Yang *et al.*, 2007; Cresswell *et al.*, 2009; Zhuo and Kambhampati, 2013; Stern and Juba, 2017; Aineto *et al.*, 2019; Bonet and Geffner, 2020). Jiménez *et al.* (2012) and Arora *et al.* (2018) present a comprehensive review of such approaches. These methods work with broad assumptions that the agent model is internally expressed in the same vocabulary as the user's (Gil, 1994; Weber *et al.*, 2011; Juba *et al.*, 2021), or at a similar level of abstraction (Mehta *et al.*, 2011; Verma *et al.*, 2021a; Nayyar *et al.*, 2022). Additionally, such methods have as input a given set of predicates in terms of which they learn the functionality descriptions of the agent.

High-level actions Works like Madumal *et al.* (2020) explain an agent's policy in terms of high-level actions but they assume that high-level actions are a part of the input whereas our approach discovers these actions. There is an orthogonal thread of research on using high-level actions in AI planning as tasks, and learning low-level policies for each of those tasks (Yang *et al.*, 2018; Lyu *et al.*, 2019; Illanes *et al.*, 2020; Kokel *et al.*, 2021). These works assume the high-level actions as input and learn the corresponding low-level policies.

As compared to the above two classes of methods, our work focuses on solving the harder problem of discovering the capabilities of the agent behavior resulting from its

planning/learning algorithms and learning the descriptions of these capabilities.

5.5 Concluding Remarks

We presented a novel approach for learning the capability description of an AI system in terms of user-interpretable concepts by combining information from passive execution traces and active query answering. Our approach works for settings where the user's conceptual vocabulary is imprecise and cannot directly express the agent's capabilities. Our empirical analysis showed that for the agents that internally use black-box deterministic policies, or search techniques, we can successfully discover the capabilities and their descriptions. Extending this approach for partially observable settings and relaxing the various assumptions we made are some of the promising future directions for this work. Extending this work to stochastic settings is currently in progress, with preliminary results available in Verma *et al.* (2023b).

In the next chapter, we see how to extend the query-response interface to work with agents in stochastic settings.

Chapter 6

LEARNING PROBABILISTIC MODELS

It is essential for users to understand what their AI systems can and can't do in order to use them safely. However, the problem of enabling users to assess AI systems with sequential decision-making (SDM) capabilities is relatively understudied. This chapter presents a new approach for modeling the capabilities of black-box AI systems that can plan and act, along with the possible effects and requirements for executing those capabilities in *stochastic settings*. This is in contrast to the previous chapters that dealt with only the deterministic settings. In this chapter, we present an active-learning approach that can effectively interact with a black-box SDM system and learn an interpretable probabilistic model describing its capabilities. Theoretical analysis of the approach identifies the conditions under which the learning process is guaranteed to converge to the correct model of the agent; empirical evaluations on different agents and simulated scenarios show that this approach is few-shot generalizable and can effectively describe the capabilities of arbitrary black-box SDM agents in a sample-efficient manner.

6.1 Overview

AI systems are becoming increasingly complex, and it is becoming difficult even for AI experts to ascertain the limits and capabilities of such systems, as they often use black-box policies for their decision-making process (Popov *et al.*, 2017; Greydanus *et al.*, 2018). E.g., consider an elderly couple with a household robot that learns and

adapts to their specific household. How would they determine what it can do, what effects their commands would have, and under what conditions? Although we are making steady progress on learning for sequential decision-making (SDM), the problem of enabling users to understand the limits and capabilities of their SDM systems is largely unaddressed. Moreover, as the example above illustrates, the absence of reliable approaches for user-driven capability assessment of AI systems limits their inclusivity and real-world deployability.

This chapter presents a new approach for *Query-based Autonomous Capability Estimation* (QACE) of black-box SDM systems in stochastic settings. Our approach uses a restricted form of interaction with the input SDM agent (referred to as SDMA) to learn a probabilistic model of its capabilities. The learned model captures high-level user-interpretable capabilities, such as the conditions under which an autonomous vehicle could back out of a garage, or reach a certain target location, along with the probabilities of possible outcomes of executing each such capability. The resulting learned models directly provide interpretable representations of the scope of SDMA's capabilities. They can also be used to enable and support approaches for explaining SDMA's behavior that require closed-form models (e.g., (Sreedharan *et al.*, 2018)). We assume that the input SDMA provides a minimal query-response interface that is already commonly supported by contemporary SDM systems. In particular, SDMA should reveal capability names defining how each of its capabilities can be invoked, and it should be able to accept user-defined instructions in the form of sequences of such capabilities. These requirements are typically supported by SDM systems by definition.

The main technical problem for QACE is to automatically compute "queries" in the form of instruction sequences and policies, and to learn a probabilistic model for each

capability based on SDMA's "responses" in the form of executions. Depending on the scenario, these executions can be in the real world, or in a simulator for safety-critical settings. Since the set of possible queries of this form is exponential in the state space, naïve approaches for enumerating and selecting useful queries based on information gain metrics are infeasible.

Main contributions This chapter presents the first approach for query-based assessment of SDMAs in stochastic settings with minimal assumptions on SDMA internals. In addition, it is also the first approach for reducing query synthesis for SDMA assessment to full-observable non-deterministic (FOND) planning (Cimatti *et al.*, 1998). Empirical evaluation shows that these contributions enable our approach to carry out scalable assessment in both embodied and vanilla SDMAs.

We express the learned models using an input concept vocabulary that is known to the target user group. Such vocabularies span multiple tasks and environments. They can be acquired through parallel streams of research on interactive concept acquisition (Kim *et al.*, 2015; Lage and Doshi-Velez, 2020) or explained to users through demonstrations and training (Schulze *et al.*, 2000). These concepts can be modeled as binary-valued *predicates* that have their associated evaluation functions (Mao *et al.*, 2022). We use the syntax and semantics of a well-established relational SDM model representation language, Probabilistic Planning Domain Definition Language (PPDDL) (Def. 2), to express the learned models.

Related work on the problem addresses model learning from passively collected observations of agent behavior (Pasula *et al.*, 2007; Martínez *et al.*, 2016; Juba and Stern, 2022); and by exploring the state space using simulators (Chitnis *et al.*, 2021; Mao *et al.*, 2022). However, passive learning approaches can learn incorrect models as they do not have the ability to generate interventional or counterfactual data;

exploration techniques can be sample inefficient because they don't take into account uncertainty and incompleteness in the model being learned to guide their exploration (see Sec. 6.6 for a greater discussion).

In addition to the key contributions mentioned earlier, our results (Sec. 6.5) show that the approaches for query synthesis in this chapter do not place any additional requirements on black-box SDMAs but significantly improve the following factors: (i) convergence rate and sample efficiency for learning relational models of SDMAs with complex capabilities, (ii) few-shot generalizability of learned models to larger environments, and (iii) accuracy of the learned model w.r.t. the ground truth SDMA capabilities. convergence rate to the sound and complete model.

6.2 Preliminaries

6.2.1 SDMA Setup

We consider SDMAs that operate in stochastic and fully observable environments. An SDMA can be represented as a 3-tuple $\langle \mathcal{X}, \mathcal{C}, \mathcal{T} \rangle$, where $\mathcal{X}$ is the environment state space that the SDMA operates in, $\mathcal{C}$ is the set of SDMA's capabilities (capability names, e.g., "place object x at location y" or "arrange table x") that the SDMA can execute, and $\mathcal{T} : \mathcal{X} \times \mathcal{C} \rightarrow \mu\mathcal{X}$ is the stochastic black-box transition model determining the effects of SDMA's capabilities on the environment. Here, $\mu\mathcal{X}$ is the space of probability distributions on $\mathcal{X}$. Note that the semantics of $\mathcal{C}$ are not known to the user(s) and $\mathcal{X}$ may not be user-interpretable. The only information available about the SDMA is the instruction set in the form of capability names, represented as $\mathcal{C}_N$.

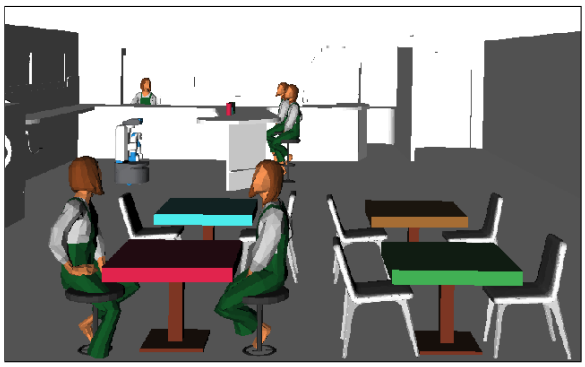


Figure 11. The cafe server robot environment in OpenRave simulator.

This isn't a restricting assumption as the SDMAs must reveal their instruction sets for usability.

Running Example Consider a cafe server robot that can pick and place items like plates, cans, etc., from various locations in the cafe, like the counter, tables, etc., and also move between these locations. A capability *`pick-item (?location ?item)`* would allow a user to instruct the robot to pick up an item like a soda can for any location. However, without knowing its description, the user would not know under what conditions the robot could execute this capability and what the effects will be.

6.2.2 Object-centric Concept Representation

We aim to learn representations that are generalizable, i.e., the transition dynamics learned should be impervious to environment-specific properties such as numbers and configurations of objects. Additionally, the learned capability models should hold in different settings of objects in the environment as long as the SDMA's capabilities does not change. To this effect, we learn the SDMA's transition model in terms of interpretable concepts that can be represented using first-order logic *predicates*. This

is a common formalism for expressing the symbolic models of SDMAs (Zhi-Xuan *et al.*, 2020; Mao *et al.*, 2022). We formally represent them using a set of object-centric predicates $\mathcal{P}$. The set of predicates used for cafe server robot in Fig. 11 can be `(robot-at ?location)`, `(empty-arm)`, `(has-charge)`, `(at ?location ?item)`, and `(holding ?item)`. Here, `?` precedes an argument that can be replaced by an object in the environment. E.g., `(robot-at tableRed)` means "robot is at the red table." As mentioned earlier, we assume these predicates along with their Boolean evaluation functions (which evaluate to true if predicate is true in a state) are available as input. Learning such predicates is also an interesting but orthogonal direction of research (Mao *et al.*, 2022; Sreedharan *et al.*, 2022; Das *et al.*, 2023).

6.2.3 Abstraction

Using an object-centric predicate representation induces an abstraction of environment states $\mathcal{X}$ to high-level logical states $\mathcal{S}$ expressible in predicate vocabulary $\mathcal{P}$. This abstraction can be formalized using a surjective function $f : \mathcal{X} \to \mathcal{S}$. E.g., in the cafe server robot, the concrete state x may refer to roll, pitch, and yaw values. On the other hand, the abstract state s corresponding to x will consist of truth values of all the predicates (Srivastava *et al.*, 2014, 2016; Mao *et al.*, 2022). Fig. 12(left) shows an example from the cafe server SDM setting, where a concrete low-level state is represented as xyz-coordinates, roll, pitch, and yaw values by the simulator. This state is then converted to a high-level state shown in the figure. We use the Boolean evaluation functions for evaluating each predicate. The state is represented as conjunction of the true predicates.

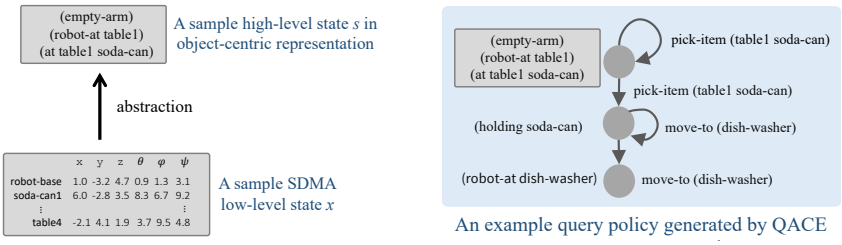


Figure 12. An example of abstraction of low-level state into a high level state (left) and an example of a policy simulation query (right). For the policy, the labels on the left of nodes correspond to state properties that must be true in those states, and the labels on right of edges correspond to the capabilities for each edge. The policy simulation query corresponds to: "Given that the robot and *soda-can* are at *table1*, what will happen if the robot follows the following policy: if there is an item on the table and arm is empty, pick up the item; if an item is in the hand and location is not dishwasher, move to the dishwasher?".

```
(:capability pick-item
 :parameters (?location ?item)
 :precondition (and
   (empty-arm) (has-charge)
   (robot-at ?location)
   (at ?location ?item))
 :effect (and (probabilistic
   0.7 (and (not (empty-arm))
        (not (at ?location ?item))
        (holding ?item))
   0.2 (and (not (has-charge)))
   0.1 (and)))          #No-change
```

Figure 13. PPDDL description for the cafe server robot's *pick-item* capability.

6.2.4 Probabilistic Transition Model

Abstraction induces an abstract transition model $\mathcal{T}' : \mathcal{S} \times \mathcal{C} \to \mu\mathcal{S}$, where $\mu\mathcal{S}$ is the space of probability distributions on $\mathcal{S}$. This is done by converting each transition $\langle x, c, x' \rangle \in \mathcal{T}$ to $\langle s, c, s' \rangle \in \mathcal{T}'$ using predicate evaluators such that $f(x) = s$ and $f(x') = s'$. Now, $\mathcal{T}'$ can be expressed as model M that is a set of parame-

terized action (capability in our case) schema, where each $c \in \mathcal{C}$ is described as $c = \langle name(c), pre(c), eff(c) \rangle$, where $name(c) \in \mathcal{C}_N$ refers to name and arguments (parameters) of c; *pre(c)* refers to the preconditions of the capability c represented as a conjunctive formula defined over $\mathcal{P}$ that must be true in a state to execute c; and *eff(c)* refers to the set of conjunctive formulas over $\mathcal{P}$, each of which becomes true on executing c with an associated probability. The result of executing c for a model M is a state $c(s) = s'$ such that $P_M(s'|s, c) > 0$ and one (and only one) of the effects of c becomes true in s'. We also use $\langle s, c, s' \rangle$ triplet to refer to $c(s) = s'$. This representation is similar to the Probabilistic Planning Domain Definition Language (PPDDL), which can compactly describe the SDMA's capabilities. E.g., the cafe server robot has three capabilities (shown here as `name(args)`): `pick-item(?location ?item)`; `place-item(?location ?item)`; and `move(?source ?destination)`. The description of `pick-item` in PPDDL is shown in Fig. 13.

6.2.5 Fully Observable Non-Deterministic (FOND) Model

A fully-observable non-deterministic (FOND) planning model (Cimatti *et al.*, 1998) can be viewed as a probabilistic planning model without the probabilities associated with each effect pair. On executing an action, one of its possible effects is chosen. The solution to these planning models is a *partial policy* $\Pi : S \rightarrow A$ that maps each state to an action that the agent should execute in that state. As shown by Cimatti *et al.* (1998) and Daniele *et al.* (1999), the solution is a (i) *weak solution* if the resulting plan may achieve the goal without any guarantee; (ii) *strong solution* if the resulting plan is guaranteed to reach the goal; and (iii) *strong cyclic solution* if the resulting plan is

```
(:action pick-item
 :parameters (?location ?item)
 :precondition (and
   (empty-arm) (has-charge)
   (robot-at ?location)
   (at ?location ?item))
 :effect (oneof
   (and (not (empty-arm))
      (not (at ?location ?item))
      (holding ?item))
   (and (not (has-charge))))
```

Figure 14. FOND description for the *pick-item* capability of the cafe server robot.

guaranteed to reach the goal under the assumption that in the limit, each action will lead to each of its effects.

Fig. 14 shows a sample FOND description of the pick-item capability (shown in Fig. 13). Note that there are no probabilities associated with each possible effect set, as the representation only shows that one of these possible set of effects is possible on executing this capability. Also, the language only supports the keyword `action`, hence it is used for representing capability in the first line.

6.2.6 Variational Distance

Given a black-box SDMA $\mathcal{A}$, we learn the probabilistic model M representing its capabilities. To measure how close M is to the true SDMA transition model $\mathcal{T}'$, we use variational distance – a standard measure in probabilistic-model learning literature (Pasula *et al.*, 2007; Martínez *et al.*, 2016; Ng and Petrick, 2019; Chitnis *et al.*, 2021). It is based on the *total variation distance* between two probability

distributions $\mathcal{T}'$ and M, given as:

$$\delta(\mathcal{T}', M) = \frac{1}{|\mathcal{D}|} \sum_{\langle s,c,s' \rangle \in \mathcal{D}} \left| P_{\mathcal{T}'}(s'|s,c) - P_M(s'|s,c) \right| \tag{6.1}$$

where $\mathcal{D}$ is the set of test samples ($\langle s, c, s' \rangle$ triplets) that we generate using $\mathcal{T}'$ to measure the accuracy of our approach. As shown by Pinsker (1964), $\delta(\mathcal{T}', M) \leq \sqrt{0.5 \times D_{KL}(\mathcal{T}' \,\|\, M)}$, where D_{KL} is the KL divergence.

6.3 The Capability Assessment Task

In this work, we aim to learn a probabilistic transition model $\mathcal{T}'$ of a black-box SDMA as a model M, given a set of user-interpretable concepts as predicates $\mathcal{P}$ along with their evaluation functions, and the capability names C_N corresponding to the SDMA's capabilities. Formally, the assessment task is:

Definition 25. Given a set of predicates $\mathcal{P}$ along with their Boolean evaluation functions, capability names $\mathcal{C}_N$, and a black-box SDMA $\mathcal{A}$ in a fully observable, stochastic, and static environment, the *capability assessment task* $\langle \mathcal{A}, \mathcal{P}, \mathcal{C}_N, \mathcal{T}' \rangle$ is defined as the task of learning the probabilistic transition model $\mathcal{T}'$ of the SDMA $\mathcal{A}$ expressed using $\mathcal{P}$.

The solution to this task is a model M that should ideally be the same as $\mathcal{T}'$ for correctness. In practice, $\mathcal{T}'$ need not be in PPDDL, so the correctness should be evaluated along multiple dimensions.

6.3.1 Notions of Model Correctness

As discussed in Sec. 6.2, variational distance is one way to capture the correctness of the learned model. This is useful when the learned model and the SDMA's model are not in the same representation. The correctness of a model can also be measured using qualitative properties such as soundness and completeness. The learned model M should be sound and complete w.r.t. the SDMA's high-level model $\mathcal{T}'$, i.e., for all combinations of c, s, and s', if a transition $\langle s, c, s' \rangle$ is possible according to $\mathcal{T}'$, then it should also be possible under M, and vice versa. Here, $\langle s, c, s' \rangle$ is consistent with M (or $\mathcal{T}'$) if $P(s'|s, c) > 0$ according to M (or $\mathcal{T}'$). We formally define this as:

Definition 26. Let $\langle \mathcal{A}, \mathcal{P}, \mathcal{C}_N, \mathcal{T} \rangle$ be a capability assessment task with a learned model M as its solution. M is *sound* iff each transition $\langle s, c, s' \rangle$ consistent with M is also consistent with $\mathcal{T}'$. M is *complete* iff every transition that is consistent with $\mathcal{T}'$ is also consistent with M.

This also means that if $\mathcal{T}'$ is also a PPDDL model, then (i) any precondition or effect learned as part of M is also present in $\mathcal{T}'$ (soundness), and; (ii) all the preconditions and effects present in $\mathcal{T}'$ should be present in M (completeness). Additionally, a probabilistic model is *correct* if it is sound and complete, and the probabilities for each effect set in each of its capabilities are the same as that of $\mathcal{T}'$.

6.3.2 Interactive Capability Assessment

To solve the capability assessment task, we must identify the preconditions and effects of each capability in terms of conjunctive formulae expressed over $\mathcal{P}$. At a very high-level, we do this by identifying that a probabilistic model can be expressed as a

set of capabilities $c \in C$, each of which has two places where we can add a predicate p, namely precondition and effect. We call these *locations* within each capability. We then enumerate through these $2 \times |\mathcal{C}|$ locations and figure out the correct form of each predicate at each of those locations. To do this we need to consider three forms: (i) adding it as p, i.e., the predicate must be true for that capability to execute (when the location is precondition), or it becomes true on executing it (when the location is effect); (ii) adding it as *not(p)*, i.e., the predicate must be false for that capability to execute (when the location is precondition), or it becomes false on executing it (when the location is effect); (iii) not adding it at all, i.e., the capability execution does not depend on it (when the location is precondition), or the capability does not modify it (when the location is effect).

6.3.2.1 Model Pruning

Let $\mathcal{M}$ represent the set of all possible transition models expressible in terms of $\mathcal{P}$ and $\mathcal{C}$. We must prune the set of possible models to solve the capability assessment task, ideally bringing it to a singleton. We achieve this by posing queries to the SDMA and using the responses to the queries as data to eliminate the inconsistent models from the set of possible models $\mathcal{M}$.

Given a location (precondition or effect in a capability), the set of models corresponding to a predicate will consist of 3 transition models: one each corresponding to the three ways we can add the predicate in that location. We call these three possible models M_T, M_F, M_I, corresponding to adding p (true), *not(p)* (false), and not adding p (ignored), respectively at that location.

Note that the actual set of possible transition models is infinite due to the proba-

bilities associated with each transition. To simplify this, we first constrain the set of possible models by ignoring the probabilities, and learn a non-deterministic transition model (commonly referred to as a FOND model (Cimatti *et al.*, 1998)) instead of a probabilistic one. We later learn the probabilities using maximum likelihood estimation based on the transitions observed as part of the query responses.

6.3.2.2 SimulatorUuse

Using the standard assumption of a simulator's availability in research on SDM, QACE solves the capability assessment task (Sec. 6.3) by issuing queries to the SDMA and observing its responses in the form of its execution in the simulator. In non-safety-critical scenarios, this approach can work without a simulator too. The interface required to answer the queries is rudimentary as the SDMA $\mathcal{A}$ need not have access to its transition model $\mathcal{T}'$ (or $\mathcal{T}$). Rather, it should be able to interact with the environment (or a simulator) to answer the queries. We next present the types of queries we use, followed by algorithms for generating them and for inferring the SDMA's model using its responses to the queries.

6.3.2.3 Policy Simulation Queries (Q_{PS})

These queries ask the SDMA $\mathcal{A}$ to execute a given policy multiple times. More precisely, a Q_{PS} query is a tuple $\langle s_I, \pi, G, \alpha, \eta \rangle$ where $s_I \in \mathcal{S}$ is a state, π is a partial policy that maps each reachable state to a capability, G is a logical predicate formula that expresses a stopping condition, α is an execution cutoff bound representing the maximum number of execution steps, and η is an attempt limit. Note that the

query (including the policy) is created entirely by our solution approach without any interaction with the SDMA. Q_{PS} queries ask $\mathcal{A}$ to execute π, η times. In each iteration, execution continues until either the stopping goal condition G or the execution bound α is reached. E.g., "Given that the robot, `soda-can`, `plate1`, `bowl3` are at `table4`, what will happen if the robot follows the following policy: if there is an item on the table and arm is empty, pick up the item; if an item is in the hand and location is not dishwasher, move to the dishwasher; if an item is in the hand and location is dishwasher, place the item in the dishwasher?" Such queries will be used to learn both preconditions and effects (Sec. 6.3.5).

A response to such queries is an execution in the simulator and η traces of these simulator executions. Formally, the response θ_{PS} for a query $q_{PS} \in Q_{PS}$ is a tuple $\langle b, \zeta \rangle$, where $b \in \{\top, \bot\}$ indicates weather if the SDMA reached a goal state $s_G \models G$, and ζ are the corresponding triplets $\langle s, c, s' \rangle$ generated during the η policy executions. If the SDMA reaches s_G even once during the η simulations, b is $\top$, representing that the goal can be reached using this policy. Next, we discuss how these responses are used to prune the set of possible models and learn the correct transition model of the SDMA.

Fig. 12(right) shows an example of a policy simulation query. Note that the initial state is shown adjacent to the top-most node. A partial policy is a mapping from a partial state to a capability. Such queries can be generated using non-deterministic planners like PRP (Muise *et al.*, 2012).

Algorithm 5: QACE Algorithm

Input : predicates $\mathcal{P}$; capability names $\mathcal{C}_N$;
state s; SDMA $\mathcal{A}$; hyperparameters α, η;
FOND Planner ρ

Output : M

1 $L \leftarrow \{pre, \mathit{eff}\} \times \mathcal{C}_N$
2 $M^* \leftarrow$ initializeModel $(\mathcal{P}, \mathcal{C}_N)$
3 **for** each $\langle l, p \rangle \in \langle L, \mathcal{P} \rangle$ **do**
4 Generate M_T, M_F, M_I by setting p at l in M^*
5 **for** each pair M_i, M_j in $\{M_T, M_F, M_I\}$ **do**
6 $q \leftarrow$ generateQuery$(M_i, M_j, \alpha, \eta, s, \rho)$
7 $\theta_{\mathcal{A}}, \mathbb{S} \leftarrow$ getResponse$(q, \mathcal{A}, s)$
8 $M^* \leftarrow$ pruneModels $(\theta_{\mathcal{A}}, M_i, M_j)$
9 $M^* \leftarrow$ learn possible stochastic effects of capability with c_N in l using ζ (in $\theta_{\mathcal{A}}$)
10 $M \leftarrow$ learnProbabilitiesOfStochasticEffects(ζ, M^*)
11 **return** M

6.3.3 Query-based Autonomous Capability Estimation (QACE) Algorithm

We now discuss how we solve the capability assessment task using the Query-based Autonomous Capability Estimation algorithm (Alg. 5), which works in two phases. In the first phase, QACE learns all capabilities' preconditions and non-deterministic effects using the policy simulation queries (Sec. 6.3.4). In the second phase, QACE converts the non-deterministic effects of capabilities into probabilistic effects (Sec. 6.3.5). We now explain the learning portion (lines 3-11) in detail.

QACE first initializes a model M^* over the predicates in $\mathcal{P}$ with capabilities having names $c_N \in \mathcal{C}_N$. All the preconditions and effects for all capabilities are empty in this initial model. QACE uses M^* to maintain the current partially learned model. QACE iterates over all combinations of L and $\mathcal{P}$ (line 4). For each pair, QACE creates 3 candidate models M_T, M_F, and M_I as mentioned earlier. It then takes 2 of these (line

5) and generates a query q (line 6) such that responses to the query q from the two models are logically inconsistent (see Sec. 6.3.4). The query q is then posed to the SDMA $\mathcal{A}$ whose response is stored as $\theta_{\mathcal{A}}$ (line 7). QACE finally prunes at least one of the two models by comparing their responses (which are logically inconsistent) with the response $\theta_{\mathcal{A}}$ of the SDMA on that query (line 8). QACE also updates the effects of all models in the set of possible models to speed up the learning process (line 9). Finally, it learns the probabilities of the observed stochastic effects using maximum likelihood estimation (line 10). An important feature of the algorithm (similar to PLEX (Mehta *et al.*, 2011) and AIA (Verma *et al.*, 2021a)) is that it keeps track of all the locations where it hasn't identified the correct way of adding a predicate. The next section presents our approach for generating the queries in line 6.

6.3.4 Algorithms for Query Synthesis

One of the main challenges in interactive model learning is to generate the queries we discussed above and to learn the agent's model using them. Although active learning (Settles, 2012) addresses the related problem of figuring out which data sets to request labels for, vanilla active learning approaches are not directly applicable here because the possible set of queries expressible using the literals in a domain is vast: it is the set of all policies expressible using $\mathcal{C}_N$. Query-based learning approaches use an estimate of the utility of a query to select "good" queries. This can be a multi-valued measure like *information gain* (Sollich and Saad, 1994), *value* (Macke *et al.*, 2021), etc. or a binary-valued attribute like *distinguishability* (Verma *et al.*, 2021a), etc. We use distinguishability as a measure to identify useful queries. According to it, a query

q is distinguishing w.r.t. two models if responses by both models to q are logically inconsistent. We now discuss methods for generating such queries.

Generating distinguishing queries QACE automates the generation of queries using search. As part of the algorithm, a model M^* is used to generate the three possible models M_T, M_F, and M_I for a specific predicate p and location l combination. Other than the predicate p at location l, these models are exactly the same. A forward search is used to generate the policy simulation queries with two possible models M_i, M_j chosen randomly from M_T, M_F, and M_I. The forward search is initiated with an initial state $\langle s_0^i, s_0^j \rangle$ as the root of the search tree, where s_0^i and s_0^j are copies of the same state s_0 from which we are starting the search. The edges of the tree correspond to the capabilities with arguments replaced with objects in the environment. Nodes correspond to the two states resulting from applying the capability in the parent state according to the two possible models. E.g., consider that a transition $\langle s_0^i, c, s_1^i \rangle$ is possible according to the model M_i, and let $\langle s_0^j, c, s_1^j \rangle$ be the corresponding transition (by applying the same effect set of c as h_i) according to the model M_j. Now there will be an edge in the forward search tree with label c such that parent node is $\langle s_0^i, s_0^j \rangle$ and child node is $\langle s_1^i, s_1^j \rangle$. The search process terminates when a node $\langle s^i, s^j \rangle$ is reached such that either the states s^i and s^j don't match, or the preconditions of the same capability were met in the state according to one of the possible models but not according to the other. Vanilla forward search scales poorly with the number of capabilities and objects in the environment. To address this we reduce the problem to a fully observable non deterministic (FOND) planning problem. This can be solved by any FOND planner. The output of this search is a policy π to reach a state where the two models, M_i and M_j predict different outcomes. The query $\langle s_I, \pi, G, \alpha, \eta \rangle$ resulting from this search is such that s_I is set to the initial state s_0, π is the output

policy, G is the goal state where the models' responses doesn't match, α and η are hyperparameters as mentioned earlier. We next see how to use these queries to prune out the incorrect models.

Generating Policy Simulation Queries using PRP QACE automates the generation of queries using FOND planning problems. QACE always generates queries to distinguish between models that differ only on one predicate corresponding to just one location (a precondition or effect in a capability). The main idea behind generating such queries is that the responses to the query q from the two models should be logically inconsistent. To generate the policy simulation queries, QACE creates a FOND planning model and a problem.

Let M_i and M_j be a pair of FOND models expressed using $\mathcal{P}$ and $\mathcal{C}$ where $i, j \in \{T, F, I\}$. QACE renames the predicates and capabilities in M_i and M_j as $\mathcal{P}_i$ and $\mathcal{P}_j$, and $\mathcal{C}_i$ and $\mathcal{C}_j$, respectively, so that there are no intersections and a pair of states in the two models can be progressed independently using pairs of capabilities. This gives a planning model $M_{ij} = \langle \mathcal{P}_{ij}, \mathcal{C}_{ij} \rangle$. Here, $\mathcal{P}_{ij} = \mathcal{P}_i \cup \mathcal{P}_j \cup \{\textit{(goal)}\}$, where *(goal)* is a 0-ary predicate. It is used to identify when the goal for the FOND planning problem is reached. For each capability $\langle c_i, c_j \rangle \in \langle \mathcal{C}_i, \mathcal{C}_j \rangle$ such that their names match, $\textit{pre}(c_{ij})$ of the combined capability c_{ij} is disjunction of preconditions of c_i and c_j. For $e(c_{ij}) \in \textit{eff}(c_{ij})$ QACE adds three conditional effects: (i) $\textit{pre}(c_i) \wedge \textit{pre}(c_j) \Rightarrow e(c_i) \wedge e(c_j)$; (ii) $\textit{pre}(c_i) \wedge \neg \textit{pre}(c_j) \Rightarrow \textit{(goal)}$; and (iii) $\neg \textit{pre}(c_i) \wedge \textit{pre}(c_j) \Rightarrow \textit{(goal)}$. An example of this process is included in the next section.

Starting from an initial state, the FOND problem uses one of these states and maintains two different copies of all the objects in the environment, one corresponding to each of the models. Each model only manipulates the objects in its own copy. QACE then solves a planning problem that has an initial state $s_{I_{ij}} = \{p_i^{*_1}, \ldots, p_i^{*_z}, p_j^{*_1}, \ldots, p_j^{*_z}\}$

and a goal state $G_{ij} = (goal) \vee [\exists p \in \mathcal{P}^*_{ij}(p_i \wedge \neg p_j) \vee (\neg p_i \wedge p_j)]$. Here, $\mathcal{P}^*$ represents the grounded version of predicates $\mathcal{P}$ using objects O in the environment. The partial policy π generated as a solution to this planning problem is a *strong solution*. As shown by Cimatti *et al.* (1998), the solution is a *strong solution* if the resulting plan is guaranteed to reach the goal. The solution partial policy will lead the two models in a state where at least one capability cannot be applied, and hence the *(goal)* predicate becomes true. This is possible because the models differ only in the way one predicate is added at a location. We formalize this with the following lemma. The proof is available in Sec. 6.4.

Lemma 12. Given two models M_i and M_j such that both are abstractions of the same FOND model, and are at the same level of abstraction with only one predicate differing in way it is added in one of the location, the intermediate FOND planning problem created using QACE to generate policy simulation queries has a strong solution.

We next present an example of a sample planning domain and problem using which QACE generates a query. Recall that a FOND planning problem consists of two components, a planning domain and a planning problem. We will see an example of both below.

FOND planning domain Consider we have a capability `move-vehicle (?frm ?to)` in the Cafe server robot, and we already know one of its preconditions; `(has-charge)`. We are now trying to find what will be the correct way to add the predicate `(robot-at ?frm)` in the precondition of this `move-vehicle (?frm ?to)` capability. Consider we have two models M_i and M_j, where $i = T$ and $j = F$. We will represent their `move-vehicle` capability as follows:

To create a query, we will combine the `move-vehicle` capabilities into a combined

```
(:action move-vehicle_i
  :parameters (?frm - loc ?to - loc)
  :precondition (and (has-charge_i)
    (robot-at_i ?frm))
  :effect (and ))

(:action move-vehicle_j
  :parameters (?frm - loc ?to - loc)
  :precondition (and (has-charge_j)
    (not (robot-at_i ?frm)))
  :effect (and ))
```

capability. This should be done in such a way that the combined capability is executed when at least one of the model's preconditions are satisfied. Hence, for each capability $\langle c_i, c_j \rangle \in \langle \mathcal{C}_i, \mathcal{C}_j \rangle$ s.t. $name(c_i) = name(c_j)$, $pre(c_{ij}) = pre(c_i) \vee pre(c_j)$.

Now on executing this combined capability, we should be able to identify if the preconditions or effects of the capabilities in the two models M_i and M_j are different. To take these into account, the effect of the combined capability will be such that for each $e(c_{ij}) \in \mathit{eff}(c_{ij})$ we add three conditional effects: (i) $pre(c_i) \wedge pre(c_j) \Rightarrow e(c_i) \wedge e(c_j)$; (ii) $pre(c_i) \wedge \neg pre(c_j) \Rightarrow$ *(goal)*; and (iii) $\neg pre(c_i) \wedge pre(c_j) \Rightarrow$ *(goal)*. The condition (ii) and (iii) helps identify that the precondition of the capability according to only one of the models M_i or M_j is satisfied. The condition (i) captures the case where the precondition of the capability according to both the models are satisfied. In this case, the effects of the capability according to both the models are applied.

Applying it here for the `move-vehicle` capability, we get:

```
(:action move-vehicle_ij
  :parameters (?frm - loc ?to - loc)
  :precondition (or
    (and (has-charge_i)
      (robot-at_i ?frm))
    (and (has-charge_j)
      (not (robot-at_j ?frm)))
  )
  :effect (and
    (when (and (has-charge_i)
        (robot-at_i ?frm)
        (has-charge_j)
        (not (robot-at_j ?frm)))
      (and )
    )
    (when (and (has-charge_i)
        (robot-at_i ?frm)
        (or (not (has-charge_j))
          (robot-at_j ?frm)))
      (and (goal))
    )
    (when (and (has-charge_j)
        (not (robot-at_j ?frm))
        (or (not (has-charge_i))
          (not (robot-at_i ?frm))))
      (and (goal))
    )
  )
)
```

Note that we have expanded $pre(c_i) \wedge \neg pre(c_j)$ using disjunction of negations of all predicates in $pre(c_j)$, etc. A pictorial example of a similar process for the ***has-charge*** predicate in effects is shown in Fig. 15 below.

FOND planning problem The FOND planner must maintain two different copies of all the objects in the environment, one corresponding to each of the models M_i and M_j. Each model only manipulates the objects in its own copy. The initial

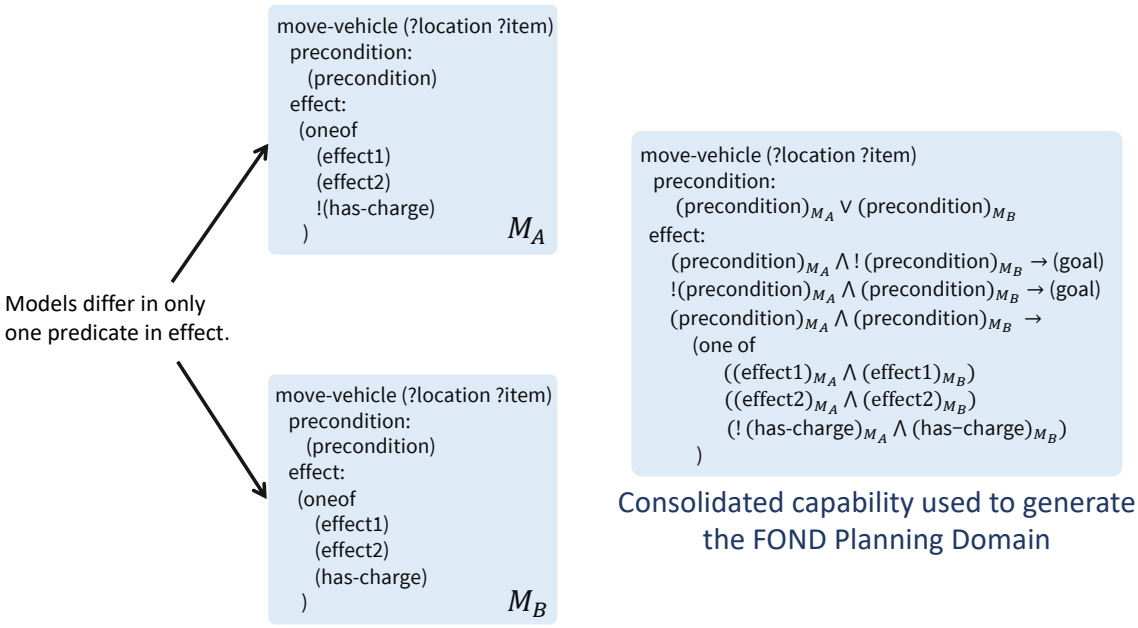


Figure 15. An example showing how two models M_i and M_j are combined to generate a FOND planning domain when the predicate is being added in effect of a capability. Note that the models only differ in one predicate having different form in both models.

state of this planning problem s_I is $\{p_i^1, \ldots, p_i^z, p_j^1, \ldots, p_j^z\}$. The goal formula G is $(goal) \vee [\exists p \in \mathcal{P}(p_i \wedge \neg p_j) \vee (\neg p_i \wedge p_j)]$. Here $(goal)$ becomes true when a capability is executed by the policy such that the precondition of that capability is satisfied according to only one of the two models. The other condition captures the fact that a state is reached such that at least one of the predicate is true according to one of the models and false according to the other. This points to a difference in the effects of the capability that was executed last.

6.3.5 Learning Probabilistic Models Using Query Responses

At this point, QACE already has a query such that the response to the query by the two possible models does not match. We next see how to prune out the model

inconsistent with the SDMA. QACE poses the query generated earlier to the SDMA and gets its response (line 7 in Alg. 5). If the SDMA can successfully execute the policy, QACE compares the response of the two models with that of the SDMA and prunes out the model whose response does not match with that of the SDMA. If the SDMA cannot execute the policy, i.e., SDMA fails to execute some capability in the policy, then the models cannot be pruned directly. In such a case, a new initial state s_0 must be chosen to generate a new query starting from that initial state. Since generating new queries for the same pair of models can be time consuming, we preempt this issue by creating a pool of states $\mathbb{S}$ that can execute the capabilities using a directed exploration of the state space with the current partially learned model as discussed below.

Directed exploration A partially learned model is a model where one or more capabilities have been learned (the correct preconditions have been identified for each capability and at least one effect is learned). Once we have such a model, we can do a directed exploration of the state space for these capabilities by only executing a learned capability if the preconditions are satisfied. This helps in reducing the sample complexity since the simulator is only called when we know that the capability will execute successfully, thereby allowing us to explore different parts of the state space efficiently. If a capability's preconditions are not learned, all of its groundings might need to be executed from the state. In the worst case, to escape local minima where no models can be pruned, we would need to perform a randomized search for a state where a capability is executable by the SDMA. In practice, we observed that using directed exploration to generate a pool of states gives at least one grounded capability instance. This helps ensure that during query generation, the approach does not spend a long time searching for a state where a capability is executable.

Learning probabilities of stochastic effects After QACE learns the non-deterministic model, to learn the probabilities of the learned effects it uses the transitions collected as part of responses to queries. This is done using Maximum Likelihood Estimation (MLE) (Fisher, 1922). For each triplet $\langle s, c, s' \rangle$ seen in the collected data, let $count_c$ be the number of times a capability c is observed. Now, for each effect set, the probability of that effect set becoming true on executing that capability c is given as the number of times that effect is observed on executing c divided by $count_c$. As we increase the value of the hyperparameter η, we increase the number of collected triplets, thereby improving the probability values calculated using this approach.

6.3.6 Example Run of the Algorithm

Consider that the set of predicates consists of *(has-charge)* and *(robot-at ?frm)*, and *move-vehicle* is one of the capability. Consider that we are starting with an empty model in line 2 of Alg. 5. Now consider that the actual precondition of the *move-vehicle* capability is *(has-charge)* $\wedge$ *(robot-at ?frm)*. The automated query generation process will involve executing the capability successfully in some state s by the policy. The SDMA can only execute the capability in s if *(has-charge)* $\wedge$ *(robot-at ?frm)* is true in s. As mentioned in Sec. 6.3.5, if s doesn't fulfill this criterion (i.e., the SDMA fails to execute the policy successfully) a new query is generated from a new initial state s'. Hence, this property of executing the capability in a state having *(has-charge)* $\wedge$ *(robot-at ?frm)* is ensured. Now, when reasoning about *(has-charge)*, the policy can ask the agent to execute that capability in the state $s \setminus$ *(has-charge)* and if the SDMA fails to execute it then

it means *(has-charge)* is part of the precondition. Similarly, this can be done for *(robot-at ?frm)* independently.

In the worst case, the search for a state s where a query policy is executable will be exponential, but as the evaluations show, we can learn the correct model much faster. We also mention a way to overcome this in Sec. 6.3.5. Please note that even for methods like reinforcement learning, the worst-case upper bound is exponential in terms of the state space.

Possible models and their pruning Now we see how QACE learns a correct model once it finds a state s where a capability is executable by the agent. Consider that QACE is processing the tuple $\langle l, p \rangle = \langle$precondition of *move-vehicle*, *(has-charge)*$\rangle$ in line 3 of Alg. 5.

Now, QACE will generate the three models in line 4: (i) M_T that has *(has-charge)* as precondition of *move-vehicle* capability; (ii) M_F that has $\neg$*(has-charge)* as precondition of *move-vehicle* capability; and (iii) M_I that has an empty precondition for *move-vehicle* capability.

Consider that QACE is considering the pair $\langle M_T, M_F \rangle$ in line 5 of Alg. 5. In the example being considered, executing the *move-vehicle* capability in the state s can help QACE distinguish between M_T and M_F. Here the model M_F will be unable to execute the *move-vehicle* capability in s, whereas the model M_T and the agent will be able to. So QACE will prune M_F in line 8.

Next, QACE will consider the pair $\langle M_T, M_I \rangle$ in line 5 of Alg. 5. Here, to distinguish between these model, QACE will execute the *move-vehicle* capability in a state s' where *(has-charge)* is false. Note that this state is also not generated manually, and the query generation does this autonomously, starting from the state s. Here the model M_T and the agent will fail to execute the capability, whereas the model M_I

will succeed. Hence QACE can prune out M_I, leading it to learn the correct model $M^* = M_T$ where *(has-charge)* is a precondition of the *move-vehicle* capability.

Now starting with this updated current partial model M^*, consider that QACE picks the tuple $\langle l, p \rangle$ = ⟨precondition of *move-vehicle*, *(robot-at ?frm)*⟩ in line 3 of Alg. 5. QACE will then generate three new models in next iteration in line 4: (i) M_T that has *(has-charge)* $\wedge$ *(robot-at ?frm)* as precondition of *move-vehicle* capability; (ii) M_F that has $\neg$*(has-charge)* $\wedge\neg$*(robot-at ?frm)* as precondition of *move-vehicle* capability; and (iii) M_I that has *(has-charge)* as precondition of *move-vehicle* capability. So essentially, QACE builds upon the already learned partial model M^* in previous iterations, and continues refining the model to eventually end up with the correct FOND model.

Once the correct set of preconditions and effects are learned, QACE counts the number of times each effect set was observed on executing each capability and perform the maximum likelihood estimation for each effect set to calculate the probabilities for each effect set. Note that a capability c will at least appear in policies for all $\langle l, p \rangle$ pairs, such that location l corresponds to a precondition or effect in c. So effectively, a capability c can appear in at least $2 \times |\mathcal{P}|$ queries. So we will have at least $2 \times |\mathcal{P}| \times \eta$ samples for each capability.

6.4 Theoretical Analysis and Correctness

We now discuss how the model M of SDMA $\mathcal{A}$ learned using QACE fulfills the notions of correctness (Sec. 6.3.1) discussed earlier. We first show that the model M^* learned before line 10 of QACE (Alg. 5) is sound and complete according to Def. 26. The proof for this is not straightforward, hence we start it by showing that the plan

in the distinguishing queries always ends up with the capability that is part of the pal tuple being concretized at that time. This will help us in limiting our analysis to, at most, the last 2 capabilities in the plan.

Proposition 1. Let M_i, M_j, where $i, j \in \{T, F, I\}$, be the two models generated by adding a predicate p in a location corresponding to a capability c to a model M. Suppose $q = \langle s_I, \pi, G, \alpha, \eta \rangle$ is a distinguishing query for two distinct models M_i, M_j. The last capability in the partial policy π to achieve G will be c.

Proof. We prove this by contradiction. Consider that the last capability of the policy π in the distinguishing query q is $c' \neq c$. Now the query q used to distinguish between M_i and M_j is generated using the FOND planning problem $\langle M_{ij}, s_{I_{ij}}, G_{ij} \rangle$, which has a solution if both the models have different precondition or at least one different effect for the same capability. Since the last capability of the policy is c', the two models either have different preconditions for c' or different effects. This is not possible as, according to Alg. 5, M_i and M_j differ only in precondition or effect of one capability c. Hence $c' = c$. □

We now use this proposition to prove Lemma 12 stated earlier.

Lemma 12. Given two models M_i and M_j such that both are abstractions of the same FOND model, and are at the same level of abstraction with only one predicate differing in way it is added in one of the location, the intermediate FOND planning problem created using QACE to generate policy simulation queries has a strong solution.

Proof (Sketch). We prove this in two parts. In the first part, we consider the case where we are refining the model in terms of the precondition of some capability. Recall that for each capability c_{ij}, we have 3 conditional effects: i) $pre(c_i) \wedge pre(c_j) \Rightarrow e(c_i) \wedge e(c_j)$; (ii) $pre(c_i) \wedge \neg pre(c_j) \Rightarrow$ *(goal)*; and (iii) $\neg pre(c_i) \wedge pre(c_j) \Rightarrow$ *(goal)*. Now, according

to proposition 1, capability c_{ij} has to be the last capability in the policy π. Since the model M_i and M_j differ only in preconditions, condition (ii) or (iii) must be true for c_{ij}. This implies that on executing c_{ij}, the (*goal*) predicate will become true, and executing this policy π will end up in reaching the goal.

In the second part, we consider the case where we are refining the model in terms of the effects of some capability. According to proposition 1, capability c_{ij} has to be the last capability in the policy π. Since the model M_i and M_j differ only in effects, condition (i) must be true for c_{ij}. This implies that on executing c_{ij}, one of the predicates will become true according to one model, and false according to another, and hence executing this policy π will end up in reaching the goal condition G_{ij}. □

Next, we prove the soundness and completeness of the learned model w.r.t. the agent model. Note that an important part of the process is to get a state s, where a capability c can be executed successfully. We can collect this information using some random traces, using a state where all capabilities are applicable, or asking the agent for a state where certain conditions are met (Q_{SR}). We use this information in the proof.

Theorem 9. Let $\mathcal{A}$ be a black-box SDMA with a ground truth transition model $\mathcal{T}'$ expressible in terms of predicates $\mathcal{P}$ and a set of capabilities $\mathcal{C}$. Let M^* be the non-deterministic model expressed in terms of predicates $\mathcal{P}^*$ and capabilities $\mathcal{C}$, and learned using the query-based autonomous capability estimation algorithm (Alg. 5) just before line 10. Let $\mathcal{C}_N$ be a set of capability names corresponding to capabilities $\mathcal{C}$. If $\mathcal{P}^* \subseteq \mathcal{P}$, then the model M^* is *sound* w.r.t. the SDMA transition model $\mathcal{T}'$. Additionally, if $\mathcal{P}^* = \mathcal{P}$, then the model M^* is *complete* w.r.t. the SDMA transition model $\mathcal{T}'$.

Proof. We first prove that given the predicates P, capability names $\mathcal{C}_H$, model of the agent $\mathcal{T}'$, and the model M^* learned by Alg. 5, M^* is sound w.r.t. the model $\mathcal{T}'$. We do this in two cases. The first one showing that the learned preconditions of all the capabilities in M^* are sound, and the second one showing the same thing for learned effects. We use M_T, M_F, and M_I to refer to models corresponding to adding p (true), *not(p)* (false), and not adding p (ignored), respectively to model M^*.

Case 1: Consider the location is precondition in a capability c where we are trying to find the correct way to add a predicate $p \in \mathcal{P}$.

Case 1.1: Let the models we are comparing be M_T and M_I (or M_F). The policy simulation query q to distinguish between these models would involve executing c in a state where p is false. Now, M_T would fail to execute c (as it has p as a positive precondition), and M_I (or M_F) would successfully execute it. If $\mathcal{A}$ can execute c in such a state, we can filter out the model M_T. We can also remove p from a state where $\mathcal{A}$ is known to execute c, and see if it can execute c. If not, we can filter out the model M_I (or M_F).

Case 1.2: Let the models we are comparing be M_F and M_I. The policy simulation query q to distinguish between these models would involve executing c in a state where p is true. M_F would fail to execute c as it has p as a negative precondition, whereas M_I would successfully execute it. If $\mathcal{A}$ can execute c in such a state, we can filter out the model M_T. We can also add p to a state where $\mathcal{A}$ is known to execute c, and see if it can execute c. If not, we can filter out the model M_I.

Case 2: Consider the location is effect in a capability c where we are trying to find the correct way to add a predicate $p \in \mathcal{P}^*$.

Case 2.1: Let the models we are comparing be M_T and M_I (or M_F). The policy simulation query q used to distinguish between these models would involve executing c in a state where p is false. After executing it, the resulting state will have p true according to M_T only. We ask the agent to simulate the policy N times, with p as the goal formula G. If p appears in any of the simulation after executing c, then we learn all the possible effects involving p. Not that the capability has identifiable effects, so if p appears in more than one effect, the corresponding effect will eventually be discovered when concretizing the predicate that uniquely identifies that effect.
Case 2.2: Let the models we are comparing be M_F and M_I. The policy simulation query q used to distinguish between these models would involve executing c in a state where p is true. After executing it, the resulting state will have p true according to M_I only. We ask the agent to simulate the policy η times, with p as the goal formula G. If p appears in any of the runs, then we learn all the possible effects involving p. Not that the capability has identifiable effects, so if p appears in more than one effect, the corresponding effect will eventually be discovered when concretizing the predicate that uniquely identifies that effect.

Combining both cases, we infer that whenever we learn a precondition or effect, it is added in the same form as in the ground truth model $\mathcal{T}'$, hence the learned model M^* is sound w.r.t. $\mathcal{T}'$.

We now prove that given the predicates $\mathcal{P}$, capability names $\mathcal{C}_H$, model of the agent $\mathcal{T}'$, and the model M^* learned by Alg. 5, M^* is complete w.r.t. the model $\mathcal{T}'$. We just showed that the model that we learn is sound as whenever we add a predicate in a precondition or effect, it is in correct mode. Now, since Alg. 5 loops over all possible combinations of predicates and capabilities, for both precondition and effect,

we will learn all the preconditions and effects correctly. Hence, the learned model will be complete w.r.t. the agent model. □

Next, we show that the final step of learning the probabilities for all the effects in each capability converges to the correct probability distribution under the assumption that all the effects of a capability are identifiable. When a capability c is executed in the environment, one of its effects $e_i(c) \in \mathit{eff}(c)$ will be observed in the environment. To learn the correct probability distribution in M, we should accurately identify that effect $e_i(c)$. Hence, the set of effects is *identifiable* if at least one state exists in the environment from which each effect can be uniquely identified when the capability is executed.

Identifiable Effects A set of effects of a capability are identifiable if there exists a state such that when we execute a capability in that state, we can identify which of its effects was executed. Let us consider a capability a, such that $\mathit{pre}(a) = \{p_1 \wedge p_2 \wedge \neg p_3\}$, and $\mathit{eff}(a) = \{\langle p_3 \wedge p_4, 0.2\rangle, \langle p_3 \wedge \neg p_2, 0.5\rangle, \langle p_3 \wedge \neg p_4 \wedge \neg p_2, 0.3\rangle\}$. The effects of this capability are identifiable because if we execute this capability in state $\{p1, p2, p4\}$, we can identify which of its effect is getting executed. This is because, on executing a, we can identify each effect as follows: (i) if the resulting state has p_4 and p_2, then it is the first effect, (ii) if the resulting state has p_4 but not p_2, then it is the second effect, and (iii) if the resulting has neither p_2 nor p_4, then it is the third effect.

We use the concept of identifiable effects to show that the probabilities for all the effects in each capability converges to the correct probability distribution.

Theorem 10. Let $\mathcal{A}$ be a black-box SDMA with a ground truth transition model $\mathcal{T}'$ expressible in terms of predicates $\mathcal{P}$ and a set of capabilities $\mathcal{C}$. Let M be the probabilistic model expressed in terms of predicates $\mathcal{P}^*$ and capabilities $\mathcal{C}$, and learned using the query-based autonomous capability estimation algorithm (Alg. 5). Let

$\mathcal{P} = \mathcal{P}^*$ and M be generated using a sound and complete non-deterministic model M^* in line 11 of Alg. 5, and let all effects of each capability $c \in \mathcal{C}$ be identifiable. The model M is *correct* w.r.t. the model $\mathcal{T}'$ in the limit as η tends to ∞, where η is hyperparameter in query Q_{PS} used in Alg. 5.

Proof (Sketch). Thm. 9 showed that the model learned by Alg. 5 is sound and complete, meaning all the preconditions and effects are correctly learned. Consider that each sample generated by asking an agent to follow a policy is i.i.d. Now, if we consider only the samples in which a capability is applied in a state such that its effects are identifiable effects, then we can use MLE to learn the correct probabilities given infinite such samples. This is a direct consequence of the result that given infinite i.i.d. samples, probabilities learned by maximum likelihood estimation converge to the true probabilities (Kiefer and Wolfowitz, 1956). □

6.5 Empirical Evaluation

We implemented Alg. 5 in Python to evaluate our approach empirically.[7] We found that our query synthesis and interactive learning process leads to (i) few shot generalization; (ii) convergence to a sound and complete model; and (iii) much greater sample efficiency and accuracy for learning lifted SDM models with complex capabilities as compared to the baseline.

Setup We used a *single* training problem with few objects (≤ 7) for all methods in our evaluation and used a test set that was composed of problems containing object counts larger than those in the training set. We ran the experiments on a cluster of Intel Xeon E5-2680 v4 CPUs with CentOS 7.9 running at 2.4 GHz with a memory

[7]Source code available at `https://github.com/AAIR-lab/QACE`

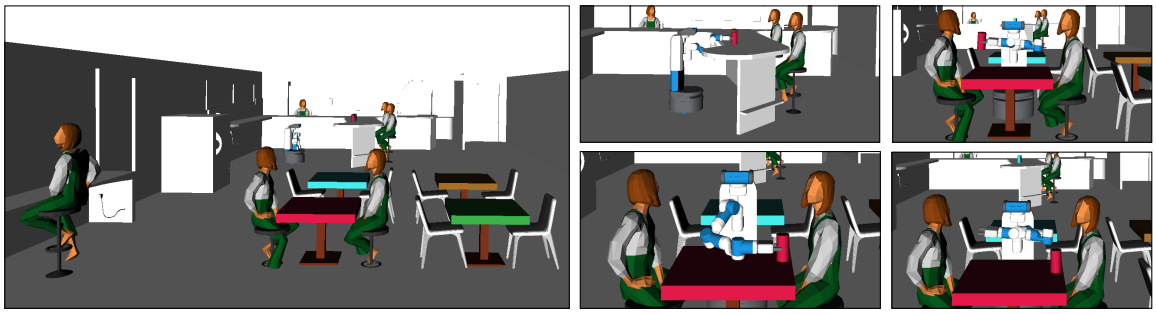


Figure 16. Screen captures from the Cafe Server Robot simulation. The complete environment is shown in the image on the left. The image grid on the right shows screen captures of multiple steps of the robot delivering a `soda-can` to a table.

limit of 8 GB and a time limit of 4 hours. We used PRP (Muise *et al.*, 2012) as the FOND planner to generate the queries (line 6 in Alg. 5). For QACE, we used $\alpha = 2d$ where d is the maximum depth of policies used in queries generated by QACE and $\eta = 5$. All of the methods in our empirical evaluation receive the same training and test sets and are evaluated on the same platform. We used Variational Distance (VD) as presented in Eq. 6.1 to evaluate the quality of the learned SDMA models.

Baseline selection We used the closest SOTA related work, GLIB (Chitnis *et al.*, 2021) as a baseline. GLIB learns a probabilistic model of an intrinsically motivated agent by sampling goals far away from the initial state and making the agent try to reach them. This can be adapted to an assessment setting by moving goal-generation based sampling outside the agent, and, to the best of our knowledge, no existing approach addresses the problem of creating intelligent questions for an SDMA. GLIB has two versions, GLIB-G, which learns the model as a set of grounded noisy deictic rules (NDRs) (Pasula *et al.*, 2007), and GLIB-L, which learns the model as a set of lifted NDRs. We used the same hyperparameters as published for the *Warehouse Robot* and *Driving Agent* and performed extensive tuning for the others and report results with the best performing settings.

The models learned using GLIB cannot be used to calculate the variational distance presented in Eq. 6.1 because for each capability GLIB learns a set of NDRs rather than a unique NDR. In order to maintain parity in comparison, we use GLIB's setup to calculate an approximation of the VD. Using it, we sample 3500 random transitions $\langle s, c, s' \rangle$ from the ground truth transition model $\mathcal{T}'$ using problems in the test set to compute a dataset of transitions $\mathcal{D}$. The sample-based, approximate VD is then given as: $\frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \mathbb{1}_{[s' \neq c_M(s)]}$, where $c_M(s)$ samples the transition using the capability in the learned model output by each method. In Fig. 18, we compare the approximate variational distance of the three approaches w.r.t. $\mathcal{D}$ as we increase the learning time. Note that we also evaluated VD for QACE using Eq. 6.1 and found that $\delta(\mathcal{T}', M) \approx 0$ for our learned model M in all SDMA settings.

6.5.1 SDMAs for Evaluation

To test the efficacy of our approach, we created SDMAs for five different settings including one task and motion planning agent and several SDMAs based on state-of-the-art stochastic planning systems from the literature: *Cafe Server Robot* is a Fetch robot (Wise *et al.*, 2016) that uses the ATM-MDP task and motion planning system (Shah *et al.*, 2020) to plan and act in a restaurant environment to serve food, clear tables, etc.; *Warehouse Robot* is a robot that can stack, unstack, and manage the boxes in a warehouse; a *Driving Agent* that can drive between locations and can repair the vehicle at certain locations; a *First Responder Robot* that can assist in emergency scenarios by driving to emergency spots, providing first-aid and water to victims, etc.; and an *Elevator Control Agent* that can control the operation of multiple elevators in a building. Tab. 12 shows the sizes of the five SDM setups in terms of number

SDM Setup	$\lvert\mathcal{P}\rvert$	$\lvert\mathcal{C}_{\mathbf{N}}\rvert$
Cafe Server Robot	5	4
Warehouse Robot	8	4
Driver Agent	4	2
First Responder Robot	13	10
Elevator Control Agent	12	10

Table 12. Size of the SDM setups in terms of number of predicates and capabilities.

of predicates and capabilities. Description for the cafe server robot is available in Sec. 6.5. Short descriptions of the other four SDM settings are presented below:

Warehouse Robot This SDM setup is implemented using the SOTA stochastic planning system used in the planning literature. This is motivated from *Exploding Blocksworld* setup introduced in the probabilistic track of International Planning Competition (IPC) 2004 (Younes *et al.*, 2005a). It features a robot that has four capabilities: `stack`, `unstack`, `pick`, and `place`. *stack* capability stacks one object on top of another, `unstack` capability removes an object from top of another object, `pick` capability picks up an object from a fixed location, and `place` capability places the object at a fixed location. The setup is non-deterministic as executing some of these capabilities can destroy the object as they might be delicate. Hence even the ground truth does not have 100% success rate in this setup.

Driver Agent This SDM setup is implemented using the SOTA stochastic planning system used in the planning literature. This is motivated from *Tireworld* setup introduced in the probabilistic track of IPC 2004 (Younes *et al.*, 2005a). It consists of a robot moving around multiple locations. The `move-vehicle` capability that takes the SDMA from one location to another can also cause it to get a flat-tire with some non-zero probability. Not all locations have the option to change tire, but if available, a `change-tire` capability will fix the flat-tire with a 100% probability.

First Responder Robot This SDM setup is inspired from *First Responders* in uncertainty track of IPC 2008 (Bryce and Buffet, 2008). The setup features two kinds of emergencies: fire and medical, involving hurt victims. Victims can be treated at the site of an emergency or the hospital. This was originally a FOND setup, and we added probabilities to all the capabilities with non-deterministic effects to make it probabilistic. The responder vehicles can also be driven from one place to another and can be loaded and unloaded with fire or medical kits. The recovery status depending on the treatment location, is different with different probabilities.

Elevator Control Agent This SDM setup is motivated from *Elevators* in the probabilistic track of IPC 2006 (Bonet and Givan, 2005). It consists of an agent managing multiple elevators on multiple floors in a single building. The capabilities of moving from one elevator to another on the same floor are probabilistic. The size of this setup is much larger than the previous three. Also, the capabilities have arities of up to five, making this setup complex from an assessment point of view.

6.5.2 Results

We present an analysis of our approach on all of the SDMAs listed above. We also present a comparative analysis with the baseline on all SDMAs except the Cafe Server Robot, whose task and motion planning system was not compatible with the baseline.

Cafe Server Robot This SDMA setup uses an 8 degrees of freedom Fetch (Wise *et al.*, 2016) robot in a cafe setting on OpenRave simulator (Diankov and Kuffner, 2008). The low-level environment state consists of continuous x, y, z, roll, pitch, and yaw values of all objects in the environment. The predicate evaluators were provided by ATM-MDP of which we used only a subset to learn a PPDDL model. Each robot

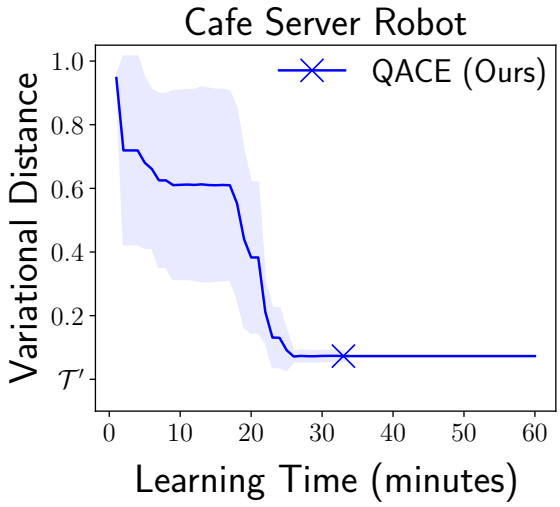


Figure 17. Variational Distance between the learned model and the ground truth with increasing time for QACE for Cafe Server Robot. $\times$ shows that the learning process ended at that time instance.

capability is refined into motion controls at run-time depending on the configuration of the objects in the environment. The results for variational distance between the learned model and the ground truth model in Fig. 17 show that despite the different vocabulary, QACE learns an accurate transition model for the SDMA.

We now discuss the comparative performance of QACE with the baseline across the four baseline-compatible SDMAs presented above.

Faster convergence The time taken for QACE to learn the final model is much lower than that of GLIB for three of the four SDMAs. This is because trace collection by QACE is more directed and hence ends up learning the correct model in *a shorter time*. The only setup where GLIB marginally outperforms QACE is Warehouse Robot, and this happens because this SDMA has just two capabilities, one of which is deterministic. Hence, GLIB can easily learn their configuration from a few observed traces. For SDMAs with complex and much larger number of capabilities – First Responder Robot and Elevator Control Agent – GLIB finds it more challenging to

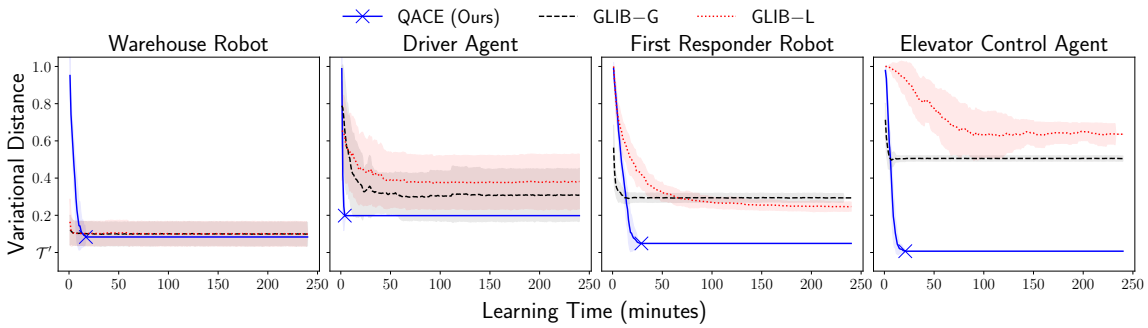


Figure 18. A comparison of the approximate variational distance as a factor of the learning time for the three methods: QACE (ours), GLIB-G, and GLIB-L (lower values better). $\times$ shows that the learning process ended at that time instance for QACE. The results were calculated using 30 runs per method per domain. Solid lines are averages across runs, and shaded portions show the standard deviation. $\mathcal{T}'$ is the ground truth model.

learn the model that is closer to the ground truth transition model. Additionally, QACE takes much fewer samples to learn the model than the baselines. In all settings, QACE is much more *sample efficient* than the baselines as QACE needed at most 4% of the samples needed by GLIB-G to reach the variational distance that it plateaued at. In contrast, GLIB-L started timing out only after processing a few samples for complex SDMAs.

Few-shot generalization To ensure that learned models are not overfitted, our test set contains problems with larger quantities of objects than those used during training. As seen in Fig. 18, the baselines have higher variational distance from the ground truth model for complex SDMA setups as compared to QACE . This shows *better few-shot generalization* of QACE compared to the baselines.

Results w.r.t. environment steps Fig. 19 show a comparison of the approximate variational distance between QACE and the baselines as a factor of the total steps taken in the environment. From the results, it is clear that QACE is able to outperform GLIB while taking far fewer steps in the environment. GLIB-L operates by babbling

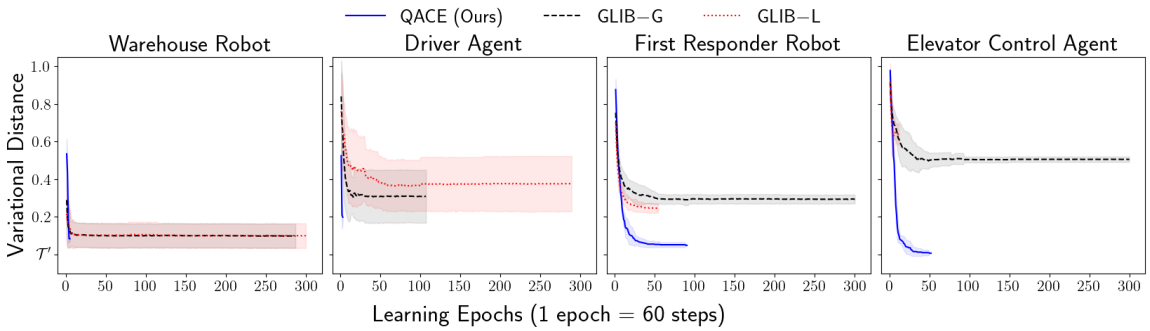


Figure 19. Results showing the trends in the approximate Variational Distance w.r.t. the total number of steps in the environment (lower values better) for the three methods: QACE (ours), GLIB-G, and GLIB-L. Lines which do not extend until the end indicate that the time limit (4 hours) was exceeded. The results were calculated using 30 runs per method per domain. Solid lines are averages across runs, and shaded portions show the standard deviation. $\mathcal{T}'$ is the ground truth model.

lifted goals and we found that the goal babbling step of GLIB-L took an inordinate amount of time leading to very few steps in the environment before the timeout of 4 hours. GLIB-G babbles grounded goals and thus can perform many steps but is not sample efficient in learning as the results show. We analyzed the cause and found that if GLIB-G learns an incorrect model, it is often quite difficult to get out of local minima since it keeps generating and following the same plan.

Evaluation w.r.t. ground truth models $\mathcal{T}'$ Fig. 20 demonstrate that QACE is able to converge to a learned model that is near-perfect compared to the ground truth model $\mathcal{T}'$. QACE is able to learn such a near-perfect model in a fraction of the time compared to the baselines (see Fig. 18 in the main chapter). QACE can learn the non-deterministic effects and preconditions in a finite number of representative environment interactions and given enough samples MLE estimates are guaranteed to converge. This is in stark contrast to GLIB whose learned NDRs cannot be easily compared to the ground truth.

Faster convergence Fig. 21 shows a zoomed in version of the Fig. 18 in the main

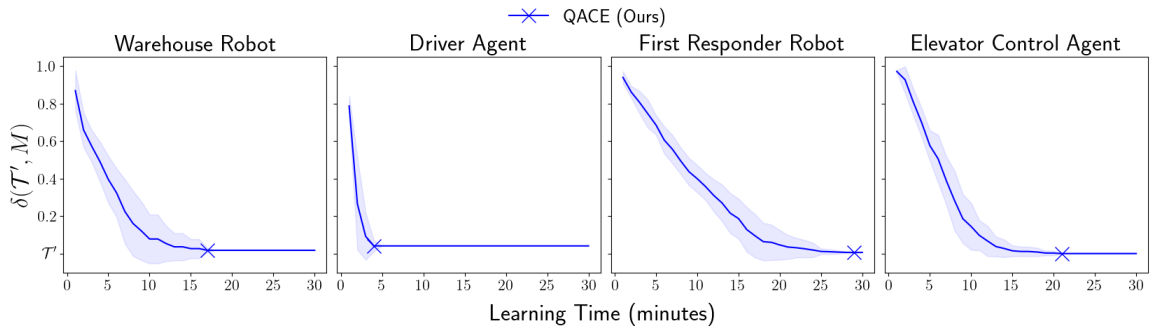


Figure 20. Results showing the comparison of QACE w.r.t. the ground truth model $\mathcal{T}'$. The plots show a trend in the variational distance (see Eq. 1) as a factor of the learning time for QACE (lower values better). $\times$ shows that the learning process ended at that time instance for QACE. The results were calculated using 30 runs per method per domain. Solid lines are averages across runs, and shaded portions show the standard deviation.

chapter. As you notice in the graph, the variational distance is very high initially, and it drops till the learning process of QACE ends (marked by $\times$ on the plots). We do not need to run QACE beyond this point and this time is short for all the domains. On the other hand, GLIB does not have a clear ending criterion. Hence we let it run for 4 hours and see that even with the extra time (and hence extra samples), it cannot learn a better model. The zoomed in plots also show that QACE does not learn the correct model in a one-shot manner, and that it actually keeps getting better with time as it processes more predicate and capability pairs.

Scalability The number of queries needed to learn the model are linear in terms of the number of predicates and capabilities (*for* loop in line 3 of Alg. 5). The total number of queries for each domain shown in Fig. 22 also correlates with the size of the domain shown in Tab. 12, supporting this hypothesis. Note that the *for* loop in line 5 of Alg. 5 only contributes to a constant factor in the running time, as only three models are possible when adding a predicate at a location.

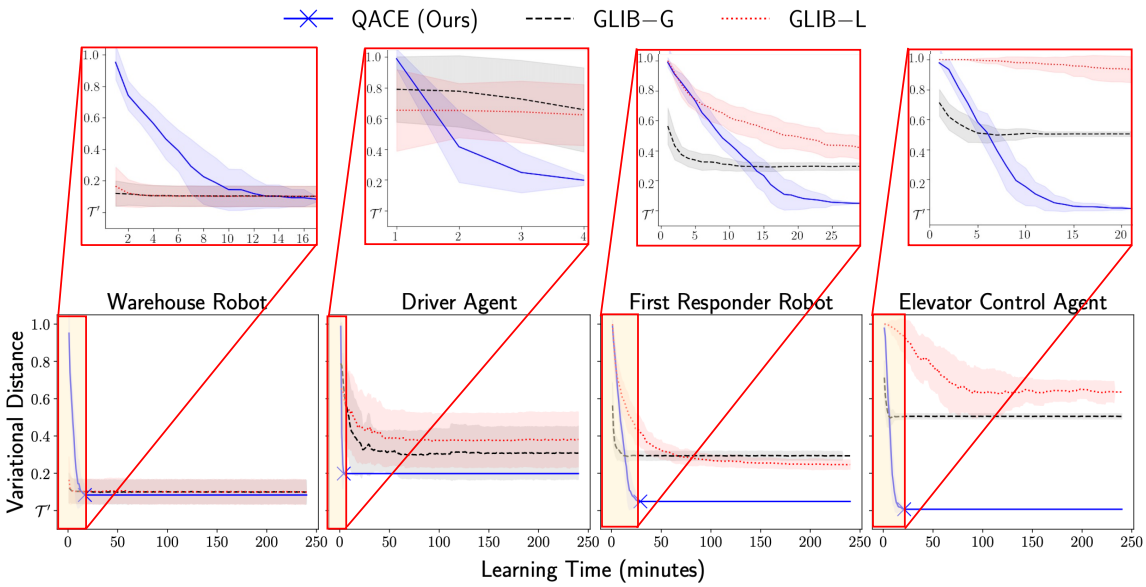


Figure 21. Results showing the comparison of QACE w.r.t. the ground truth model $\mathcal{T}'$. The plots show a trend in the variational distance (see Eq. 1) as a factor of the learning time for QACE (lower values better). $\times$ shows that the learning process ended at that time instance for QACE. The results were calculated using 30 runs per method per domain. Solid lines are averages across runs, and shaded portions show the standard deviation. The zoomed in version shows the plots till learning process for QACE ends (marked using $\times$ in the zoomed-out plots). Note that QACE does not run beyond this.

6.6 Related Work

The problem of learning probabilistic relational agent models from a given set of observations has been well studied (Pasula *et al.*, 2007; Mourão *et al.*, 2012; Martínez *et al.*, 2016; Juba and Stern, 2022). Jiménez *et al.* (2012) and Arora *et al.* (2018) present comprehensive reviews of such approaches. We next discuss the closest related research directions.

Passive learning Several methods learn a probabilistic model of the agent and environment from a given set of agent executions. Pasula *et al.* (2007) learn the models in the form of noisy deictic rules (NDRs) where an action can correspond

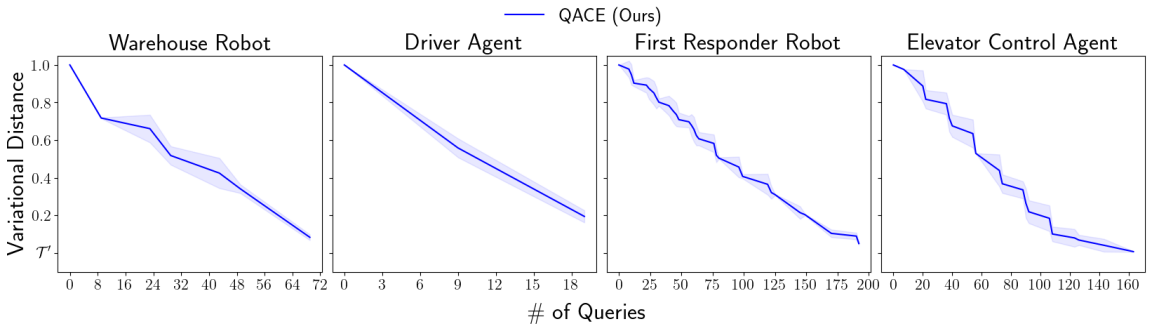


Figure 22. Results showing the avg. number of queries issued by QACE across 30 runs to achieve a specific variational distance (VD). Shaded regions represent one std. deviation. A VD of 0 (zero) corresponds to the ground truth model $\mathcal{T}'$.

to multiple NDRs and also model noise. Mourão *et al.* (2012) learn such operators using action classifiers to predict the effects of an action. Rodrigues *et al.* (2011a) learn non-deterministic models as a collection of rule sets and learn these rule sets incrementally. They take a bound on the number of rules as input. Juba and Stern (2022) provide a theoretical framework to learn safe probabilistic models with a range of probabilities for each probabilistic effect while assuming that each effect is atomic and independent of others. A common issue with such approaches is that they are susceptible to incorrect and sometimes inefficient model learning as they cannot control the input data used for learning or carry out interventions required for accurate learning.

Sampling of transitions Several approaches learn operator descriptions by exploring the state space in the restricted setting of deterministic models (Ng and Petrick, 2019; Jin *et al.*, 2022). A few reinforcement learning approaches have been developed for learning the relational probabilistic action model by exploring the state space using pre-determined criteria to generate better samples (Ng and Petrick, 2019). Konidaris *et al.* (2018) explore learning PPDDL models for planning, but they aim to learn the high-level symbols needed to describe a set of input low-level options, and these

symbols are not interpretable. GLIB (Chitnis *et al.*, 2021) also learns probabilistic relational models using goal sampling as a heuristic for generating relevant data, whereas we use active querying using guided forward search for this. Our empirical analysis shows that our approach of synthesising queries yield greater sample efficiency and correctness profiles than the goal generation used in this approach.

Active learning Several active learning approaches learn automata representing a system's model (Angluin, 1988; Aarts *et al.*, 2012; Pacharoen *et al.*, 2013; Vaandrager, 2017). These approaches assume access to a teacher (or an oracle) that can determine whether the learned automaton is correct and provide a counterexample if it is incorrect. This is not possible in the black-box SDMA settings that constitute the focus of this work.

6.7 Concluding Remarks

In this work, we presented an approach for learning a probabilistic model of an agent using interactive querying. We showed that the approach is few-shot generalizable to larger environments and learns a sound and complete model faster than state-of-the-art approaches in a sample-efficient manner.

QACE describes the capabilities of the robot in terms of predicates that the user understands (this includes novice users as well as more advanced users like engineers). Understanding the limits of the capabilities of the robot can help with the safe usage of the robot, and allow better utilization of the capabilities of the robot. Indirectly, this can reduce costs since the robot manufacturer need not consider all possible environments that the robot may possibly operate in. The use of our system can also be extended to formal verification of SDMAs.

QACE can also be used by standard explanation generators as they need an agent's model. Such models are hard to obtain (as we also illustrate in this chapter) and our approach can be used to compile them when they are not available to start with.

Limitations and Future Work In this work, we assume that the agent can be connected to a simulator to answer the queries. In some real-world settings, this assumption may be limiting as users might not have direct access to such a simulator. Formalizing the conditions under which is it safe to ask the queries directly to the agent in the real-world is a promising direction for the future work.

Chapter 7

CAUSAL ACCURACY

In this chapter, we perform a causal accuracy analysis of the deterministic models we learn in Chapter 3. This analysis also expands to the models learned in other chapter with some restrictions. We introduce dynamic causal decision networks (DCDNs) that capture the causal structure of planning models expressed in STRIPS-like languages.

We now explain how AIA models the causally accurate relationships of the domain.

7.1 Causal Accuracy of the Learned Models

We compare the properties of models learned by AIA with those of approaches that learn the models from observational data only. For the methods that learn models in STRIPS-like the learned models can be classified as causal, but it is not necessary that they are sound with respect to the ground truth model $M^{\mathcal{A}}$ of the agent $\mathcal{A}$. E.g., in case of the robot driver discussed earlier, these methods can learn a model where the precondition of the action drive is `src_blue` if all the observation traces that are provided to it as input had `src_blue` as true. This can happen if all the source locations are painted blue. To avoid such cases, some of these methods run a pre-processing or a post-processing step that removes all static predicates from the preconditions. However, if there is a paint action in the domain that changes the color of all source locations, then these ad-hoc solutions will not be able to handle that. Hence, these techniques may end up learning spurious preconditions as they do not have a way to distinguish between correlation and causation.

On the other hand, it is also not necessary that the models learned by approaches using only observational data are complete with respect to the ground truth model $M^{\mathcal{A}}$ of the agent $\mathcal{A}$. This is because they may miss to capture some causal relationships if the observations do not include all the possible transitions, or contains only the successful actions. E.g., if we have additional predicates `(city_from ?loc)`, and `(city_to ?loc)` in the domain, and all the observed transitions are for the transitions within same city, then the model will not be able to learn if the source city and destination city have to be same for driving a truck between them.

7.2 Causal Models

In this section, we provide an overview of terminology regarding causal implications from Halpern (2015). We will use this framework to show that models learned by our approach are causally accurate.

Definition 27. A *causal model* M is defined as a 4-tuple $\langle U, V, R, F \rangle$ where U is a set of exogenous variables (whose values are determined by factors outside the model), V is a set of endogenous variables (whose values are directly or indirectly derived from the exogenous variables), R is a function that associates with every variable $Y \in U \cup V$ a nonempty set $R(Y)$ of possible values for Y, and F is a function that associates with each endogenous variable $X \in V$ a structural function denoted as F_X such that F_X maps $\times_{Z \in (U \cup V - \{X\})} R(Z)$ to $R(X)$.

Note that the values of exogenous variables are not determined by the model; a setting $\vec{u}$ of values of exogenous variables is termed as a *context* by Halpern (2016). This helps in defining a causal setting as:

Definition 28. Given a setting of exogenous variables $u \in U$, a *causal setting* is a pair $(M, \vec{u})$ consisting of a causal model M and context $\vec{u}$.

A causal formula φ is true or false in a causal model, given a context. Hence, $(M, \vec{u}) \models \varphi$ if the causal formula φ is true in the causal setting $(M, \vec{u})$.

Every causal model M can be associated with a directed graph, $G(M)$, in which each random variable X is represented as a vertex and the causal relationships between these variables are represented as directed edges between members of $U \cup \{V \setminus X\}$ and X Pearl (2009). We use the term causal networks when referring to these graphs to avoid confusion with the notion of causal graphs used in the planning literature (Helmert, 2004).

To perform an analysis with interventions, we use *do-calculus* introduced in Pearl (1995). To perform interventions on a set of variables $X \in V$, do-calculus assigns values $\vec{x}$ to $\vec{X}$, and evaluates the effect using the causal model M. This is termed as $do(\vec{X} = \vec{x})$ action. To define this concept formally, we first define *submodels* (Pearl, 2009).

Definitions 5-8 are by Halpern (2016). These definitions summarize the concepts we use to define and assess the causal accuracy of the learned agent models.

Definition 29. Let M be a causal model, X a set of variables in V, and $\vec{x}$ a particular realization of $\vec{X}$. A *submodel* $M_{\vec{x}}$ of M is the causal model $M_{\vec{x}} = \langle U, V, R, F^{\vec{x}} \rangle$ where $F^{\vec{x}}$ is obtained from F by setting $X' = x'$ (for each $X' \in \vec{X}$) instead of the corresponding $F_{X'}$, and setting $F^{\vec{x}}_{Y} = F_Y$ for each $Y \notin X$.

We now define what it means to intervene $\vec{X} = \vec{x}$ using the action $do(\vec{X} = \vec{x})$. Let M be a causal model, X a set of variables in V, and $\vec{x}$ a particular realization of $\vec{X}$. The effect of action $do(\vec{X} = \vec{x})$ on M is given by the submodel $M_{\vec{x}}$.

In general, there can be uncertainty about the effects of these interventions, leading to probabilistic causal networks, but in this work, we work with fully observable and deterministic settings, hence assume that interventions do not lead to uncertain effects.

We can also derive the structure of causal networks using interventions in the real world, as interventions allow us to find if a variable Y depends on another variable X. We use Halpern (2016)'s notion of dependence as follows.

Definition 30. A variable Y *depends on* a variable X if there is some setting of all the variables in $U \cup V \setminus \{X, Y\}$ such that varying the value of X in that setting results in a variation in the value of Y.

We now use these concepts to define what a causal formula is (Halpern, 2016) and then use it to define what we mean by an actual cause.

Definition 31. Given a signature $\mathcal{S} = (U, V, R)$, a *primitive event* is a formula of the form $X = x$, for $X \in V$ and $x = R(X)$. A *causal formula* is $[\vec{Y} \leftarrow \vec{y}]\varphi$, where φ is a Boolean combination of primitive events, $\vec{Y} = \langle Y_1, Y_2, \ldots Y_i \rangle$ are distinct variables in V, and $y_i \in R(Y_i)$.

$[\vec{Y} \leftarrow \vec{y}]\varphi$ means that φ would hold if Y_k were set to y_k, for $k = 1, \ldots, i$. We next formally define an actual cause.

Definition 32. Let $X \subseteq V$ be a subset of endogenous variables V, and φ be a boolean causal formula expressible using variables in V. $\vec{X} = \vec{x}$ is an *actual cause* of φ in the causal setting $(M, \vec{u})$, i.e., $(\vec{X} = \vec{x}) \overset{(M,\vec{u})}{\rightsquigarrow} \varphi$, if the following conditions hold:

AC1. $(M, \vec{u}) \models (\vec{X} = \vec{x})$ and $(M, \vec{u}) \models \varphi$.

AC2. There is a set $\vec{W}$ of variables in V and a setting $\vec{x}'$ of the variables in $\vec{X}$ such that if $(M, \vec{u}) \models \vec{W} = \vec{w}^*$, then $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}', \vec{W} \leftarrow \vec{w}^*]\neg\varphi$.

AC3. $\vec{X}$ is minimal; there is no strict subset $\vec{X}'$ of $\vec{X}$ such that $\vec{X}' = \vec{x}'$ satisfies conditions AC1 and AC2, where $\vec{x}'$ is the restriction of $\vec{x}$ to the variables in $\vec{X}$.

AC1 mentions that unless both φ and $\vec{X} = \vec{x}$ occur at the same time, φ cannot be caused by $\vec{X} = \vec{x}$. AC2[8] mentions that there exists a $\vec{x}'$ such that if we change a subset $\vec{X}$ of variables from some initial value $\vec{x}$ to $\vec{x}'$, keeping the value of other variables $\vec{W}$ fixed to $\vec{w}^*$, φ will also change. AC3 is a minimality condition which ensures that there are no spurious elements in $\vec{X}$.

In this section, we'll first show a mapping between the STRIPS-like models that we learn and the causal models defined earlier. And then we'll define the causal soundness and completeness of one causal model w.r.t. another causal model, and show that the models learned by Alg. 1 are causally accurate.

7.2.1 Representing Planning Models as Causal Networks

The classical causal model framework used in Def. 27 lacks the temporal elements and decision nodes needed to express the causal relationships in the planning models.

To express actions in the model, we use the decision nodes similar to Dynamic Decision Networks Kanazawa and Dean (1989). To express the temporal behavior of planning models, we use the notion of Dynamic Causal Models (Pearl, 2009) and Dynamic Causal Networks (DCNs) (Blondel *et al.*, 2017). These are similar to causal models and causal networks respectively, with the only difference that the variables in these are time-indexed, allowing for analysis of temporal causal relations between the variables. We also introduce additional boolean variables to capture the executability

[8] Halpern (2016) terms this version of AC2 as AC2(a^m)

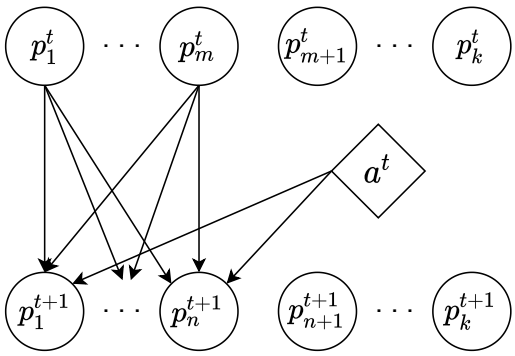


Figure 23. An example of a Dynamic Causal Decision Network (DCDN). p_i^t and p_i^{t+1} are the action-parameter instantiated predicates at time t and $t+1$ respectively and a_t is a decision node representing the decision to execute action the parameterized a at time t.

of the actions. The resulting causal model is termed as a causal action model, and we express such models using a Dynamic Causal Decision Network (DCDN).

A general structure of a dynamic causal decision network is shown in Fig. 23. All the decision variables and the executability variables p_i^t, where $i \in [0, k]$, where k is the number of instantiated predicates, in a domain are endogenous. There is an edge from each predicate in an action's precondition to each predicate in an action's effect. All the variables are endogenous because we can perform interventions on them as needed.

We now show a mapping between the components of the causal models used in Def. 27 and the planning models defined in Def. 1. The exogenous variables $\mathcal{U}$ map to the static predicates (Helmert, 2009) in the domain, i.e., the ones that do not appear in the effect of any action; $\mathcal{V}$ maps to the non-static predicates; $\mathcal{R}$ maps each predicate to $\top$ if the predicate is true in a state, or $\bot$ when the predicate is false in a state; $\mathcal{F}$ calculates the value of each variable depending on the other variables that cause it. This is captured by the values of state predicates and executability variables being changed due to other state variables and decision variables.

7.2.1.1 Causal Soundness and Completeness

Before we prove that the models learned by Alg. 1 are causally correct, we first define the notions of causal soundness and completeness of a pair of models wrt. each other.

Definition 33. Let $\vec{\mathcal{U}}$ and $\vec{\mathcal{V}}$ be the vectors of exogenous and endogenous variables, respectively; and Φ be the set of all boolean causal formulas expressible over variables in $\mathcal{V}$.

A causal model M_1 is *causally complete* with respect to another causal model M_2 if for all possible settings of exogenous variables, the causal relationships that are implied by the model M_1 are a superset of the set of causal relationships implied by the model M_2, i.e., $\forall \vec{u} \in \vec{\mathcal{U}}, \forall \vec{X}, \vec{X'} \subseteq \vec{\mathcal{V}}, \forall \varphi, \varphi' \in \Phi, \exists \vec{x} \in \vec{X}, \exists \vec{x'} \in \vec{X'}$ s.t. $\{\langle \vec{X}, \vec{u}, \varphi, \vec{x} \rangle : (\vec{X} = \vec{x}) \overset{(M_2,\vec{u})}{\rightsquigarrow} \varphi\} \subseteq \{\langle \vec{X'}, \vec{u}, \varphi', \vec{x'} \rangle : (\vec{X'} = \vec{x'}) \overset{(M_1,\vec{u})}{\rightsquigarrow} \varphi'\}$.

A causal model M_1 is *causally sound* with respect to another causal model M_2 if for all possible settings of exogenous variables, the causal relationships implied by M_1 are a subset of the causal relationships implied by M_2, i.e., $\forall \vec{u} \in \vec{\mathcal{U}}, \forall \vec{X}, \vec{X'} \subseteq \vec{\mathcal{V}}, \forall \varphi, \varphi' \in \Phi, \exists \vec{x} \in \vec{X}, \exists \vec{x'} \in \vec{X'}$ s.t. $\{\langle \vec{X}, \vec{u}, \varphi, \vec{x} \rangle : (\vec{X} = \vec{x}) \overset{(M_1,\vec{u})}{\rightsquigarrow} \varphi\} \subseteq \{\langle \vec{X'}, \vec{u}, \varphi', \vec{x'} \rangle : (\vec{X'} = \vec{x'}) \overset{(M_2,\vec{u})}{\rightsquigarrow} \varphi'\}$.

We now show that the model(s) learned by AIA are causally sound and complete.

Theorem 11. Given an agent $\mathcal{A}$ with a ground truth model $M^{\mathcal{A}}$ (unknown to the agent interrogation algorithm AIA), the action model M learned by AIA is causally sound and complete with respect to $M^{\mathcal{A}}$.

Proof. We first show that M is sound with respect to $M^{\mathcal{A}}$. Assume that some $\vec{X} = \vec{x}$ is an actual cause of φ according to M in the setting $\vec{u}$, i.e., $(\vec{X} = \vec{x}) \overset{(M,\vec{u})}{\rightsquigarrow} \varphi$. Now

by Thm 3, M contains palm tuples that are consistent with $M^{\mathcal{A}}$. Hence any palm tuple that is present in M will also be present in $M^{\mathcal{A}}$, implying that under the same setting $\vec{u}$ according to $M^{\mathcal{A}}$ $\vec{X} = \vec{x}$ is an actual cause of φ.

Now lets assume that some $\vec{X} = \vec{x}$ is an actual cause of φ according to $M^{\mathcal{A}}$ in the setting $\vec{u}$, i.e., $(\vec{X} = \vec{x}) \overset{(M^{\mathcal{A}},\vec{u})}{\rightsquigarrow} \varphi$. Now by Thm. 3, M contains exactly the same palm tuples as $M^{\mathcal{A}}$. Hence any palm tuple that is present in $M^a g$ will also be present in M, implying that under the same setting $\vec{u}$ according to M $\vec{X} = \vec{x}$ is an actual cause of φ. Hence the action model M learned by the agent interrogation algorithm are sound and complete with respect to the model $M^{\mathcal{A}}$. □

7.3 Related Work

There have been some recent approaches that learn causal dynamics of a sequential decision-making system (Madumal *et al.*, 2020; Wang *et al.*, 2022, 2024; Nashed *et al.*, 2023; Amitai *et al.*, 2024). Some approaches even use counterfactuals for reasoning Ma *et al.* (2023). These approaches either lack the theoretical guarantees we provide, need extra information about the dependency graph, or are not as scalable as our approach as we increase the domain size.

7.4 Concluding Remarks

We introduced dynamic causal decision networks (DCDNs) to represent causal structures in STRIPS-like domains; and showed that the models learned using the agent interrogation algorithm are causally accurate, and are sound and complete with respect to the corresponding unknown ground truth models. This is an important

step in proving the accuracy of the models as the models that capture the causally accurate information are preferred over the ones which are only empirically accurate.

Chapter 8

QUERY COMPLEXITY ANALYSIS

In this chapter, we provide an extended analysis of the complexity of the queries that AIA uses to learn the agent's model. We use the complexity analysis of relational queries by Vardi [1982, 1995] to find the membership classes for data, expression, and combined complexity of AIA's queries.

8.1 Types of Complexity

Vardi (1982) introduced three kinds of complexities for relational queries. In the notion of query complexity, a specific query is fixed in the language, then *data complexity* – given as function of size of databases – is found by applying this query to arbitrary databases. In the second notion of query complexity, a specific database is fixed, then the *expression complexity* – given as function of length of expressions – is found by studying the complexity of applying queries represented by arbitrary expressions in the language. Finally, *combined complexity* – given as a function of combined size of the expressions and the database – is found by applying arbitrary queries in the language to arbitrary databases.

These notions can be defined formally as follows Vardi (1995):

Definition 34. The complexity of a query is measured as the complexity of deciding if $t \in Q(B)$, where t is a tuple, Q is a query, and B is a database.

- The *data complexity* of a language $\mathcal{L}$ is the complexity of the sets $Answer(Q_e)$

for queries e in $\mathcal{L}$, where $Answer(Q_e)$ is the answer set of a query Q_e given as: $Answer(Q_e) = \{(t, B) \mid t \in Q_e(B)\}$.

- The *expression complexity* of a language $\mathcal{L}$ is the complexity of the sets $Answer_{\mathcal{L}}(B)$, where $Answer_{\mathcal{L}}(B)$ is the answer set of a database B with respect to a language $\mathcal{L}$ given as:
  $Answer_{\mathcal{L}}(B) = \{(t, e) \mid e \in \mathcal{L} \text{ and } t \in Q_e(B)\}$.
- The *combined complexity* of a language $\mathcal{L}$ is the complexity of the set $Answer_{\mathcal{L}}$, where $Answer_{\mathcal{L}}$ is the answer set of a language $\mathcal{L}$ given as:
  $Answer_{\mathcal{L}} = \{(t, B, e) \mid e \in \mathcal{L} \text{ and } t \in Q_e(B)\}$.

Vardi [1982, 1995] gave standard complexity classes for queries written in specific logical languages. We show the membership of our queries in these classes based on the logical languages we write the queries in.

8.2 Action Precondition Queries

In this chapter, we introduce a new class of queries called *action precondition queries* $\mathcal{Q}_{AP}$. These queries, similar to plan outcome queries, are parameterized by s_I and π, but have a different response type.

A response to the action precondition queries can be either of the form "I can execute the plan completely and at the end of it, truck $t1$ is at location $l1$" when the plan is successfully executed, or of the form "I can execute the plan till step ℓ and the action a_ℓ failed because precondition p_i was not satisfied" when the plan is not fully executed. To make the responses consistent in all cases, we introduce a dummy action a_{fail} whose precondition is never satisfied. Hence, the responses are always of the form, "I can execute the plan till step ℓ and the action a_ℓ failed because precondition

p_F was not satisfied". If a_ℓ is a_{fail} and $\ell = len(\pi)$, then we know that the original plan was executed successfully by the agent. Formally, the response θ_{AP} for action precondition queries is a tuple $\langle \ell, p_F \rangle$, where ℓ is the number of steps for which the plan π could be executed, and $p_F \subseteq P$ is the set of preconditions of the failed action a_F. If the plan π cannot be executed fully according to the agent model $\mathcal{M}^{\mathcal{A}}$ then $\ell < len(\pi) - 1$, otherwise $\ell = len(\pi) - 1$. Also, $\mathcal{Q}_{AC} : \mathcal{U} \to \mathbb{N} \times P$, where $\mathcal{U}$ is the set of all the models that can be generated using the predicates P and actions $\mathbb{A}$, and $\mathbb{N}$ is the set of natural numbers.

8.3 Membership Complexity Classes for Queries

Complexity for Plan Outcome Queries Theoretically, the asymptotic complexity of AIA is $O(|\mathbb{P}^*| \times |\mathbb{A}|)$, but it does not take into account how much computation is needed to answer the queries or to evaluate their responses. This complexity just shows the amount of computation needed in the worst case to derive the agent model by AIA. Here, we present a more detailed analysis of the complexity of AIA's queries using the results of relational query complexity by Vardi (1982).

This analysis takes into account the computational effort that the agent will have to put in to answer the queries. This is because we want to have minimal requirements on the agent to support AAM, and hence we don't want to ask questions that are too complex to answer. This analysis will also form a foundation for our future work on comparing different types of queries.

According to the notion of query complexity in Vardi (1982), a specific query is fixed in the language, then *data complexity* – given as function of size of databases – is found by applying this query to arbitrary databases. In the second notion of

query complexity, if a specific database is fixed, then the *expression complexity* – given as a function of the length of expressions – is found by studying the complexity of applying queries represented by arbitrary expressions in the language. Finally, *combined complexity* – given as a function of combined size of the expressions and the database – is found by applying arbitrary queries in the language to arbitrary databases.

Theorem 12. The membership classes of data, expression, and combined complexities of plan outcome queries are AC^0, ALOGTIME, and PTIME respectively.

Proof. To analyze $\mathcal{Q}_{PO}$'s complexity, let us assume that the agent has stored the possible transitions it can make (in propositional form) using the relations $R(\mathit{valid}, s, a, s', \mathit{succ})$, where $\mathit{valid}, \mathit{succ} \in \{\top, \bot\}$, $s, s' \in S$, $a \in A$; and $N(\mathit{valid}, n, n_+)$, where $\mathit{valid} \in \{\top, \bot\}$, $n, n_+ \in \mathbb{N}$, $0 \leq n \leq L$, and $0 \leq n_+ \leq L + 1$, where L is the maximum possible length of a plan in the $\mathcal{Q}_{PO}$ queries. L can be an arbitrarily large number, and it does not matter as long as it is finite. Here, S and A are sets of grounded states and actions respectively. succ is $\top$ if the action was executed successfully, and is $\bot$ if the action failed. valid is $\top$ when none of the previous actions had $\mathit{succ} = \bot$. This stops an action to change a state if any of the previous actions failed, thereby preserving the state that resulted from a failed action. Whenever $\mathit{succ} = \bot$ or $\mathit{valid} = \bot$, $s = s'$ and $n = n_+$ signifying that applying an action where it is not applicable does not change the state.

Assuming the length of the query plan, $\mathit{len}(\pi) = D$, we can write a query in first

order logic, equivalent to the plan outcome query as

$$\{(s_D, n_D) \mid \exists s_1, \ldots, \exists s_{D-1}, \exists succ_1, \ldots, \exists succ_{D-1}, \exists n_1, \ldots, \exists n_{D-1}$$
$$R(\top, s_0, a_1, s_1, succ_1) \wedge R(succ_1, s_1, a_2, s_2, succ_2) \wedge \cdots \wedge R(succ_{D-1}, s_{D-1}, a_D, s_D, \top) \wedge$$
$$N(\top, 0, n_1) \wedge N(succ_1, n_1, n_2) \wedge \cdots \wedge N(succ_{D-1}, n_{D-1}, n_D)\}$$

The output of the query contains the free variables $s_D = s_\ell$ and $n_D = \ell$. Such first order (FO) queries have the expression complexity and the combined complexity in PSPACE (Vardi, 1982). The data complexity class of FO queries is AC^0 (Immerman, 1987).

The following results use the analysis in Vardi (1995). The query analysis given above depends on how succinctly we can express the queries. In the FO query shown above, we have a lot of spurious quantified variables. We can reduce its complexity by using *bounded-variable* queries. Normally, queries in a language $\mathcal{L}$ assume an inifinite supply $x_1, x_2, \ldots$ of individual variables. A *bounded-variable* version $\mathcal{L}^k$ of the language $\mathcal{L}$ is one which can be obtained by restricting the individual variables to be among $x_1, \ldots, x_k$, for $k > 0$. Using this, we can reduce the quantified variables in FO query shown earlier, and rewrite it more succinctly as an FO^k query by storing temporary query outputs.

$$E(succ, s, a, s', succ', n, n') = R(succ, s, a, s', succ') \wedge N(succ, n, n')$$
$$\alpha_1(succ, s, a_1, s', succ', n, n') = E(\top, s_0, a_1, s', succ', 0, n')$$

We then write subsequent queries corresponding to each step of the query plan as

$$\begin{aligned}
&\alpha_{i+1}(succ, s, a_{i+1}, s', succ', n, n') = \\
&\quad \exists s_1, \exists succ_1, \exists n_1 \{E(succ, s, a_{i+1}, s_1, succ_1, n_1) \wedge \\
&\quad\quad \exists s, \exists succ, \exists n [succ = succ_1 \wedge s = s_1 \wedge \\
&\quad\quad\quad n = n_1 \wedge \alpha_i(succ, s, a_i, s', succ', n, n')]\}
\end{aligned}$$

Here i varies from 1 to D, and the value of k is 6 because of 6 quantified variables – $s, s_1, succ, succ_1, n$, and n_1. This reduces the expression and combined complexity of these queries to ALOGTIME and PTIME respectively. Note that these are the membership classes as it might be possible to write the queries more succinctly. □

Complexity for Action Precondition Queries For a detailed analysis of $\mathcal{Q}_{AP}$'s complexity, let us assume that the agent stores the possible transitions it can make (in propositional form) using the relations $R(valid, s, a, s', succ)$, where $valid, succ \in \{\top, \bot\}$, $s, s' \in S$, $a \in A$; and $\mathcal{S}(p, s)$, where $p \in P$, $s \in S$. $\mathcal{S}$ contains (p, s) if a grounded predicate p is in state s.

Now, we can write a query in first order logic, equivalent to the action precondition query as:

$$\begin{aligned}
&\{(p) \mid (\forall s_1 \, \mathcal{S}(p, s_1) \Rightarrow \exists s' R(\top, s_1, a_1, s', \top)) \wedge \\
&\quad (\forall s_1 \, \neg \mathcal{S}(p, s_1) \Rightarrow \forall s' R(\top, s_1, a_1, s', \bot))\}
\end{aligned}$$

This formulation is equivalent to the FO^k queries with $k = 2$. This means that the data, expression and combined complexity of these queries are in complexity classes AC^0, ALOGTIME, and PTIME respectively.

8.4 Concluding Remarks

The results for complexity classes of the queries presented in this chapter holds assuming that the agent stores all the transitions using a mechanism equivalent to relational databases where it can search through states in linear time. For the simulator agents that we generally encounter, this assumption almost never holds true. Even though both the queries have membership in the same complexity class, an agent will have to spend more time in running the action precondition query owing to the exhaustive search of all the states in all the cases, whereas for the plan outcome queries, the exhaustive search is not always needed.

Additionally, plan outcome queries place very little requirements on the agent to answer the queries, whereas action precondition queries require an agent to use more computation to generate it's responses. Action precondition queries also force an agent to know all the transitions beforehand. So if an agent does not know its model but has to execute an action in a state to learn the transition, action precondition queries will perform poorly as agent will execute that action in all possible states to answer the query. On the other hand, to answer plan outcome queries in such cases, an agent will have to execute at most L actions (maximum length of the plan) to answer a query.

Evaluating the responses of queries will be much easier for the action precondition queries, whereas evaluating the responses of plan outcome queries is not straightforward, as discussed in Verma *et al.* (2021a). As mentioned earlier, the agent interrogation algorithm that uses the plan outcome queries has asymptotic complexity $O(|\mathbb{P}^*| \times |\mathbb{A}|)$ for evaluating all agent responses. On the other hand, if an algorithm is implemented with action precondition queries, its asymptotic complexity for evaluating all agent

responses will reduce to $O(|\mathbb{A}|)$. This is because AAM needs to ask two queries for each action. The first query in a state where it is guaranteed that the action will fail, this will lead AAM to learn the action's precondition. After that AAM can ask another query in a state where the action will not fail, and learn the action's effects. This will also lead to an overall less number of queries.

So there is a tradeoff between the computation efforts needed for evaluation of query responses and the computational burden on the agent to answer those queries.

Chapter 9

APPLICATION TOWARDS AGENT IMPROVEMENT

This chapter introduces an application of the algorithms developed as part of assessment to improve the agent itself. This new approach for continual planning and model learning in non-stationary stochastic environments is expressed using relational representations. Such capabilities are essential for the deployment of sequential decision-making systems in the uncertain, constantly evolving real world. Working in such practical settings with unknown (and non-stationary) transition systems and changing tasks, the proposed framework models gaps in the agent's current state of knowledge and uses them to conduct focused, investigative explorations. Data collected using these explorations is used for learning generalizable probabilistic models for solving the current task despite continual changes in the environment dynamics. Empirical evaluations on several benchmark domains show that this approach significantly outperforms planning and RL baselines in terms of sample complexity in non-stationary settings. Theoretical results show that the system reverts to exhibit desirable convergence properties when stationarity holds.

9.1 Overview

This chapter addresses the problem of planning in non-stationary stochastic settings with unknown domain dynamics. In particular, we consider problems where a goal-oriented agent is not given a closed-form model of the probabilities of states that may result upon execution of an action. Furthermore, these probabilities can change

at potentially unknown time steps as the agent is executing in the environment. Such settings are commonly encountered by planning systems in the real-world. For example, an autonomous warehouse robot would be expected to continue achieving goals through different paths when some corridors get blocked due to spills or when the layouts of storage racks change to accommodate changing inventory profiles. Currently, such changes require renewed modeling by domain experts thus limiting the scope and deployability of automated planning methods.

These settings are technically challenging due to the need to correctly model uncertainty about the agent's knowledge when a discrepancy is detected, and to conduct focused exploration that can improve the agent's knowledge for subsequent planning. Prior work on the problem investigates the role of randomized exploration for addressing non-stationarity. E.g., if the rate of *novelty* events inducing non-stationarity are sufficiently low compared to the timesteps available for learning in each epoch of stationary dynamics, Reinforcement Learning (RL) techniques such as Q-Learning with variations of ϵ-greedy exploration can be guaranteed to successively converge to optimal policies. However, these methods are likely to be sample-inefficient as the collection of new data is not easily focused towards parts of the environment that changed.

We present a new framework for continual learning and planning under non-stationarity for such settings (Sec. 9.3.2), develop solution algorithms for this paradigm (Sec. 9.3.4) and evaluate their performance across various forms of the problem, depending on whether the change in dynamics is known to the agent and whether the agent conducts comprehensive re-learning or need-based learning (Sec. 9.4).

Our approach addresses the challenges discussed above with autonomous processes for deliberative data gathering, planning, and model learning. It starts with the inputs

available to a standard RL agent (a simulator, action names, and a reward generator), but instead of learning a policy, it interacts with the environment to first learn a relational probabilistic planning model geared towards solving the current goal, and then uses it to compute solution policies. When a discrepancy is detected, it flags aspects of the currently learned model that are no longer accurate, and conducts investigative exploration with auto-generated epistemically-guided policies to re-learn aspects that may have changed. The problem of computing useful investigative policies is non-trivial. This is reduced to a fully-observable non-deterministic (FOND) (Cimatti *et al.*, 1998) planning problem and solved without interacting with the simulator. The computed investigative policies are then executed and the resulting data is used to learn more accurate models. Although these executions are not focused on policy learning for the current task, they are used to learn and maintain relational Probabilistic Planning Domain Description Language (PPDDL) style models. We show that (i) this significantly increases transferability and generalizability of learning, and (ii) the resulting paradigm vastly outperforms SOTA RL and existing model-based RL paradigms.

Our main contribution is the first known approach for using information about epistemic uncertainty of a logic-based internal probabilistic model to create exploration strategies, learn better models, and then compute plans even as transition systems change. Additionally, this is also the first approach to interleave *active learning* with epistemic exploration to discover a stochastic symbolic model suited for task transfer in non-stationary environments. Empirical analysis on non-stationary versions of benchmark domains show that in such settings our approach (i) significantly reduces the sample complexity compared to SOTA baselines; (ii) can quickly adapt to changes

in environment dynamics; and (iii) performs very close to an oracle that has access to all the information about changes in the environment apriori.

9.2 Preliminaries

Relational Markov Decision Processes (RMDPs) We model tasks as RMDPs expressed in PPDDL (Younes *et al.*, 2005b). An RMDP environment or *domain* $\mathcal{D}^{\uparrow} = \langle \mathcal{P}^{\uparrow}, \mathcal{A}^{\uparrow} \rangle$ is a tuple consisting of a set of parameterized predicates $\mathcal{P}^{\uparrow}$ and actions $\mathcal{A}^{\uparrow}$. Here, $\mathcal{P}^{\uparrow}$contains predicates of the form $p^{\uparrow}(x_1, \ldots, x_m)$, and $\mathcal{A}^{\uparrow}$contains actions of the form $a^{\uparrow}(x_1, \ldots, x_n)$, where x_i are the *parameters*. We use $\uparrow$ to specify lifted predicates and actions with variables as arguments and omit the parameterization when it is clear from context. A *grounded* RMDP task (or problem) is defined as a tuple $M = \langle \mathcal{D}^{\uparrow}, O, S, A, \delta, R, s_0, g, \gamma \rangle$ where O is a set of objects. A literal $p(o_1, \ldots, o_n)$ represents a grounded predicate parameterized with objects $o_i \in O$. Formally, predicates are grounded by computing a mapping between their parameters to the objects, $\sigma(p^{\uparrow}(x_1, \ldots, x_n), [o_1, \ldots, o_n]) = p(o_1, \ldots, o_n)$, where $p^{\uparrow} \in \mathcal{P}^{\uparrow}$, $o_i \in O$. Similarly, σ can also be used to lift grounded predicates and actions. We refer to $\mathcal{P}$ as the set of all possible grounded predicates derivable using $\mathcal{P}^{\uparrow}$ and O. For clarity, we use the notation $e^{\uparrow}$ to denote whether an entity e is lifted and use e otherwise.

A state s is a complete valuation of all possible predicates $p \in \mathcal{P}$. Following the closed-world assumption, predicates whose values are false are omitted from the state representation. The set of all possible subsets of predicates forms the state space S of the RMDP M. Similarly, the action space A of M is formed by grounding each action $a^{\uparrow} \in \mathcal{A}^{\uparrow}$. $\delta : S \times A \times S \to [0, 1]$ is the transition function and is implemented by a simulator. For a given *transition* $\tau = (s, a, s')$, $\delta(s, a, s')$ specifies the probability

of executing action $a \in A$ in a state $s \in S$ and reaching a state $s' \in S$. Naturally, $\sum_{s' \in S} \delta(s, a, s') = 1$ for any $s \in S$ and $a \in A$.

The simulator $\Delta : S \times A \to S$ is a function that returns a state s' on executing a in s by sampling over δ. Executing an action $\Delta(s, a)$ constitutes one *step* on the simulator. $|\Delta|$ represents the total steps executed by the simulator and $\Delta_{\mathcal{S}} \in \mathbb{N}^+$ indicates the simulator step budget after which the simulator cannot be used. s_0 is the initial state and g is a conjunctive first-order logic goal formula obtained using $\mathcal{P}^{\uparrow}$and O. A goal state $s_g \in S$ is a state such that $s_g \models g$. $R : S \times A \to \{0, -1\}$ is the reward function and $R(s, a)$ indicates the reward obtained for executing action a in state s. For all $a \in A$, we set $R(s_g, a) = 0$ for any goal state s_g and $R(s, a) = -1$ otherwise. $\gamma \in [0, 1)$ is the discount factor. Execution begins in the initial state and terminates when a goal state is reached or when a horizon $H \in \mathbb{N}^+$ has been exceeded. An RMDP task is *accomplished* whenever execution terminates in a goal state.

Running Example Consider a robot that is deployed to assist in a warehouse. The robot is equipped with sensors and actuators (e.g., camera, wheels, grippers, etc.) that can help it perform a variety of tasks such as cleaning floors, restocking shelves, etc. Such tasks could be specified by using a domain with $\mathcal{P}^{\uparrow} = \{\texttt{robot-at}^{\uparrow}(r_x, l_x), \texttt{box-at}^{\uparrow}(l_x, b_x), \texttt{holding}^{\uparrow}(r_x, b_x), \texttt{handempty}^{\uparrow}(r_x)\}$. $\mathcal{A}^{\uparrow}$ would consist of actions such as $\texttt{move-from}^{\uparrow}(r_x, l_x, l_y), \texttt{pick-up}^{\uparrow}(r_x, l_x, b_x)$, etc. with their transition function implemented by a simulator.

Example RMDP task Consider an environment with one robot r_1, two locations l_1, l_2, and one box b_1. An RMDP task of moving b_1 to l_1 and parking r_1 anywhere could be modeled as M where $O = \{r_1, l_1, l_2, b_1\}$, $s_0 = \{\texttt{handempty}(r_1), \texttt{robot-at}(r_1, l_1), \texttt{box-at}(b_1, l_2)\}$, and $g = \texttt{box-at}(b_1, l_1) \wedge \exists l_x \ \texttt{robot-at}(r_1, l_x)$.

A solution to an RMDP is a *deterministic policy* $\pi : S \to A$ that maps states to

actions. The value of a state s when following a policy π is defined as the expected cumulative reward obtained when executing a in s and following π thereafter, i.e., $V^{\pi}(s) = R(s,a) + \gamma \sum_{s' \in S} \delta(s,a,s') V^{\pi}(s')$ The objective of an RMDP is to compute an *optimal* policy π^* that maximizes the expected reward obtained by following it.[9] Model-based RMDP algorithms compute $\pi^*(s_0)$ by solving the *Bellman Optimality Equation* iteratively starting from s_0 (Sutton and Barto, 1998):

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s' \in S} \delta(s,a,s') V^*(s') \right] \tag{9.1}$$

The above equation requires access to closed-form knowledge of the transition function δ. When such information is unavailable, RL-based RMDP algorithms use sample estimates of Q-values instead. Given a policy π, the Q-value of a state s when executing action a is defined as the expected reward obtained when executing a in s and following π thereafter, i.e. $Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t) | S_0 = s, A_0 = a\right]$. The Q-Learning Equation (Watkins, 1989) can be written as: as:

$$Q(s,a) = (1-\alpha) Q(s,a) + \alpha \left[R(s,a) + \gamma \max_{a' \in A} Q(s',a') \right]$$

where $\alpha \in [0,1]$ is the learning rate. It employs an exploration strategy such as ϵ-greedy wherein a random action is selected with probability ϵ and selecting the greedy action $\arg\max_a Q(s,a)$ otherwise. Q-Learning has been shown to converge to the optimal policy (Sutton and Barto, 1998).

PPDDL transition models Our approach learns lifted PPDDL models that can be used for stochastic planning using Eqn. 9.1. We note that the simulator's implementation of the transition function could be arbitrary and does not need to be a PPDDL model. Given an RMDP M, a PPDDL model $\mathcal{M}_a$ for an action $a(o_1, \ldots, o_n) \in A$ is

[9] Without loss of generality, we focus on optimal policies that are optimal w.r.t. the initial state s_0.

a tuple $\langle Pre_a, Prob_a, Eff_a \rangle$. We omit the subscript when it is clear from context. *Pre* represents the precondition and is expressed as a conjunctive formula of predicates $p \in \mathcal{P}_a$ where $\mathcal{P}_a = \{\sigma(p^\uparrow(x_1, \ldots, x_m), [o_i, \ldots, o_j] | p^\uparrow \in \mathcal{P}^\uparrow\}$. *Prob* is a list of probabilities such that $\sum_i Prob[i] = 1$. *Eff* is a list of effects. Each effect $Eff[i] \in Eff$ is a tuple $\langle Eff[i]^-, Eff[i]^+ \rangle$ both of which are sets composed of predicates $p \in \mathcal{P}_a$.

An action a is *applicable* in a state s iff $s \models Pre$. An effect $Eff[i]$ when applied to a state s results in a state $s \setminus Eff[i]^- \cup Eff[i]^+$. Applying an action a to a state s results in exactly one effect $Eff[i]$ being applied with probability $Prob[i]$ if the action is applicable else the state remains unchanged. A PPDDL transition model $\mathcal{M} = \{\mathcal{M}_a | a \in A\}$ translates to a closed-form specification of the transition function δ of M, i.e., $\mathcal{M} \equiv \delta$. A lifted (grounded) PPDDL model $\mathcal{M}_{a^\uparrow}(\mathcal{M}_a)$ can be easily obtained from $\mathcal{M}_a(\mathcal{M}_{a^\uparrow})$ using σ. As is the case with RMDP domains, several RMDP tasks from a single domain can also share the same lifted PPDDL model $\mathcal{M}^\uparrow = \{\mathcal{M}_{a^\uparrow} | a \in \mathcal{A}^\uparrow\}$.

Example The $\texttt{pick-up}^\uparrow(r_x, l_x, b_x)$ action described in the running example could be modeled as a PPDDL model $\mathcal{M}_{pick\text{-}up^\uparrow}$ with precondition $Pre = \texttt{box-at}^\uparrow(b_x, l_x) \wedge \texttt{robot-at}^\uparrow(r_x, l_x) \wedge \texttt{handempty}^\uparrow(r_x)$ to indicate that the action is applicable only when the robot is not holding anything is at the same location as the box. The effects could be modeled as $Eff[0] = \langle \{\neg\texttt{box-at}^\uparrow(b_x, l_x), \neg\texttt{handempty}^\uparrow(r_x)\}\ \{\texttt{holding}^\uparrow(r_x, b_x)\} \rangle$ to indicate that the robot successfully picked up the box and is currently holding it. Similarly, another effect $Eff[1] = \{\}$ with $Prob[1] = 0.1$ could be used to model a slippery gripper with a 10% chance to fail to pick-up the box.

Definition 35 ($\mathcal{M}$-Consistent Transition)**.** Given a PPDDL model $\mathcal{M}$ and an action $a(o_1, \ldots, o_n) \in A$ of an RMDP M, a transition $\tau = (s, a, s')$ where $s, s' \in S$ is said to be $\mathcal{M}$-consistent, $\tau \rightleftharpoons \mathcal{M}$, iff $s = s'$ when $s \not\models Pre$ or $\exists i$ such that $Prob[i] > 0$ and $s' = s \setminus Eff[i]^- \cup Eff[i]^+$ whenever $s \models Pre$.

A lifted PPDDL model $\mathcal{M}_{a^\uparrow}$ is implicitly converted to a grounded PPDDL model $\mathcal{M}_{\sigma(a^\uparrow,o_1,\ldots,o_n)}$ when checking for $\mathcal{M}$-consistency w.r.t. a transition τ.

PPDDL Model-Learning Given a dataset $\mathcal{T}$ that is composed of a set of transitions $\tau = (s, a, s')$ obtained from an RMDP task, the PPDDL model-learning problem is to compute a model $\mathcal{M}$ s.t. $\tau \rightleftharpoons \mathcal{M}$ for any $\tau \in \mathcal{T}$. The two major techniques of model learning are active and passive learning. Active learners interactively explore the state space to generate $\mathcal{T}$ for learning the model whereas passive learners require $\mathcal{T}$ to be provided as input. We use active learning as it has been shown to work well for deterministic, non-stationary settings (Nayyar *et al.*, 2022).

9.3 Our Approach

We now begin by describing the problem that we address, followed by a detailed overview of our approach.

Definition 36 (RMDP equivalence)**.** Given a domain $\mathcal{D}^\uparrow$ and RMDP tasks M_i and M_j derived using $\mathcal{D}$, we define $M_i = M_j$ iff their objects are the same $O_{M_i} = O_{M_j}$, the initial state and goals are equal $s_{o_{M_i}} = s_{o_{M_j}}$ and $g_{M_i} = g_{M_j}$, and the transition systems are equivalent $\delta_{M_i} = \delta_{M_j}$.

Definition 37 (Continual Planning under Non-Stationarity)**.** Given a stream of RMDP tasks $\overline{M} = \langle M_1, \ldots, M_n \rangle$ where $M_i \neq M_{i+1}$, a simulator Δ with budget $\Delta_{\mathcal{S}}$ per task, and with the simulators transition system changing at arbitrary intervals, the objective is to maximize the total tasks accomplished within $|\overline{M}|\Delta_{\mathcal{S}}$.

The above problem setting captures many real-world scenarios where environment dynamics often change *in situ*, i.e., while the agent is actively solving a stream of tasks

and without informing the agent. E.g., events like liquid spills on the gripper affecting its friction, navigation pathways being blocked, etc. are outside the robot's control and can arbitrarily change at any given moment. Implicitly, this translates to the agent indirectly optimizing a new RMDP task with the same goal but different transition system. The overall objective is to enable solving all tasks in a sample-efficient fashion thus making it essential to learn-and-transfer knowledge. An agent that learns a fixed model of the environment or one that is incapable of detecting such change can thus perform quite poorly or dangerously.

We consider the following taxonomy of the methods for continual planning under non-stationarity; (a) Adaptive vs. Non-adaptive learners where adaptive learners can automatically adapt to unknown changes in the transition system, whereas the other cannot; (b) Comprehensive vs. Need-based learners where the former completely learn a new model from scratch whereas the latter only perform updates to fix the model w.r.t. transitions that are not $\mathcal{M}$-consistent.

9.3.1 Adaptive Model Learning

Our approach integrates planning and learning by continually learning and updating a PPDDL model of the environment and using it to accomplish tasks. We develop an active, need-based learner that automatically detects and adapts to changes in the transition system. Our approach actively monitors simulator execution and performs active learning when transitions are inconsistent with the current model. We maintain sample efficiency by performing directed exploration while learning the model. We now describe the components that facilitate continual learning for planning.

Active Query-based Model Learning (AQML) We use an active learning ap-

proach as it can cope with non-stationarity. Existing approaches using active learning are sample inefficient since they are comprehensive learners that relearn from scratch. Building upon the Active Query-based Model Learning framework (AQML) (Verma *et al.*, 2023a), we develop a paradigm that can work in the presence of non-stationarity.

Definition 38 (Policy Trace)**.** Given an RMDP M and simulator Δ, a policy trace $\Delta_\pi = \langle s_0, a_0, \ldots, a_{n-1}, s_n \rangle$ of a policy π is a sequence of states and actions where $s_i \in S, a_i \in A$ s.t. $a_i = \pi(s_i)$ and $s_{i+1} = \Delta(s_i, a_i)$.

Definition 39 (p-distinguishing policies)**.** Given an RMDP M, a predicate p, policies π_1, π_2 and a simulator Δ, π_1 and π_2 are p-distinguishing policies iff $\exists i$ s.t. for policy traces Δ_{π_1} and Δ_{π_2}, $p \in s_i^{\Delta_{\pi_1}}$ and $p \notin s_i^{\Delta_{\pi_2}}$.

AQML is an epistemic method that seeks to prune the space of models under consideration by guiding exploration towards states that can help update the model. The key observation is that for any given $a^\uparrow \in \mathcal{A}^\uparrow$, a predicate $p^\uparrow$ can appear as a positive precondition, a negative precondition, or not appear at all in $\mathcal{M}_{a^\uparrow}$. Similarly, $p^\uparrow$ could appear in any of these modes in any of the effect lists of $\mathcal{M}_{a^\uparrow}$. This induces an exponentially large number of models over which a model-learner must search. We can prune this search space by selecting a predicate $p^\uparrow$ and generating candidate models $\mathcal{M}_{a^\uparrow}^{+p_{(Pre|Eff)}}$ $\mathcal{M}_{a^\uparrow}^{-p_{(Pre|Eff)}}$ $\mathcal{M}_{a^\uparrow}^{\otimes p_{(Pre|Eff)}}$ where $p^\uparrow$ appears in a positive ($+$), negative ($-$), or absent ($\otimes$) mode in the preconditions $Pre^\uparrow$ or effects $Eff^\uparrow$ respectively. Ignoring probabilities, AQML uses a combination of any two pairs of these models, and *reduces* query synthesis to a Fully Observable Non-Deterministic (FOND) problem. The central idea behind this reduction is that the two models being used correspond to two separate copies of each predicate in the FOND problem, and a solution is found when a state is reached such that the two copies of predicates do not match. This problem

can be passed to off-the-shelf solvers and the solution to these FOND problems are policies that AQML uses as *queries* to the planning agent. Due to the nature of these models where only a single predicate is changed, solution policies of any pair of these models are guaranteed to be p-distinguishing or unsolvable. AQML then checks which model of the predicate $p^{\uparrow}$ is consistent with the simulator and updates $\mathcal{M}_{a^{\uparrow}}$ appropriately (either in preconditions or one of the effects). The process repeats for the next predicate $p'^{\uparrow}$ with the difference being that the learned information about $p^{\uparrow}$ can now be considered by the FOND planner in the subsequent learning process.

Example Upon identifying that $\neg\texttt{handempty}^{\uparrow}(r_x)$ is an effect of the $\texttt{pick-up}^{\uparrow}(r_x, l_x, b_x)$ action, AQML can generate distinguishing queries by using a FOND planner to resolve other uncertainties such as whether $\neg\texttt{handempty}^{\uparrow}(r_x)$ is a precondition of $\texttt{put-down}^{\uparrow}(r_x, l_x, b_x)$. AQML does this by generating two abstract models, one with predicate $\texttt{handempty}^{\uparrow}(r_x)$ in the precondition of $\texttt{put-down}^{\uparrow}(r_x, l_x, b_x)$, and another where it is absent. As part of the policy generated by the FOND planner it would be ensured that $\neg\texttt{handempty}^{\uparrow}(r_x)$ is true in the state before executing the `put-down` action (possibly by executing a pick-up action).

The key insight is that unlike other methods, this learning methodology does not wait for random exploration to generate p-distinguishing policies but rather actively encourages exploration by utilizing information about parts of the model that are inaccurate. We discuss how such components are annotated in Sec. 9.3.2. This leads to improved sample efficiency in converging to a model $\mathcal{M} \equiv \delta$, i.e., $\mathcal{M}$ translates to a closed-form specification of the transition function δ. Once a p-distinguishing policy is identified, probabilities can be estimated using Maximum Likelihood Estimation (MLE) by executing the policy η times where η is a configurable hyperparameter that represents the sampling frequency.

There are two difficulties with vanilla AQML. Firstly, complete models are learned in a single pass in order to guarantee correctness. Secondly, this framework assumes stationarity of the simulator and the query synthesis process is not resilient to changing environment dynamics during the model-learning loop. As a result, AQML cannot efficiently use learned information to update the model when only small parts of the transition system change.

9.3.2 Non-stationarity Aware Model Learning

We significantly alter the AQML framework so that it can work even if the transition system changes during the model-learning process (as policy traces are being generated using the simulator) and enable it to selectively and correctly learn information that is not consistent with the learned model. We accomplish this by always monitoring executions of the simulator. If a transition $\tau = (s, a, s')$ is not consistent w.r.t. the learned model $\mathcal{M}$, i.e., $\tau \not\models \mathcal{M}$, then we simultaneously update the model-learning process since a new query now needs to be synthesized that can resolve the inconsistency. To do so, we identify the predicates $p^{\uparrow}$ in the preconditions (or effects) of a that were inconsistent with the model and then we add $p^{\uparrow}$ in the precondition (or effect) of a to be relearned. This also applies to inconsistencies identified as policy traces are being generated as a part of the model-learning process. The new FOND problem will not include $p^{\uparrow}$ in the action a in any form in its precondition (or effect) and thus the planner will need to compute an alternate solution for the current query.

Example If a predicate $\neg p^{\uparrow} \in Pre_a$, $p \in s$ and $s \neq s'$ then this means that the action executed successfully on the simulator and the precondition $\neg p^{\uparrow}$ is incorrectly

represented in the currently learned model $\mathcal{M}_a^{\uparrow}$ and must be relearned. We then add $\mathcal{M}_a^{+l_{pre}}$ and $\mathcal{M}_a^{\otimes l_{pre}}$ to the list of models that need to be considered again by the query-synthesis process.

9.3.3 Goodness of Fit Tests

Another key difficulty when operating in non-stationary environments is when the transitions themselves are consistent w.r.t. the preconditions and effects but are drawn from a significantly different distribution. For example, two models of an action with similar preconditions and effects but differing only in the probabilities of effects can impact the ability of an agent to solve a task.

Example In our running example of a slippery gripper, as the probability of slippage increases, the optimal policy might switch to navigating to a human operator and communicating to them to pick up the object.

Such changes cannot be quickly reflected if only MLE estimates are used to compute probabilities since these estimates can be slow to adapt to the new distribution. We mitigate this by including *goodness-of-fit* tests in the planning and learning loop that actively invigilate whether the distributions have undergone shift and can promptly restart the MLE estimation process.

We use Pearson's chi-square test (Pearson, 1992) for detecting o.o.d. effects as follows. Once a model $\mathcal{M}_{a^{\uparrow}}$ for an action has been learned (or a new task is specified), we initialize a table entry $\mathit{Freq}_{a^{\uparrow}}[i] = 0$ for each effect $\mathit{Eff}[i] \in \mathcal{M}_{a^{\uparrow}}$. Whenever a new $\mathcal{M}$-consistent transition (s, a, s') is obtained using the simulator, we identify the index i s.t. $s' = s \setminus \mathit{Eff}[i]^{-} \cup \mathit{Eff}[i]^{+}$. We then increment $\mathit{Freq}_{a^{\uparrow}}[i]$ and perform a goodness of fit test using Pearson's chi-square test with 0 degrees of freedom.

$$\chi^2 = \sum_{i=1}^{n} \frac{(Freq_{a^\uparrow}[i] - F \times Prob_{a^\uparrow}[i]))^2}{F \times Prob_{a^\uparrow}[i]}$$

where $F = \sum_{i=1}^{n} Freq_{a^\uparrow}[i]$ is total observed frequency for a. If the confidence computed using χ^2 is less than some threshold θ (0.05 in our experiments), the goodness-of-fit test is deemed to have failed and we reset the probabilities for all effects in a. To ensure that we have enough samples, we only perform this test when $F > 100$. We then update the probabilities using the recorded frequencies via MLE, i.e., $Prob_{a^\uparrow}[i] = \frac{Freq_{a^\uparrow}[i]}{F}$.

9.3.4 Continual Learning and Planning (CLaP)

Our approach of continual learning of PPDDL models has two key advantages. Firstly, since we learn models, Eqn. 9.1 can be used to compute policies for the task without needing to collect experience from the simulator. Secondly, lifted PPDDL models are *generalizable* in that they can be zero-shot transferred to tasks with differing object names, quantities, and/or goals. For example, the same $\texttt{pick-up}^\uparrow(r_x, l_x, b_x)$ action described earlier can be reused by different RMDP tasks with differing numbers of robots, locations, and/or packages. This methodology allows our approach to solve tasks efficiently.

Alg. 6 describes our overall process for continual learning and planning. The algorithm takes as input an RMDP task M, a simulator Δ, a simulator budget $\Delta_{\mathcal{S}}$, a learned model $\mathcal{M}^\uparrow$, and hyperparameters H, η, β, and θ representing the horizon, sampling count, failure threshold, and confidence threshold respectively. Note that in the context of Alg. 6, M only specifies the initial state s_0 and goal g for the task. The transition system represented by the simulator can arbitrarily change at any time but the agent still perceives it as the same task. Alg. 6 attempts to compute a policy π

Algorithm 6: Continual Learning and Planning

Input : RMDP M, Simulator Δ, Simulator Budget $\Delta_{\mathcal{S}}$, Learned Model $\mathcal{M}^{\uparrow}$, Horizon H, Sampling Count η, Threshold θ, Failure Threshold β

Output : $\mathcal{M}^{\uparrow}$

1 $s \leftarrow s_0$; $h \leftarrow 0$; $f \leftarrow 0$
2 $\pi \leftarrow \text{modelBasedSolver}(S, A, s_0, g, \mathcal{M}^{\uparrow}, R, \gamma, H)$
3 **while** $|\Delta| < \Delta_{\mathcal{S}}$ **do**
4 **if** $f > \beta$ *or* $\mathit{unreachableGoal}(s_0, g, \mathcal{M}^{\uparrow}, \pi)$ **then**
5 $\text{explore}(\mathcal{M}^{\uparrow}, \Delta)$
6 **if** $\mathit{needsLearning}(\mathcal{M})$ **then**
7 $\mathcal{M}^{\uparrow} \leftarrow \text{learnModel}(\Delta, \mathcal{M}^{\uparrow})$
8 $\pi \leftarrow \text{modelBasedSolver}(S, A, s_0, g, \mathcal{M}^{\uparrow}, R, \gamma, H)$
9 $a \leftarrow \pi(s)$
10 $s' \leftarrow \Delta(s, a)$
11 $h \leftarrow h + 1$
12 **if** $(s, a, s') \rightleftharpoons \mathcal{M}^{\uparrow}$ **then**
13 $\mathcal{M} \leftarrow \text{goodnessOfFitTest}(s, a, s', \Delta, \mathcal{M}^{\uparrow}, \theta, \mathit{Freq})$
14 **else**
15 $\mathcal{M}^{\uparrow} \leftarrow \text{addInconsistentPredicates}(s, a, s', \mathcal{M}^{\uparrow})$
16 **if** $s \models g$ *or* $h \geq H$ **then**
17 $s \leftarrow s_0$; $f \leftarrow f + 1$ iff $s \not\models g$
18 **else**
19 $s \leftarrow s'$
20 **return** $\mathcal{M}^{\uparrow}$

for M using the learned model $\mathcal{M}^{\uparrow}$ (line 2) using an off-the-shelf RMDP solver such as LAO* (Hansen and Zilberstein, 2001).

If the transition graph of π derived using $\mathcal{M}^{\uparrow}$ has no path to the goal or if the goal has not been reached for a certain threshold (lines 4-5) the agent performs an exploration of the state space using the simulator in order to find a transition that is not $\mathcal{M}$-consistent. Initially, when the learned model is empty, this step allows the agent to quickly discover transitions for which useful learning can be performed. We used random walks of length H to conduct this exploration step in our experiments. If an inconsistent transition is discovered as part of the exploration process, then several

models to consider are added to the model learner using the approach in Sec. 9.3.2. This causes model learning to be invoked to resolve the inconsistency and updates the learned model $\mathcal{M}^{\uparrow}$ (line 7). We note that, as mentioned in Sec. 9.3.2, if new inconsistencies are identified during the model learning then they are resolved as well. Since the model has been updated, a new policy is computed (line 8).

Once any learning steps are complete and π has been computed, we execute an action $a = \pi(s)$ on the simulator (lines 9-10). If $(s, a, \Delta(s, a)) \rightleftharpoons \mathcal{M}$, then a goodness of fit test is performed to improve probability estimates as noted in Sec. 9.3.3 (line 13). An inconsistent transition always adds new models for the inconsistencies that need to be resolved by the model learner (line 15). If the goal is reached or the horizon is exceeded, the simulator is reset to the initial state and the total failures are incremented accordingly (lines 16-17). Finally, once the budget is exhausted (line 3) the learned model is returned (line 20) that can be used for solving future tasks.

9.3.5 Theoretical Results

Definition 40 (Variational Distance (VD))**.** Given an RMDP M, let $\mathcal{Z} = \{(s, a, s') | s, s' \in S, a \in A\}$ be a set of transitions. Also let $\mathcal{M}$ and $\mathcal{M}'$ be two models. The Variational Distance (VD) between these two models is then defined as $\mathrm{VD}_{\mathcal{Z}}(\mathcal{M}, \mathcal{M}') = \frac{1}{|\mathcal{Z}|} \sum_{\zeta \in \mathcal{Z}} |\mathbb{1}_{\zeta \rightleftharpoons \mathcal{M}} - \mathbb{1}_{\zeta \rightleftharpoons \mathcal{M}'}|$.

Definition 41 (Locally Convergent Model Learning)**.** Given an RMDP M, let $\mathcal{M}$ be the current model and $\mathcal{M}_\delta$ be the accurate (unknown) model s.t. $\mathcal{M}_\delta \equiv \delta$. Consider ε to be an error bound on the variational distance between two models. Model learning is locally convergent iff $\forall \varepsilon$ such that $0 < \varepsilon < \mathrm{VD}_{\tau_n}(\mathcal{M}, \mathcal{M}_\delta)$, $\exists n \in \mathbb{N}$ and a set τ_n of n

distinct transitions sampled from δ, s.t. the model $\mathcal{M}'$ learned using any $\mathcal{T}$ containing $\tau_n (\tau_n \subseteq \mathcal{T})$ will satisfy: $\text{VD}_{\mathcal{T}}(\mathcal{M}', \mathcal{M}_\delta) \leq \varepsilon < \text{VD}_{\tau_n}(\mathcal{M}, \mathcal{M}_\delta)$.

Theorem 13. Let M be an RMDP with a series of transition system changes $\delta_1, \ldots, \delta_n$ at timesteps $t_1, \ldots, t_n$ implemented using a simulator Δ, then during each stationary epoch between t_i and t_{i+1} Alg. 6 performs locally convergent model learning.

Proof (Sketch). Let $\mathcal{M}$ be the learned model at timestep i. By the correctness property of AQML (Thm. 2 in Verma *et al.* (2023a)) the set of transitions that $\mathcal{M}$ can generate must be a subset of the ones that $\mathcal{M}_{\delta_i}$ can. Let $Z = \{s : (s, a, s') | s, s' \in S, a \in A)\}$ and let $z = |Z|$. Let $\text{VD}(\mathcal{M}, \mathcal{M}_\delta)$ be x/z. ε has to be such that $0 < \varepsilon < x/z$. Let $\mathcal{M}'$ be the model learned using a set of transitions τ_n that are consistent with $\mathcal{M}_\delta$ but cannot be generated by $\mathcal{M}$. Choose τ_n such that τ_n has exactly $n (> z\epsilon)$ elements. Now, using the model $\mathcal{M}'$ that AQML learns, it will be able to generate τ_n in addition to all the transitions that $\mathcal{M}$ could generate. This implies: $\text{VD}\ (\mathcal{M}, \mathcal{M}_\delta) - VD(\mathcal{M}', \mathcal{M}_\delta) = x/z - (x - n)/z > x/z - (x - z\epsilon)/z = x/z - x/z + (z\epsilon)/z = \varepsilon$, and we have the desired result with τ_n as the set that is required for local convergence. By properties of AQML (Thm. 1 in Verma *et al.* (2023a)) any superset of transitions valid under $\mathcal{M}_\delta$ that contains τ_n will also reduce VD by at least ε. □

9.4 Experiments

We implemented our approach (Alg. 6) in Python and performed an empirical evaluation on four benchmark domains using a single core on a Xeon E5-2680 v4 CPU running at 2.4 GHz with a memory limit of 8 GiB. We found that our approach leads to significantly better transfer performance as compared to the baselines. We

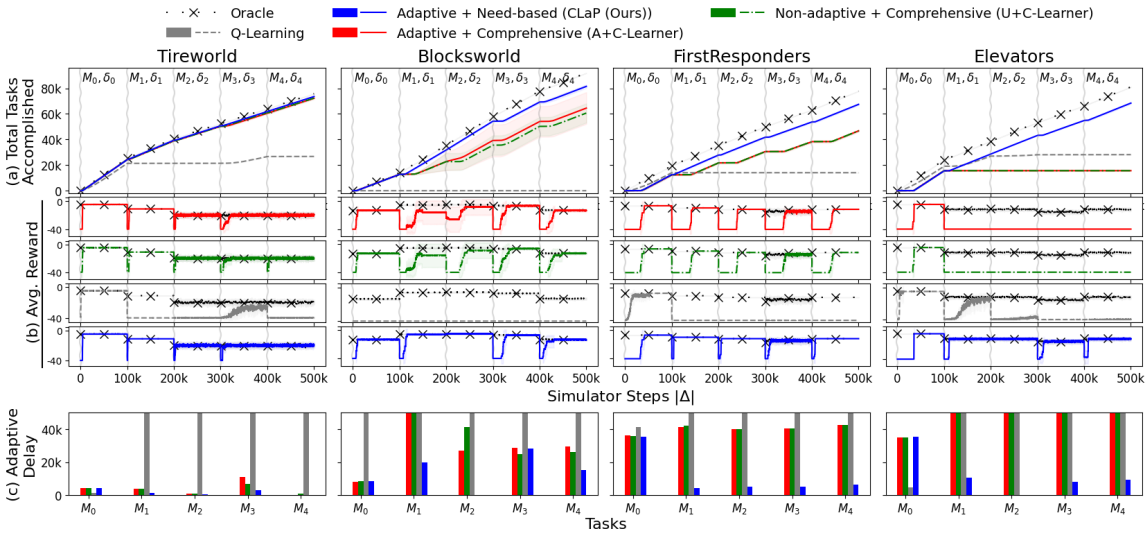


Figure 24. Results (best viewed in color) from our experiments averaged across 10 runs with 1-std deviation (shaded). (a) plots the learning curves of the methods, (b) plots the avg. reward obtained by greedily running the policy computed 10 times (for clarity, the Oracle's avg. reward is annotated with $\times$ periodically), (c) plots the total steps needed to achieve steady-state performance equal to the Oracle's (truncated at 40k for clarity). Higher values are better for (a) and (b); lower for (c). Vertical squiggly lines denote the step where a new task M_{i+1} and transition system δ_{i+1} were loaded ($M_i \neq M_{i+1}$ and $\delta_i \neq \delta_{i+1}$).

describe the empirical setup that we used for conducting the experiments followed by a discussion of the obtained results (Sec. 9.4.1).

Domains We used four benchmark domains that have been used in various International Probabilistic Planning Competitions (IPPCs) [10] for our experiments. We used these benchmark domains since ground truth models for them are available and we synthesized simulators using these domains.

We briefly describe the domains that we used below. We refer to each domain as $\mathcal{D}^{\uparrow}(|\mathcal{P}^{\uparrow}|, |\mathcal{A}^{\uparrow}|)$ to indicate the total number of predicates and actions in the domain. **Tireworld**$(4, 2)$ is a popular domain that has been used in several IPPCs. The

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

objective of this IPPC benchmark is to drive from the initial position to the goal position (accounting for flat tires that can stochastically occur).

FirstResponders$(13, 10)$ is a domain inspired from emergency services. The objective is to put out all fires and treat all victims. To do so, a planning agent needs to be able to plan to reach locations under fire and put them out (refilling water as needed) and also treat victims either at the fire site or ferry them to a hospital if the injuries are too severe.

Elevators$(9, 10)$ is a stochastic extension of the deterministic Miconic (Long and Fox, 2003) domain wherein there are several new objectives such as coins to be collected and elements such as shafts and gates that constrain navigation.

Blocksworld$(5, 4)$ is an environment where the goal is to arrange blocks in specific configurations. The IPPC variant is ExplodingBlocks wherein the table can be destroyed whilst stacking blocks. We tried to generate problems for ExplodingBlocks but were unsuccessful and as a result used the ergodic version instead where stacking blocks has a chance to drop them on the table. Nevertheless, the non-stationarity we introduce (described below) can often introduce dead-end states (i.e., states from which the goal cannot be reached).

Task Generation All tasks in the benchmark suite share a single transition system and, to the best of our knowledge, there are no official problem generators that can introduce non-stationarity and generate tasks for it. Thus, we introduced non-stationarity by generating new domain files obtained by changing a randomly selected action from the domain file of the previous task that was generated. We performed between 0-3 changes in both the preconditions and effects of the selected action by adding or deleting a predicate or by modifying an existing predicate in the action's model and ensured that at least one change was made. This method of introducing

non-stationarity resulted in the transition system of the final task being significantly different from the benchmark task with several actions changed.

Task Setup We generated five different tasks $M_0, \ldots, M_4$ with different initial states and goals. M_0 was the benchmark task and the others were generated using Breadth First Search. We used $\gamma = 0.9$ and horizon $H = 40$ for all tasks.

Baselines We used Q-Learning as our non-transfer RL baseline. We also used an Oracle that has complete access to the closed-form model of the simulator and uses LAO* to compute policies. This baseline provides an upper bound on the performance achievable by any algorithm. We also use two AQML-based methods: A+C-Learner and U+C-Learner. Both approaches learn comprehensive models. The former (latter) is adaptive (non-adaptive) to transition system changes, i.e., A+C-Learner tries to compute a policy and if an inconsistency is detected, learns from scratch whereas U+C-Learner is *informed* that the transition system has changed in order to relearn. We used QACE (Verma *et al.*, 2023a) as the AQML-based model-learning algorithm in these baselines. These methods are compared against our learner (CLaP) which is an active, adaptive, need-based learning system implementing Alg. 6.

A+C/U+C-Learner are SOTA methods for learning stochastic PPDDL models (deterministic model learners are inapplicable in our setting). We also considered ILM (Ng and Petrick, 2019) since it can learn stochastic noisy deictic rules but were unable to get it to work despite employing significant effort (and contacting the authors).

Hyperparameters We used $\alpha = 0.3$ for Q-Learning, $\eta = 100$ for the AQML-based methods and CLaP. Additionally, we used $\beta = 10$ and $\theta = 0.05$ for CLaP.

9.4.1 Analysis of Results

As mentioned in Sec. 9.2, we consider a task accomplished when a goal state is reached. We used a simulator budget $\Delta_{\mathcal{S}} = 100k$ for each task. The transition system is kept stationary for $\Delta_{\mathcal{S}}$ steps. The simulator is then loaded with a new task M_{i+1} and a new transition system δ_{i+1}.

Fig. 24 shows the obtained results from our experiments with 10 different random seeds used by the algorithms. We analyze the results to answer the following questions.

a. Is CLaP sample efficient?

b. Are CLaP solutions performant?

c. Are CLaP solutions generalizable?

Evaluation Metrics We use the following evaluation metrics to answer the questions above; We answer (a) by plotting learning curves that showcase how many tasks were accomplished during the learning process; We answer (b) by comparing the policy quality wherein at every $k = 100$ simulator steps, we freeze the computed policy and generate 10 policy traces each starting from the initial state s_0 of the task with a maximum horizon of 40. These simulations do not count towards the simulators budget. We report the average reward obtained while doing so; We answer (c) by computing the adaptive delay (Balloch *et al.*, 2022) which measures how many steps are necessary in the environment before the steady-state performance converges to that of the Oracle's.

It is clear from Fig. 24 that our approach of continual learning and planning (CLaP) outperforms both non-transfer (Q-Learning) and model-based methods; A+C/U+C-Learner.

(a) Sample Efficiency Our results in Fig. 24(a) show that CLaP has a much better sample complexity compared to the baselines. The learning curves from FirstResponders, Elevators and Blocksworld show that our approach can accomplish significantly more tasks than the baselines. Q-Learning does not learn and transfer any knowledge and thus needs to collect large amounts of experience to solve tasks.

A+C-Learner and U+C-Learner cannot efficiently correct the model when transition systems change since they need to learn all actions to converge. This drawback of comprehensive learners is highlighted in the results on the Elevators domain where even Q-Learning was able to outperform these methods. For the Elevators domain, the transition system change rendered some actions executable from states that were reachable only over very long horizons. The transition system of most of these actions had not changed and were not very useful to solve the task. The comprehensive learners exhausted the simulator's budget trying to relearn these task-irrelevant actions and thus were not able to solve the task. CLaP on the other hand only lazily-evaluates whether to learn a fraction of the model or not and was able to quickly fix the learned model and compute a policy that could solve the task. These trends can also be seen in FirstResponders where comprehensive learners must relearn 10 actions from scratch every time an inconsistency is observed.

(b) Better Task Performance Fig. 24(b) shows that avg. rewards of CLaP policies are very close to the Oracle's. This suggests that our learned models are often good approximations of the transition system. CLaP's policies converge to those of the Oracle's across all tasks in our evaluation.

(c) Better Generalizablity Our approach has a significantly lower adaptive delay (Fig. 24(c)), i.e., CLaP is able to utilize and transfer the learned knowledge across problems efficiently compared to the baselines that take a significant number of samples

to converge to the Oracle's performance. For example, CLaP zero-shot transferred (adaptive delay was 0) between Blocksworld tasks M_1 and M_2 requiring no learning to solve task M_2 while also matching the Oracle's performance. In cases where adaptation was needed (e.g., between Blocksworld tasks M_0, M_1, and M_2, M_3) CLaP few-shot learns the required knowledge to accomplish the task with policy qualities similar to that of the Oracle. In general, CLaP's adaptive delay was the best amongst all baselines.

We also conducted a directed experiment to evaluate the adaptability of our method to changing distributions. To do this, we generate two tasks from a 2-armed bandit domain. Pulling any of the levers stochastically takes the agent to the goal. Thus, the optimal policy is to repeatedly pull the lever with the highest probability of reaching the goal. In task one, the first (second) lever would succeed with probability 0.8 (0.5). In the second, it was 0.1 (0.9) respectively with preconditions and effects unchanged. CLaP utilizes goodness of fit tests and thus was able to adapt to this distribution shift and chose lever 1 (2) for task one (two). A+C-Learner cannot adapt to such changes and continued to use lever 1 for task two. This resulted in its policies being 9x worse than CLaP's with overall only ≈950 goals achieved compared to CLaP's ≈1550 ($\Delta_{\mathcal{S}} = 1000$ per task, $\eta = 10$). Fig. 25 shows the results obtained from this experiment. CLaP uses goodness of fit tests and thus is able to quickly identify that the distribution of the first lever has changed. Once the correct probabilities are learned for the first lever it computes a new policy that identifies that lever 2 is more lucrative.

Limitations and Future Work Currently, CLaP does not consider the task goal in the model learning process (line 7 of Alg. 6). Making optimistic estimates about

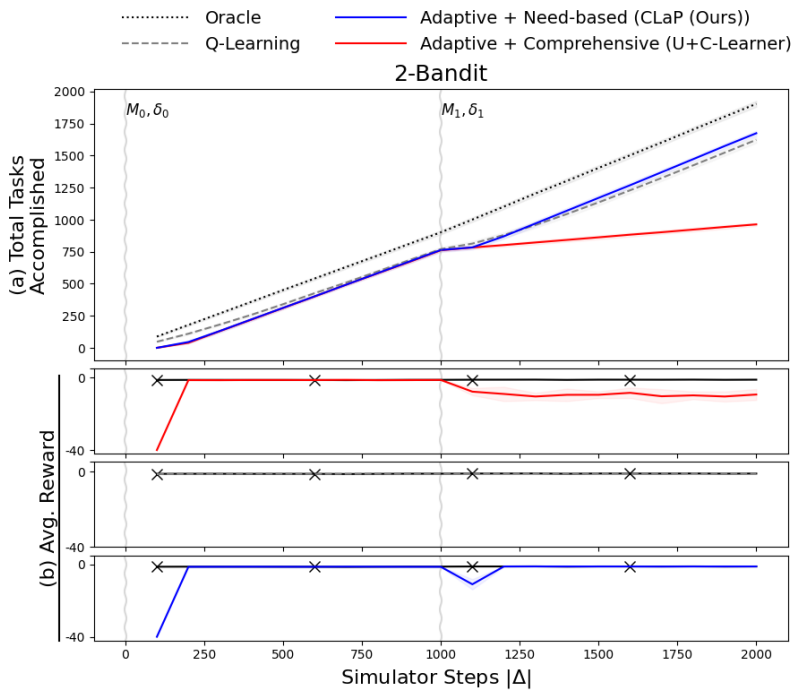


Figure 25. Results from our directed 2-armed bandit test averaged across 10 runs with 1-std deviation (shaded). (a) plots the learning curves of the methods, (b) plots the avg. reward obtained by greedily running the policy computed 10 times (for clarity, the Oracle's avg. reward is annotated with $\times$ periodically). Vertical squiggly lines denote the step where a new task M_{i+1} and transition system δ_{i+1} were loaded ($M_i \neq M_{i+1}$ and $\delta_i \neq \delta_{i+1}$).

the model w.r.t. the goal might allow the model learner to expend fewer samples for learning a model that can accomplish the task.

We do not take into account transition system changes or goals that could be provided in advance. CLaP could utilize that information to develop a curriculum so that useful, unlikely-to-change actions are prioritized to be learned early even if they do not contribute towards the current task's goal.

When is it better to learn-from-scratch There were not many performance gains compared to A+C/U+C-Learner in the Tireworld domain. This is because Tireworld is a small domain with only 2 (4) actions (predicates) that need to be learned. It

is intuitively clear that if the transition system significantly changes then relearning from scratch could save some computational effort. Devising heuristics that can evaluate whether learning from scratch would be easier than correcting the model is an interesting problem that we plan to investigate in future work.

9.5 Related Work

There has been plenty of work for transfer in RL (Mnih *et al.*, 2015; Schulman *et al.*, 2017) and on non-stationarity (commonly referred to as *novelty* in the RL literature). We focus on approaches that transfer across RMDP tasks. Tadepalli *et al.* (2004) provides an extensive overview for relational RL approaches.

Model-Based Reinforcement Learning The Dyna framework (Sutton, 1990) forms the basis of several model-based reinforcement learning (MBRL) approaches wherein experience from the environment is used to simultaneously learn a model and use the model to generate synthetic experience that is used for learning updates. Ng and Petrick (2019) use conjunctive first-order features to learn models and generalizable policies that transfer to related classes of RMDPs. Their approach does not perform guided exploration to resolve ambiguities. REX (Lang *et al.*, 2012) enables MBRL to automatically learn tasks autonomously. One challenge with this approach is learning accurate models since exploration can be sparse when using REX. V-MIN (Martínez *et al.*, 2017) integrates model-learning and planning with RL by requesting demonstrations from a teacher if it cannot find a policy whose expected value is greater than a certain threshold. The requirement of an available teacher limits the transfer capabilities of this approach. Taskable RL (TRL) (Illanes *et al.*, 2020) and RePReL (Kokel *et al.*, 2023) show how Hierarchical Reinforcement Learning (HRL)

using the options framework can be used for TRL. They use symbolic plans to guide the RL process. This approach requires models provided as input and are not learned. In contrast, our generates its own data for learning models using an active learning process.

Learning Models for Non-Stationary Settings GRL (Karia and Srivastava, 2022) train a neural network to learn reactive policies that can transfer to problems from the same domain but with different state spaces. Their approach is limited to only changes in the state space and cannot adapt to changes in the transition dynamics. Nayyar *et al.* (2022) and Musliner *et al.* (2021) learn models for non-stationary environments that can be integrated into the interleaved learning and planning loop. However, their approach only learns deterministic models and requires the use of optimal agents and observation traces to identify changes in transition dynamics. Bryce *et al.* (2016) address the problem of learning the updated mental model of a user using particle filtering given prior knowledge about the user's mental model. However, they assume that the entity being modeled can tell the learning system about flaws in the learned model if needed. Eiter *et al.* (2010) propose a framework for updating action laws depicted in the form of graphs representing the state space. They assume that changes can only happen in effects, and that knowledge about the state space and what effects might change is available beforehand. There is a recent body of work on adapting symbolic models to novelties in open-world environments for reinforcement learning (Goel *et al.*, 2022; Balloch *et al.*, 2023; Sreedharan and Katz, 2023; Mohan *et al.*, 2024). These methods are limited to deterministic settings and/or can only learn new models from passively collected data.

9.6 Concluding Remarks

We developed a sample-efficient method for transferring epistemial knowledge between an interleaved learning and planning process. Our approach can easily handle non-stationary environments on-the-fly by automatically detecting any changes that are inconsistent with the learned model. We reduce sample complexity by only considering the parts of the model that are inconsistent with the simulator's execution and selectively update the model. We are resilient to changes in the transition system even if it occurs during the model learning process. We show that when the transition system is stationary our approach is locally convergent. Furthermore, our learned lifted models easily transfer to new tasks. Our empirical results show that our approach significantly reduces sample complexity whilst remaining performant w.r.t. the optimal policy.

Chapter 10

CONCLUSIONS AND FUTURE WORK

This dissertation looked at the area of third party assessment of black-box AI systems under various settings. It defined the requirements in an AI system that would enable their assessment, and also defined the query-response interface that can make such an assessment possible. The thesis systematically fulfilled the four desiderata, namely interpretability, correctness, generalizability, and minimal requirements, for any third-party assessment of an AI system. This dissertation starts a new area of research that is necessary if lay people are going to routinely interact with adaptive AI systems that are like black boxes. In the future, more theoretical analysis can be done on this topic.

Extended Query Complexity Analysis Theoretically, the asymptotic complexity of AIA (with plan outcome queries) is $O(|P^*| \times |A|)$, but it does not take into account how much computation is needed to answer the queries or to evaluate their responses. This complexity just shows the amount of computation needed in the worst case to derive the agent model by AIA. I plan to perform a more detailed analysis of the complexity of AIA's queries in terms of data, expression, and combined complexities using the results of relational query complexity by Vardi (1982, 1995).

Chapter 8 presented an evaluation of membership classes of plan outcome queries for deterministic settings. I plan to extend this to cover more types of queries covered in Chapter 6 and to calculate tighter complexity bounds for the queries, instead of just their membership classes. This analysis would involve using relational database queries equivalent to the queries that I use in my work, and calculating their complexity.

Limitations and Future Directions The algorithms and methods presented in this dissertation work well for AI systems with an access to simulator in fully observable settings. There is still a lot of work needed to extend this work for settings with partial observability. Such settings also closely mimic major deployments of AI systems in the wild.

Second limitation of this work is the assumption of simulator access to the environment where the AI system is deployed. To remove this requirements, a physics-model for the deployed environment must be built for the simulator. Generating such a model can be done by the AI assessment system in the future. The assessment system can interact with the environment to generate this model and provide it as input to the simulator.

In addition to addressing these limitations in the future, the thesis work can be extended to:

- partially observable settings to increase its scope of applicability.
- classical reinforcement learning (RL) settings to make RL more sample efficient.
- analyze large language models as AI agents to evaluate their capabilities. This can go beyond assessing LLMs capabilities to just generate and interpret formal specification (Karia *et al.*, 2024a).
- make AI systems compliant with Level II assistive AI (Srivastava, 2021) by integrating this work with interfaces like JEDAI (Shah *et al.*, 2022), PDSim (De Pellegrin and Petrick, 2024), JEDAI-Ed (Dobhal *et al.*, 2024), etc.
- empower end-to-end world model learning by extending works like Shah *et al.* (2024) to active learning settings.

Aarts, F., J. De Ruiter and E. Poll, "Formal models of bank cards for free", in "2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops", (2013).

Aarts, F., P. Fiterau-Brostean, H. Kuppens and F. Vaandrager, "Learning register automata with fresh value generation", in "International Colloquium on Theoretical Aspects of Computing", (2015).

Aarts, F., F. Heidarian, H. Kuppens, P. Olsen and F. Vaandrager, "Automata learning through counterexample guided abstraction refinement", in "Proceedings of the 18th International Symposium on Formal Methods", (2012).

Aarts, F. and F. Vaandrager, "Learning i/o automata", in "Proc. International Conference on Concurrency Theory", (2010).

Agrawal, P., A. V. Nair, P. Abbeel, J. Malik and S. Levine, "Learning to poke by poking: Experiential learning of intuitive physics", in "Proc. NIPS", (2016).

Aineto, D., S. J. Celorrio and E. Onaindia, "Learning action models with minimal observability", Artificial Intelligence **275**, 104–137 (2019).

Aler, R., D. Borrajo and P. Isasi, "Genetic programming of control knowledge for planning", in "Proc. AIPS", (1998).

Amir, E. and A. Chang, "Learning partially observable deterministic action models", JAIR **33**, 349–402 (2008).

Amir, E. and S. Russell, "Logical filtering", in "Proc. IJCAI", (2003).

Amitai, Y., Y. Septon and O. Amir, "Explaining reinforcement learning agents through counterfactual action outcomes", in "Proc. AAAI", (2024).

Angluin, D., "Learning regular sets from queries and counterexamples", Information and Computation **75**, 2, 87–106 (1987).

Angluin, D., "Queries and concept learning", Machine Learning **2**, 4, 319–342 (1988).

Angluin, D. and D. Fisman, "Learning regular omega languages", Theoretical Computer Science **650**, 57–72 (2016).

Anjomshoae, S., A. Najjar, D. Calvaresi and K. Främling, "Explainable agents and robots: Results from a systematic literature review", in "Proc. AAMAS", (2019).

Arndt, A. D., J. B. Ford, B. J. Babin and V. Luong, "Collecting samples from online services: How to use screeners to improve data quality", International Journal of Research in Marketing (2021).

Arora, A., H. Fiorino, D. Pellier, M. Métivier and S. Pesty, "A review of learning planning action models", The Knowledge Engineering Review **33**, E20 (2018).

Bäckström, C. and P. Jonsson, "Bridging the gap between refinement and heuristics in abstraction", in "Proc. IJCAI", (2013).

Balac, N., D. Gaines and D. Fisher, "Learning action models for navigation in noisy environments.", in "ICML 2000 MLSK Workshop", (2000).

Balloch, J. C., Z. Lin, M. Hussain, A. Srinivas, R. Wright, X. Peng, J. M. Kim and M. O. Riedl, "Novgrid: A flexible grid world for evaluating agent response to novelty", in "AAAI 2022 Spring Symposium on Designing AI for Open Worlds", (2022).

Balloch, J. C., Z. Lin, X. Peng, M. Hussain, A. Srinivas, R. Wright, J. M. Kim and M. O. Riedl, "Neuro-symbolic world models for adapting to open world novelty", in "Proc. AAMAS", (2023).

Barredo Arrieta, A., N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila and F. Herrera, "Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI", Information Fusion **58**, 82–115 (2020).

Battaglia, P., R. Pascanu, M. Lai, D. Jimenez Rezende and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics", in "Proc. NIPS", (2016).

Benson, S., "Inductive learning of reactive action models", in "Proc. ICML", (1995).

Blondel, G., M. Arias and R. Gavaldà, "Identifiability and transportability in dynamic causal networks", International Journal of Data Science and Analytics **3**, 2, 131–147 (2017).

Bonet, B. and H. Geffner, "Learning first-order symbolic representations for planning from the structure of the state space", in "Proc. ECAI", (2020).

Bonet, B. and R. Givan, "5th International planning competition: Non-deterministic track – call for participation", (2005).

Bongard, J. and H. Lipson, "Active coevolutionary learning of deterministic finite automata", Journal of Machine Learning Research **6**, 2, 1651–1678 (2005).

Botinčan, M. and D. Babić, "Sigma*: Symbolic learning of input-output specifications", in "Proc. 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages", (2013).

Bryce, D., J. Benton and M. W. Boldt, "Maintaining evolving domain models", in "Proc. IJCAI", (2016).

Bryce, D. and O. Buffet, "6th International Planning Competition: Uncertainty Part", in "Proceedings of the 6th International Planning Competition", (2008).

Camacho, A. and S. A. McIlraith, "Learning interpretable models expressed in linear temporal logic", in "Proc. ICAPS", (2019).

Cassel, S., F. Howar, B. Jonsson, M. Merten and B. Steffen, "A succinct canonical register automaton model", Journal of Logical and Algebraic Methods in Programming **84**, 1, 54–66 (2015).

Cassel, S., F. Howar, B. Jonsson and B. Steffen, "Active learning for extended finite state machines", Formal Aspects of Computing **28**, 2, 233–263 (2016).

Chakraborti, T., S. Sreedharan, Y. Zhang and S. Kambhampati, "Plan explanations as model reconciliation: Moving beyond explanation as soliloquy", in "Proc. IJCAI", (2017).

Chitnis, R., T. Silver, J. Tenenbaum, L. P. Kaelbling and T. Lozano-Perez, "GLIB: Efficient exploration for relational model-based reinforcement learning via goal-literal babbling", in "Proc. AAAI", (2021).

Cimatti, A., M. Roveri and P. Traverso, "Strong planning in non-deterministic domains via model checking", in "Proc. AIPS", (1998).

Cranmer, M., A. Sanchez Gonzalez, P. Battaglia, R. Xu, K. Cranmer, D. Spergel and S. Ho, "Discovering symbolic models from deep learning with inductive biases", in "Proc. NeurIPS", (2020).

Cresswell, S. and P. Gregory, "Generalised domain model acquisition from action traces", in "Proc. ICAPS", (2011).

Cresswell, S., T. McCluskey and M. West, "Acquisition of object-centred domain models from planning examples", in "Proc. ICAPS", (2009).

Daniele, M., P. Traverso and M. Y. Vardi, "Strong cyclic planning revisited", in "Proc. ECP", (1999).

Das, D., S. Chernova and B. Kim, "State2Explanation: Concept-based explanations to benefit agent learning and user understanding", in "Proc. NeurIPS", (2023).

De Pellegrin, E. and R. P. A. Petrick, "Planning domain simulation: An interactive system for plan visualisation", in "Proc. ICAPS", (2024).

Denis, F., A. Lemay and A. Terlutte, *Learning Regular Languages using RFSAs* (Springer Berlin Heidelberg, 2001).

Dhurandhar, A., P.-Y. Chen, R. Luss, C.-C. Tu, P. Ting, K. Shanmugam and P. Das, "Explanations based on the missing: Towards contrastive explanations with pertinent negatives", in "Proc. NeurIPS", (2018).

Diankov, R. and J. Kuffner, "Openrave: A planning architecture for autonomous robotics", Tech. Rep. CMU-RI-TR-08-34, Carnegie Mellon University, USA (2008).

Dobhal, D., J. Nagpal, R. Karia, P. Verma, R. K. Nayyar, N. Shah and S. Srivastava, "Using explainable AI and hierarchical planning for outreach with robots", arXiv preprint arXiv:2404.00808 (2024).

Doshi-Velez, F. and B. Kim, *Considerations for Evaluation and Generalization in Interpretable Machine Learning*, pp. 3–17 (Springer International Publishing, 2018).

Duffy, N., J. Crowley and G. Lacey, "Object detection using colour", in "Proc. ICPR", (2000).

Dupont, P., "Incremental regular inference", in "Proc. Third International Colloquium on Grammar Inference", (1996).

Eiter, T., E. Erdem, M. Fink and J. Senko, "Updating action domain descriptions", in "Proc. IJCAI", (2005).

Eiter, T., E. Erdem, M. Fink and J. Senko, "Updating action domain descriptions", Artificial Intelligence **174**, 15, 1172–1221 (2010).

Farzan, A., Y. F. Chen, E. M. Clarke, Y. K. Tsay and B. Y. Wang, "Extending automated compositional verification to the full class of omega-regular languages", in "Proc. TACAS", (2008).

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

Fisher, R. A., "On the mathematical foundations of theoretical statistics", Philosophical Transactions of the Royal Society of London, Series A **222**, 594-604, 309–368 (1922).

Fiterau-Brostean, P., R. Janssen and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations", in "Proc. International Conference on Computer Aided Verification", (2016).

Fox, M. and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains", JAIR **20**, 1, 61–124 (2003).

Fragkiadaki, K., P. Agrawal, S. Levine and J. Malik, "Learning visual predictive models of physics for playing billiards", in "Proc. ICLR", (2016).

Giantamidis, G. and S. Tripakis, "Learning moore machines from input-output traces", in "International Symposium on Formal Methods", (2016).

Gil, Y., "Learning by experimentation: Incremental refinement of incomplete planning domains", in "Proc. ICML", (1994).

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

Goel, S., Y. Shukla, V. Sarathy, M. Scheutz and J. Sinapov, "RAPid-Learn: A framework for learning to recover for handling novelties in open-world environments", in "Proc. ICDL", (2022).

Gregory, P. and S. Cresswell, "Domain model acquisition in the presence of static relations in the LOP system", in "Proc. ICAPS", (2015).

Gregory, P. and A. Lindsay, "Domain model acquisition in domains with action costs", in "Proc. ICAPS", (2016).

Greydanus, S., A. Koul, J. Dodge and A. Fern, "Visualizing and understanding Atari agents", in "Proc. ICML", (2018).

Halpern, J. Y., "A modification of the Halpern-Pearl definition of causality", in "Proc. IJCAI", (2015).

Halpern, J. Y., *Actual Causality* (The MIT Press, 2016).

Hansen, E. A. and S. Zilberstein, "LAO*: A heuristic search algorithm that finds solutions with loops", AIJ **129**, 1-2, 35–62 (2001).

Hart, P. E., N. J. Nilsson and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths", IEEE Transactions on Systems Science and Cybernetics **4**, 2, 100–107 (1968).

Helmert, M., "A planning heuristic based on causal graph analysis", in "Proc. ICAPS", (2004).

Helmert, M., "The fast downward planning system", JAIR **26**, 191–246 (2006).

Helmert, M., "Concise finite-domain representations for pddl planning tasks", Artificial Intelligence **173**, 5-6, 503–535 (2009).

Helmert, M. and C. Domshlak, "Landmarks, critical paths and abstractions: What's the difference anyway?", in "Proc. ICAPS", (2009).

Helmert, M., P. Haslum, J. Hoffmann *et al.*, "Flexible abstraction heuristics for optimal sequential planning", in "Proc. ICAPS", (2007).

Hoffmann, J. and B. Nebel, "The FF planning system: Fast plan generation through heuristic search", JAIR **14**, 253–302 (2001).

Howar, F., B. Steffen, B. Jonsson and S. Cassel, "Inferring canonical register automata", in "Proc. International Workshop on Verification, Model Checking, and Abstract Interpretation", (2012).

Illanes, L., X. Yan, R. T. Icarte and S. A. McIlraith, "Symbolic plans as high-level instructions for reinforcement learning", in "Proc. ICAPS", (2020).

Immerman, N., "Expressibility as a complexity measure: Results and directions", Tech. Rep. YALEU/DCS/TR-538, Department of Computer Science, Yale University (1987).

Isberner, M., F. Howar and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning", in "Proc. International Conference on Runtime Verification", (2014).

James, S., B. Rosman and G. Konidaris, "Learning portable representations for high-level planning", in "Proc. ICML", (2020).

Jiménez, S., T. De La Rosa, S. Fernández, F. Fernández and D. Borrajo, "A review of machine learning for automated planning", The Knowledge Engineering Review **27**, 4, 433–467 (2012).

Jin, M., Z. Ma, K. Jin, H. H. Zhuo, C. Chen and C. Yu, "Creativity of AI: Automatic symbolic option discovery for facilitating deep reinforcement learning", in "Proc. AAAI", (2022).

Juba, B., H. S. Le and R. Stern, "Safe learning of lifted action models", in "Proc. KR", (2021).

Juba, B. and R. Stern, "Learning probably approximately complete and safe action models for stochastic worlds", in "Proc. AAAI", (2022).

Kanazawa, K. and T. Dean, "A model for projection and action", in "Proc. IJCAI", (1989).

Kansky, K., T. Silver, D. A. Mély, M. Eldawy, M. Lázaro-Gredilla, X. Lou, N. Dorfman, S. Sidor, S. Phoenix and D. George, "Schema networks: Zero-shot transfer with a generative causal model of intuitive physics", in "Proc. ICML", (2017).

Karia, R., D. Dobhal, D. Bramblett, P. Verma and S. Srivastava, "Can LLMs translate SATisfactorily? Assessing LLMs in generating and interpreting formal specifications", in "AAAI 2024 Spring Symposium on User-Aligned Assessment of Adaptive AI Systems", (2024a).

Karia, R. and S. Srivastava, "Relational abstractions for generalized reinforcement learning on symbolic problems", in "Proc. IJCAI", (2022).

Karia, R., P. Verma, G. Vipat and S. Srivastava, "Epistemic exploration for generalizable planning and learning in non-stationary stochastic settings", in "NeurIPS 2023 GenPlan Workshop", (2023).

Karia, R., P. Verma, G. Vipat and S. Srivastava, "Epistemic exploration for generalizable planning and learning in non-stationary settings", in "Proc. ICAPS", (2024b).

Kearns, M. J. and U. V. Vazirani, *An Introduction to Computational Learning Theory* (MIT Press, 1994).

Kennedy, R., S. Clifford, T. Burleigh, P. D. Waggoner, R. Jewell and N. J. G. Winter, "The shape of and solutions to the mturk quality crisis", Political Science Research and Methods **8**, 4, 614–629 (2020).

Khan, F. S., R. M. Anwer, J. van de Weijer, A. D. Bagdanov, M. Vanrell and A. M. Lopez, "Color attributes for object detection", in "Proc. CVPR", (2012).

Kiefer, J. and J. Wolfowitz, "Consistency of the maximum likelihood estimator in the presence of infinitely many incidental parameters", The Annals of Mathematical Statistics pp. 887–906 (1956).

Kim, B., J. A. Shah and F. Doshi-Velez, "Mind the gap: A generative approach to interpretable feature selection and extraction", in "Proc. NeurIPS", (2015).

Kim, B., M. Wattenberg, J. Gilmer, C. Cai, J. Wexler, F. Viegas and R. Sayres, "Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (TCAV)", in "Proc. ICML", (2018).

Kokel, H., A. Manoharan, S. Natarajan, B. Ravindran and P. Tadepalli, "Reprel: Integrating relational planning and reinforcement learning for effective abstraction", in "Proc. ICAPS", (2021).

Kokel, H., S. Natarajan, B. Ravindran and P. Tadepalli, "RePReL: A unified framework for integrating relational planning and reinforcement learning for effective abstraction in discrete and continuous domains", Neural Comput. Appl. **35**, 23, 16877–16892 (2023).

Konidaris, G., L. P. Kaelbling and T. Lozano-Perez, "From skills to symbols: Learning symbolic representations for abstract high-level planning", JAIR **61**, 1, 215–289 (2018).

Krishnan, A., A. Williams and C. Martens, "Towards action model learning for player modeling", in "Proc. AIIDE", (2020).

Kučera, J. and R. Barták, "LOUGA: Learning planning operators using genetic algorithms", in "Knowledge Management and Acquisition for Intelligent Systems", (2018).

Lage, I. and F. Doshi-Velez, "Learning interpretable concept-based models with human feedback", in "ICML Workshop on Human Interpretability in Machine Learning", (2020).

Lamanna, L., A. E. Gerevini, A. Saetti, L. Serafini and P. Traverso, "On-line learning of planning domains from sensor data in pal: Scaling up to large state spaces", in "Proc. AAAI", (2021a).

Lamanna, L., A. Saetti, L. Serafini, A. Gerevini and P. Traverso, "Online learning of action models for pddl planning", in "Proc. IJCAI", (2021b).

Lang, T., M. Toussaint and K. Kersting, "Exploration in relational domains for model-based reinforcement learning", JMLR **13**, 3725–3768 (2012).

Levine, J. and D. Humphreys, "Learning action strategies for planning domains using genetic programming", in "Applications of Evolutionary Computing", (2003).

Li, Y., Y.-F. Chen, L. Zhang and D. Liu, "A novel learning algorithm for büchi automata based on family of dfas and classification trees", Information and Computation **281**, 104678 (2021).

Lindsay, A., "Reuniting the locm family: An alternative method for identifying static relationships", in "ICAPS 2021 KEPS Workshop", (2021).

Liu, G., O. Schulte, W. Zhu and Q. Li, "Toward interpretable deep reinforcement learning with linear model U-trees", in "Proc. ECML PKDD", (2018).

Long, D. and M. Fox, "The 3rd international planning competition: Results and analysis", JAIR **20**, 1–59 (2003).

Lyu, D., F. Yang, B. Liu and S. Gustafson, "Sdrl: Interpretable and data-efficient deep reinforcement learning leveraging symbolic planning", in "Proc. AAAI", (2019).

Ma, X., S. Mishra, A. Beirami, A. Beutel and J. Chen, "Let's do a thought experiment: Using counterfactuals to improve moral reasoning", in "ICML 2023 Neural Conversational AI Workshop", (2023).

Macke, W., R. Mirsky and P. Stone, "Expected value of communication for planning in ad hoc teamwork", in "Proc. AAAI", (2021).

Madumal, P., T. Miller, L. Sonenberg and F. Vetere, "Explainable reinforcement learning through a causal lens", in "Proc. AAAI", (2020).

Maler, O. and I.-E. Mens, "Learning regular languages over large alphabets", in "Proc. TACAS", (2014).

Maler, O. and A. Pnueli, "On the learnability of infinitary regular sets", Information and Computation **118**, 2, 316–326 (1995).

Malle, B. F., *How the Mind Explains Behavior: Folk Explanations, Meaning, and Social Interaction* (The MIT Press, 2004).

Mao, J., T. Lozano-Pérez, J. B. Tenenbaum and L. P. Kaelbing, "PDSketch: Integrated domain programming, learning, and planning", in "Proc. NeurIPS", (2022).

Martínez, D., G. Alenyà, C. Torras, T. Ribeiro and K. Inoue, "Learning relational dynamics of stochastic domains for planning", in "Proc. ICAPS", (2016).

Martínez, D. M., G. Alenyà, T. Ribeiro, K. Inoue and C. Torras, "Relational reinforcement learning for planning with exogenous effects", JMLR **18**, 78:1–78:44 (2017).

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. S. Weld and D. Wilkins, "PDDL – The planning domain definition language", Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Comp. Vision and Control (1998).

Mehta, N., P. Tadepalli and A. Fern, "Autonomous learning of action models for planning", in "Proc. NIPS", (2011).

Miller, T., "Explanation in artificial intelligence: Insights from the social sciences", Artificial Intelligence **267**, 1–38 (2019).

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning", nature **518**, 7540, 529–533 (2015).

Moerman, J., "Learning product automata", in "Proc. 14th International Conference on Grammatical Inference", (2018).

Mohan, S., W. Piotrowski, R. Stern, S. Grover, S. Kim, J. Le, Y. Sher and J. de Kleer, "A domain-independent agent architecture for adaptive operation in evolving open worlds", Artificial Intelligence p. 104161 (2024).

Mou, Y. and K. Xu, "The media inequality: Comparing the initial human-human and human-AI social interactions", Computers in Human Behavior **72**, 432–440 (2017).

Mourão, K., L. Zettlemoyer, R. P. A. Petrick and M. Steedman, "Learning STRIPS operators from noisy and incomplete observations", in "Proc. UAI", (2012).

Muise, C., S. McIlraith and C. Beck, "Improved non-deterministic planning by exploiting state relevance", in "Proceedings of the 22nd International Conference on Automated Planning and Scheduling", (2012).

Musliner, D. J., M. J. Pelican, M. McLure, S. Johnston, R. G. Freedman and C. Knutson, "OpenMIND: Planning and adapting in domains with novelty", in "Proc. CACS", (2021).

Nashed, S. B., S. Mahmud, C. V. Goldman and S. Zilberstein, "Causal explanations for sequential decision making under uncertainty", in "Proc. AAMAS", (2023).

Nayyar, R. K., P. Verma and S. Srivastava, "Differential assessment of black-box AI agents", in "Proc. AAAI", (2022).

Ng, J. H. A. and R. P. A. Petrick, "Incremental learning of planning actions in model-based reinforcement learning", in "Proc. IJCAI", (2019).

Oncina, J. and P. García, "Inferring regular languages in polynomial update time", Pattern Recognition and Image Analysis **1**, 49–61 (1992).

Pacharoen, W., T. Aoki, P. Bhattarakosol and A. Surarerks, "Active learning of nondeterministic finite state machines", Mathematical Problems in Engineering p. 373265 (2013).

Parekh, R. and V. Honavar, "An incremental interactive algorithm for regular grammar inference", in "Proc. Third International Colloquium on Grammar Inference", (1996).

Pasula, H. M., L. S. Zettlemoyer and L. P. Kaelbling, "Learning symbolic models of stochastic domains", JAIR **29**, 309–352 (2007).

Paulus, R., C. Xiong and R. Socher, "A deep reinforced model for abstractive summarization", in "Proc. ICML", (2018).

Pearl, J., "Causal diagrams for empirical research", Biometrika **82**, 4, 669–688 (1995).

Pearl, J., *Causality: Models, Reasoning, and Inference* (Cambridge University Press, 2009).

Pearson, K., *On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to have Arisen from Random Sampling*, pp. 11–28 (Springer New York, New York, NY, 1992).

Peled, D., M. Y. Vardi and M. Yannakakis, "Black box checking", Journal of Automata Languages and Combinatorics **7**, 2, 225–246 (2001).

Perez-Liebana, D., J. Liu, A. Khalifa, R. D. Gaina, J. Togelius and S. M. Lucas, "General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms", IEEE Transactions on Games **11**, 3, 195–214 (2019).

Perez-Liebana, D., S. Samothrakis, J. Togelius, T. Schaul and S. Lucas, "General video game AI: Competition, challenges and opportunities", in "Proc. AAAI", (2016).

Pinsker, M. S., *Information and Information Stability of Random Variables and Processes* (Holden-Day, Inc., 1964).

Popov, I., N. Heess, T. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez and M. Riedmiller, "Data-efficient deep reinforcement learning for dexterous manipulation", arXiv preprint arXiv:1704.03073 (2017).

Qualtrics, "Qualtrics XM", https://www.qualtrics.com/, URL https://www.qualtrics.com/, accessed: 2022-05-10 (2005).

Raffelt, H., B. Steffen, T. Berg and T. Margaria, "Learnlib: A framework for extrapolating behavioral models", International Journal on Software Tools for Technology Transfer **11**, 5, 393–407 (2009).

Randazzo, R., "What went wrong with Uber's Volvo in fatal crash? Experts shocked by technology failure", The AZ Republic (2018).

Richardson, M. and P. Domingos, "Markov logic networks", Machine learning **62**, 1, 107–136 (2006).

Rintanen, J., "Madagascar: Scalable planning with SAT", in "Proc. 8th International Planning Competition.", (2014).

Rivest, R. L. and R. E. Schapire, "Inference of finite automata using homing sequences", Information and Computation **103**, 103, 51–73 (1993).

Rodrigues, C., P. Gérard and C. Rouveirol, "Incremental learning of relational action models in noisy environments", in "Proc. ILP", (2011a).

Rodrigues, C., P. Gérard, C. Rouveirol and H. Soldano, "Active learning of relational action models", in "Proc. ILP", (2011b).

Rodriguez, I. D., B. Bonet, J. Romero and H. Geffner, "Learning first-order representations for planning from black box states: New results", in "Proc. KR", (2021).

Roy, R., D. Fisman and D. Neider, "Learning interpretable models in the property specification language", in "Proc. IJCAI", (2020).

Russell, S. J., "Rationality and intelligence", Artificial Intelligence **94**, 1-2, 57–77 (1997).

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

Schaul, T., "A video game description language for model-based or interactive learning", in "2013 IEEE Conference on Computational Intelligence in Games (CIG)", (2013).

Schulman, J., F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal policy optimization algorithms", arXiv preprint arXiv:1707.06347 (2017).

Schulze, K. G., R. N. Shelby, D. J. Treacy, M. C. Wintersgill, K. VanLehn and A. Gertner, "Andes: An active learning, intelligent tutoring system for Newtonian Physics", Themes in Education **1**, 2, 115–136 (2000).

Settles, B., *Active Learning* (Morgan & Claypool Publishers, 2012).

Shah, N., D. Kala Vasudevan, K. Kumar, P. Kamojjhala and S. Srivastava, "Anytime integrated task and motion policies for stochastic environments", in "Proc. ICRA", (2020).

Shah, N., J. Nagpal, P. Verma and S. Srivastava, "From reals to logic and back: Inventing symbolic vocabularies, actions, and models for planning from raw data", arXiv preprint arXiv:2402.11871 (2024).

Shah, N., P. Verma, T. Angle and S. Srivastava, "JEDAI: A system for skill-aligned explainable robot planning", in "Proc. AAMAS", (2022).

Shahaf, D. and E. Amir, "Logical circuit filtering", in "Proc. IJCAI", (2007).

Shahbaz, M. and R. Groz, "Inferring mealy machines", in "Proc. 2nd World Congress on Formal Methods", (2009).

Shirazi, A. and E. Amir, "First-order logical filtering", Artificial Intelligence **175**, 1, 193–219 (2011).

Silver, T. and R. Chitnis, "PDDLGym: Gym environments from PDDL problems", in "ICAPS 2020 PRL Workshop", (2020).

Sollich, P. and D. Saad, "Learning from queries for maximum information gain in imperfectly learnable problems", in "Proc. NeurIPS", (1994).

Sreedharan, S., T. Chakraborti and S. Kambhampati, "Foundations of explanations as model reconciliation", Artificial Intelligence p. 103558 (2021).

Sreedharan, S., A. O. Hernandez, A. P. Mishra and S. Kambhampati, "Model-free model reconciliation", in "Proc. IJCAI", (2019).

Sreedharan, S. and M. Katz, "Optimistic exploration in reinforcement learning using symbolic model estimates", in "Proc. NeurIPS", (2023).

Sreedharan, S., U. Soni, M. Verma, S. Srivastava and S. Kambhampati, "Bridging the gap: Providing post-hoc symbolic explanations for sequential decision-making problems with inscrutable representations", in "Proc. ICLR", (2022).

Sreedharan, S., S. Srivastava and S. Kambhampati, "Hierarchical expertise level modeling for user specific contrastive explanations", in "Proc. IJCAI", (2018).

Srivastava, S., "Unifying principles and metrics for safe and assistive AI", in "Proc. AAAI", (2021).

Srivastava, S., E. Fang, L. Riano, R. Chitnis, S. Russell and P. Abbeel, "Anytime integrated task and motion policies for stochastic environments", in "Proc. ICRA", (2014).

Srivastava, S., S. Russell and A. Pinto, "Metaphysics of planning domain descriptions", in "Proc. AAAI", (2016).

Steffen, B., F. Howar and M. Merten, *Introduction to Active Automata Learning from a Practical Perspective*, pp. 256–296 (Springer Berlin Heidelberg, 2011).

Stern, R. and B. Juba, "Efficient, safe, and probably approximately complete learning of action models", in "Proc. IJCAI", (2017).

Student, "The probable error of a mean", Biometrika **6**, 1, 1–25 (1908).

Sutton, R. S., "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming", in "Proc. ICML", (1990).

Sutton, R. S. and A. G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, 1998).

Tadepalli, P., R. Givan and K. Driessens, "Relational reinforcement learning: An overview", in "ICML 2004 RRL Workshop", (2004).

Vaandrager, F., "Model learning", Communications of the ACM **60**, 2, 86–95 (2017).

Vardi, M. Y., "The complexity of relational query languages (extended abstract)", in "Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing", (1982).

Vardi, M. Y., "On the complexity of bounded-variable queries (extended abstract)", in "Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems", (1995).

Čertický, M., "Real-time action model learning with online algorithm 3sg", Applied Artificial Intelligence **28**, 7, 690–711 (2014).

Verma, A., V. Murali, R. Singh, P. Kohli and S. Chaudhuri, "Programmatically interpretable reinforcement learning", in "Proc. ICML", (2018).

Verma, P., R. Karia and S. Srivastava, "Autonomous capability assessment of sequential decision-making systems in stochastic settings", in "Proc. NeurIPS", (2023a).

Verma, P., R. Karia, G. Vipat, A. Gupta and S. Srivastava, "Learning AI-system capabilities under stochasticity", in "NeurIPS 2023 GenPlan Workshop", (2023b).

Verma, P., S. R. Marpally and S. Srivastava, "Asking the right questions: Learning interpretable action models through query answering", in "Proc. AAAI", (2021a).

Verma, P., S. R. Marpally and S. Srivastava, "Learning user-interpretable descriptions of black-box AI system capabilities", in "ICAPS 2021 KEPS Workshop", (2021b).

Verma, P., S. R. Marpally and S. Srivastava, "Discovering user-interpretable capabilities of black-box planning agents", in "Proc. KR", (2022).

Verma, P. and S. Srivastava, "Learning generalized models by interrogating black-box autonomous agents", in "AAAI 2020 GenPlan Workshop", (2020).

Verma, P. and S. Srivastava, "Learning causal models of autonomous agents using interventions", in "IJCAI 2021 GenPlan Workshop", (2021).

Verma, P. and S. Srivastava, "Learning causally accurate models for autonomous assessment of deterministic black-box agents", Tech. Rep. TR-ASUSCAI-2024-001, School of Computing and Augmented Intelligence, Arizona State University (2024a).

Verma, P. and S. Srivastava, "User-aligned autonomous capability assessment of black-box AI systems", in "AAAI 2024 Spring Symposium on User-Aligned Assessment of Adaptive AI Systems", (2024b).

Volpato, M. and J. Tretmans, "Approximate active learning of nondeterministic input output transition systems", in "Proc. 15th International Workshop on Automated Verification of Critical Systems", (2015).

Walsh, T. J. and M. L. Littman, "Efficient learning of action schemas and web-service descriptions", in "Proc. AAAI", (2008).

Wang, X., "Learning planning operators by observation and practice", in "International Conference on Planning Systems", (1994).

Wang, Z., C. Wang, X. Xiao, Y. Zhu and P. Stone, "Building minimal and reusable causal state abstractions for reinforcement learning", in "Proc. AAAI", (2024).

Wang, Z., X. Xiao, Z. Xu, Y. Zhu and P. Stone, "Causal dynamics learning for task-independent state abstraction", in "Proc. ICML", (2022).

Watkins, C., *Learning from Delayed Rewards*, Ph.D. thesis, King's College, Cambridge, UK (1989).

Weber, C., D. Morwood and D. Bryce, "Goal-directed knowledge acquisition", in "ICML 2011 Workshop on Planning and Acting with Uncertain Models", (2011).

Wise, M., M. Ferguson, D. King, E. Diehr and D. Dymesich, "Fetch and freight: Standard platforms for service robot applications", in "IJCAI Workshop on Autonomous Mobile Service Robots", (2016).

Wu, K., Q. Yang and Y. Jiang, "ARMS: An automatic knowledge engineering tool for learning action models for AI planning", Knowledge Engineering Review **22**, 2, 135–152 (2007).

Xu, J. and J. Laird, "Instance-based online learning of deterministic relational action models", in "Proc. AAAI", (2010).

Yang, F., D. Lyu, B. Liu and S. Gustafson, "Peorl: Integrating symbolic planning and hierarchical reinforcement learning for robust decision-making", in "Proc. IJCAI", (2018).

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

Younes, H. L. S. and M. L. Littman, "PPDDL 1.0: An extension to PDDL for expressing domains with probabilistic effects", Tech. Rep. CMU-CS-04-167, Carnegie Mellon University, USA (2004).

Younes, H. L. S., M. L. Littman, D. Weissman and J. Asmuth, "The first probabilistic track of the International Planning Competition", JAIR **24**, 851–887 (2005a).

Younes, H. L. S., M. L. Littman, D. Weissman and J. Asmuth, "The first probabilistic track of the international planning competition", JAIR **24**, 851–887 (2005b).

Zettlemoyer, L. S., H. M. Pasula and L. P. Kaelbling, "Logical particle filtering", in "Probabilistic, Logical and Relational Learning - A Further Synthesis", (2008).

Zhi-Xuan, T., J. Mann, T. Silver, J. Tenenbaum and V. Mansinghka, "Online bayesian goal inference for boundedly-rational planning agents", in "Proc. NeurIPS", (2020).

Zhuo, H. H. and S. Kambhampati, "Action-model acquisition from noisy plan traces", in "Proc. IJCAI", (2013).

Zhuo, H. H. and S. Kambhampati, "Model-lite planning: Case-based vs. model-based approaches", Artificial Intelligence **246**, 1–21 (2017).

Zhuo, H. H., T. Nguyen and S. Kambhampati, "Model-lite case-based planning", in "Proc. AAAI", (2013).

Zhuo, H. H., J. Peng and S. Kambhampati, "Learning action models from disordered and noisy plan traces", arXiv preprint arXiv:1908.09800 (2019).

Zhuo, H. H. and Q. Yang, "Action-model acquisition for planning via transfer learning", Artificial Intelligence **212**, 80–103 (2014).

Zhuo, H. H., Q. Yang, D. H. Hu and L. Li, "Learning complex action models with quantifiers and logical implications", Artificial Intelligence **174**, 18, 1540–1569 (2010).

Zhuo, H. H., Q. Yang, R. Pan and L. Li, "Cross-domain action-model acquisition for planning via web search", in "Proc. ICAPS", (2011).

APPENDIX A

USER STUDY DETAILS

In this appendix, we describe the details of the user study survey that was given to the study participants. The participants were split into two groups. The capability group and the primitive action group.

A.1 Game Description

The participants in both the groups were shown the description of the game. As shown in Fig. 26, this description lists out the rules of the game.

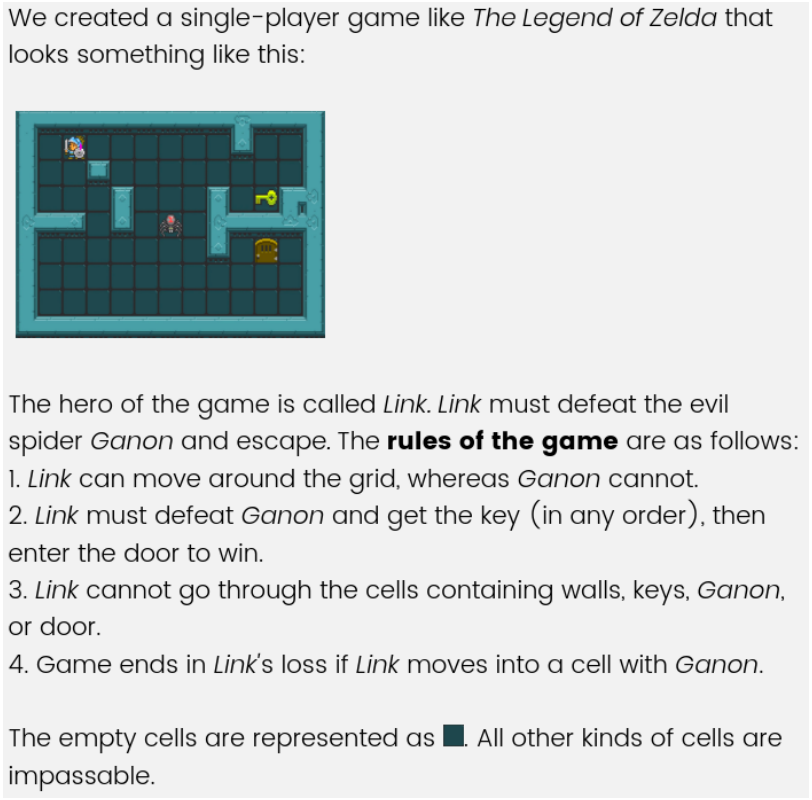


Figure 26. Game description shown to the study participants

A.2 Capability Descriptions

The participants are then shown the next part based on which group they fall in. The participants in the capability group are shown description of 6 parameterized actions, each generated using boilerplate templates for each predicate. We show

here (Fig. 27) the description of the capability $c4$ whose learned description was shown in Fig.1(d) in the main paper. The participants are also given an option to choose between eight possible descriptions of from which they choose the correct summarization of that capability. This is illustrated in Fig. 27.

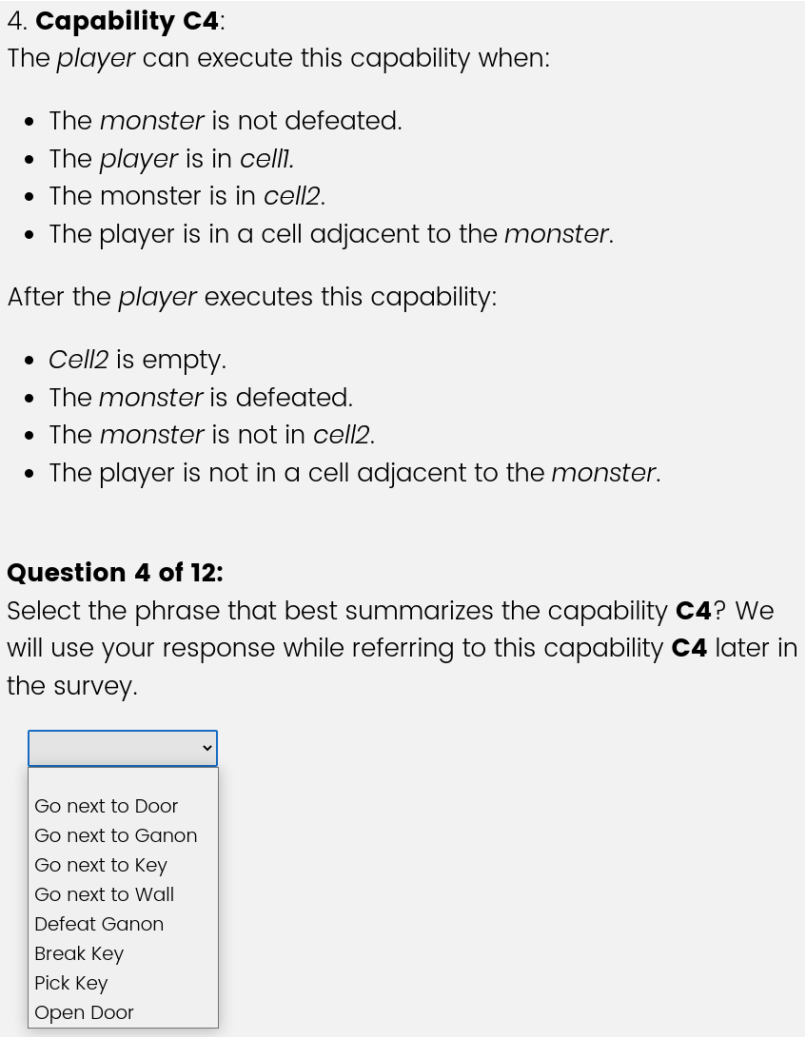


Figure 27. Description of the capability C4 with summarization options.

A.3 Action Descriptions

Similar to the capability group, the participants in the primitive action group are shown textual descriptions of the keystrokes, with five options to choose from. Each

option provides a possible description of the keyboard in English. Fig. 28 shows the description of keystroke W with the five options.

Notice that we tried to keep the same format for the description of actions as that of capabilities, i.e., of the form "if $\langle x \rangle$ conditions hold then $\langle y \rangle$ happens." Also, the description of capabilities are parameterized by the player, monster, cells, etc. whereas the description in primitive actions use the object names like Link, Gannon, etc. directly.

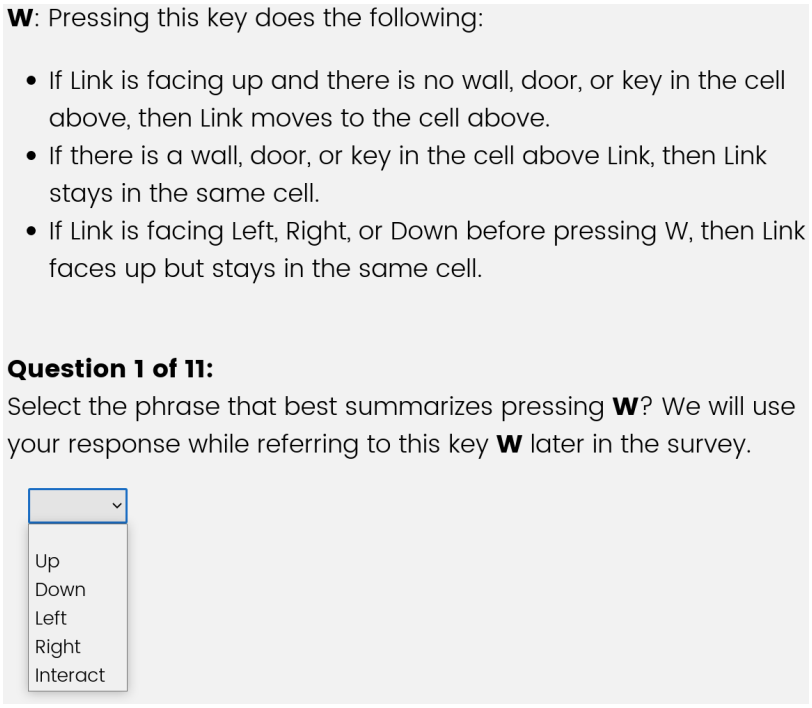


Figure 28. Description of the keystroke W with summary options

A.4 Questions

After showing the capability and action descriptions, the participants of both the groups are shown the same questions. These questions give two-game images and ask the participant the sequence of capabilities or actions (depending on the user's group) that the agent should execute to reach the goal state from the initial state. One such question is shown in Fig. 29. There were six such questions in total shown.in total to all the participants.

Sanity Question: One of these six was a sanity check question. The answer was given

in the question itself. The responses for any participant who got this question wrong were discarded.

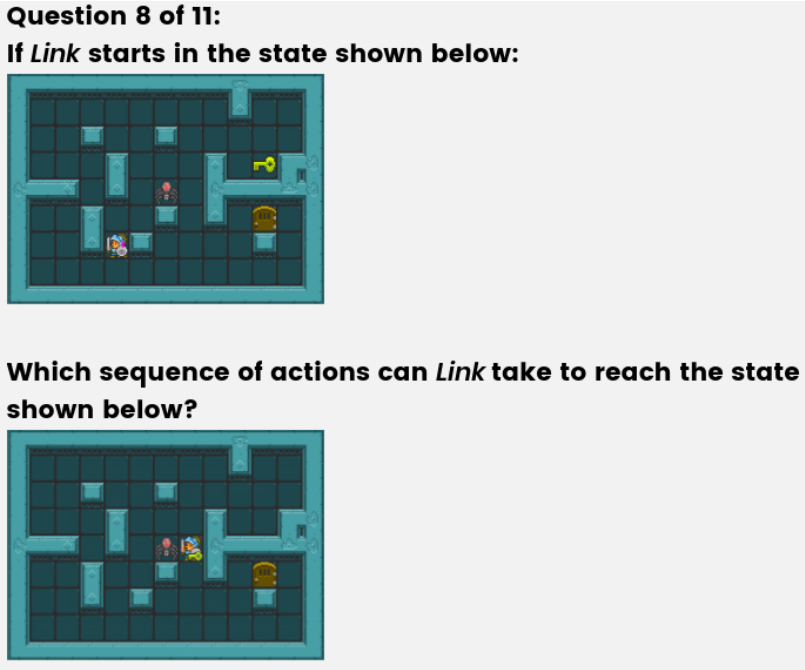


Figure 29. A sample user study question

A.5 Options

The options given to the two sets of users for the same question differed because the capability group participants were given options in terms of capability sequence that the agent can execute (shown in Fig. 30), whereas the primitive action group participants were given options in terms of sequences of primitive actions (shown in Fig. 31). Note that these options refer to the question shown in Fig. 29.

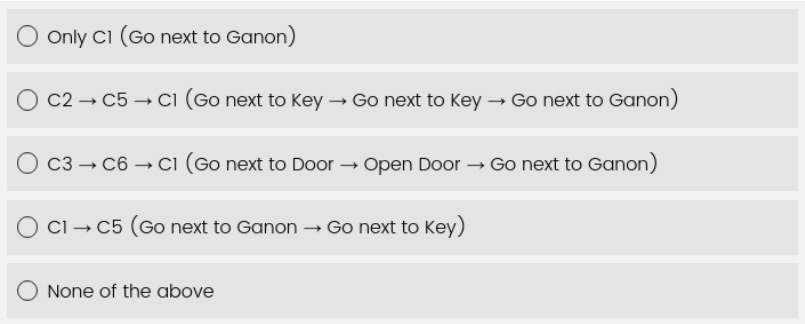


Figure 30. Options for question in Fig. 29 given to capability group participants

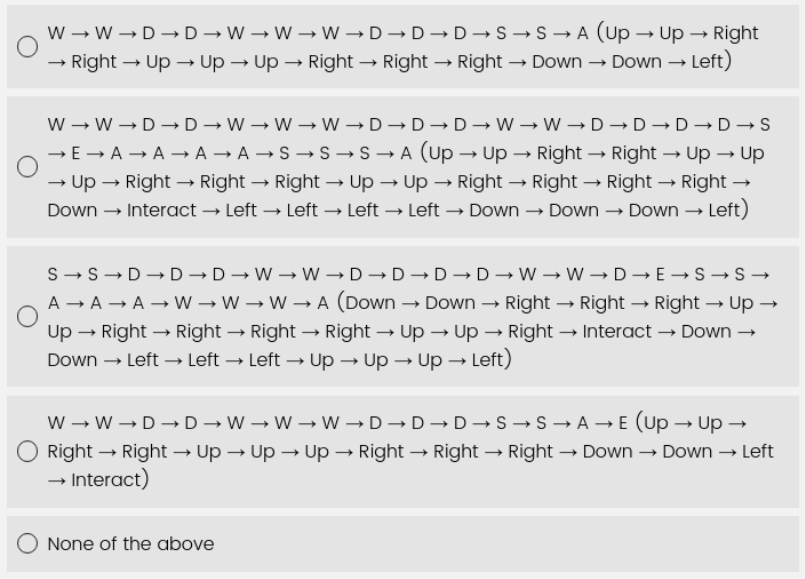


Figure 31. Options for question in Fig. 29 given to primitive action group participants

APPENDIX B

IRB APPROVAL

APPROVAL: MODIFICATION

Siddharth Srivastava
SCAI: Computing and Augmented Intelligence, School of
-
siddharths@asu.edu

Dear Siddharth Srivastava:

On 8/26/2021 the ASU IRB reviewed the following protocol:

Type of Review:	Modification / Update
Title:	Identifying the Computational Basis of Understandability
Investigator:	Siddharth Srivastava
IRB ID:	STUDY00012779
Funding:	Name: DOD-NAVY: Office of Naval Research (ONR), Grant Office ID: FP00026015, Funding Source ID: grant13190885
Grant Title:	None
Grant ID:	None
Documents Reviewed:	None

The IRB approved the modification.

When consent is appropriate, you must use final, watermarked versions available under the "Documents" tab in ERA-IRB.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (HRP-103).

All in-person interactions with human subjects require the completion of the ASU Daily Health Check by the ASU members prior to the interaction and the use of face coverings by researchers, research teams and research participants during the interaction. These requirements will minimize risk, protect health and support a safe research environment. These requirements apply both on- and off-campus.

The above change is effective immediately until further notice and replaces all previously published guidance. Thank you for your continued commitment to ensuring a healthy and productive ASU community.

Sincerely,

IRB Administrator

cc: Kyle Elliott
Nancy Cooke
Siddharth Srivastava