

**Collaborate and Explain On-the-Fly:
Knowledge-based Reasoning and Learning in
Ad Hoc Teamwork**

Hasra Dodampe Gamage



Doctor of Philosophy
Institute of Perception, Action and Behaviour
School of Informatics
The University of Edinburgh
2026

Abstract

Ad hoc teamwork requires an agent to collaborate with previously unknown teammates (human, AI) without prior coordination. State of the art methods address this primarily as a learning problem, using large datasets of prior observations to model teammate behavior and determine the actions of the ad hoc agent. Such data and the computational resources required are often unavailable in complex domains, and the resulting models are opaque and difficult to revise as the domain, team composition, or agents' capabilities change.

Motivated by these challenges, this thesis introduces KAT, an architecture that integrates knowledge-based reasoning and data-driven learning, enabling an ad hoc agent to reason, adapt, and explain its decisions using substantially fewer resources than purely data-driven methods. Given a goal, the ad hoc agent determines its actions through non-monotonic logical reasoning with prior domain-specific commonsense knowledge, models learned and revised rapidly to predict the behavior of other agents, and future tasks anticipated using generic knowledge of similar situations in a pre-trained Large Language Model. The agent further uses a combination of a pretrained Large Language Model and decision-tree induction to incrementally acquire and revise knowledge, in the form of objects, actions, and axioms governing domain dynamics, from natural language descriptions and observations of other agents' behavior; and generates relational descriptions as on-demand explanations of its own decisions and beliefs, and those of other agents, in response to different types of questions.

We ground and experimentally evaluate KAT in a variety of environments with different levels of complexity, demonstrating reliable, efficient, transparent, and scalable performance: a substantial improvement over purely knowledge-based baselines, and comparable or better performance than purely data-driven baselines, while using orders of magnitude fewer resources.

Lay Summary

Imagine a household assistive AI agent joining a family for the first time. It has never worked with them before, does not know their daily routines, and has no prior agreements about how tasks are shared. Yet it still needs to help, for example by loading the dishwasher alongside a family member, or by taking heated food out of the microwave while coordinating with another to prepare breakfast. To be useful, the agent must quickly understand how others work and adapt to their behavior on the fly. This challenge is known as *ad hoc teamwork*, where an agent must collaborate with others (humans, AI agents) without prior coordination.

Most existing approaches try to solve this problem by learning how different teammates act in a variety of situations by studying large amounts of past examples. These methods can struggle when data is limited, when the environment is complex, or when teammates change their behavior unexpectedly. They also tend to operate like “black boxes”, making it hard to understand why the agent makes certain decisions or how it interprets the behavior of others.

This thesis explores a new approach. Instead of relying primarily on large amounts of data, our system combines logical reasoning—using prior commonsense knowledge about the world, with the ability to quickly learn teammates behavior from limited examples. This allows the agent to cooperate effectively, even with limited information or when things change unexpectedly.

Additionally, the agent uses an existing large language model to anticipate high-level future goals in the environment and interpret natural-language descriptions from humans to acquire new knowledge. It also learns new knowledge from observations. This enables it to continually expand its understanding of the domain while interacting with others.

Finally, the agent explains its decisions and beliefs and those of its teammates in human understandable terms as answers to different types of questions.

We show that this approach works across a range of environments, even when the agent has limited information or when the team composition changes. Further, we show that the provided explanations are accurate and the agent’s reasoning with newly acquired knowledge improves both its own performance and that of the team.



For Shaggy – thank you for all the love.

Acknowledgements

Throughout my doctoral studies, I have been fortunate to receive support, guidance, and encouragement from many people and organizations, without which this work would not have been possible.

I am deeply grateful to my supervisor, Prof. Mohan Sridharan, for his guidance, insightful feedback, and mentorship throughout this journey. His encouragement has been invaluable in shaping both this research and my development as a scholar. Thanks to his advice, I was able to present my research at a high-profile conference early in my academic career, giving me both a valuable head start and confidence in my path as a researcher.

I would also like to thank my committee members at the University of Birmingham, Prof. Ales Leonardis and Prof. Chris Baber, for their thoughtful guidance and constructive feedback, which helped shape the development and presentation of this work. My gratitude extends to my committee members at the University of Edinburgh, Prof. Alex Lascarides and Prof. Ram Ramamoorthy, for their careful advice and insights, which strengthened this thesis further.

I am thankful to my colleagues and friends in the IRLab at the University of Birmingham and IPAB at the University of Edinburgh. Their support and stimulating discussions made the long hours of research and coding more enjoyable and engaging. Special thanks to Laura, Michalina, Yanrong, and Oliver for their friendship and for always taking the time to listen to my presentations and offer constructive feedback.

I am grateful to the U.S. Office of Naval Research Award for financial support, which allowed me to focus fully on my research.

My foremost thanks go to my family for their unwavering encouragement and love throughout this journey. To my husband, your patience, support, and belief in me made this journey possible in ways words cannot express.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Hasra Dodampe Gamage)

Table of Contents

List of Figures	xvii
List of Tables	xix
List of Acronyms	xxiii
1 Introduction	1
1.1 Ad Hoc Teamwork	1
1.2 Aims, Contributions and Scope	3
1.3 Thesis Structure	9
2 Background	13
2.1 Logical Reasoning	14
2.1.1 Non-Monotonic Logical Reasoning	16
2.1.2 Modal Logic	19
2.2 Declarative Languages	21
2.2.1 Answer Set Programming	22
2.3 Action Languages	24
2.4 Automated Planning	27
2.4.1 Classical Planning	27
2.4.2 Planning under Incomplete Information	29
2.4.3 Plan Recognition	31
2.4.4 Negotiation	32
2.5 Ecological Rationality and Heuristics	33
2.5.1 Fast and Frugal Trees	35
2.5.2 Constructing Fast and Frugal Trees	37
2.5.3 IFAN Algorithm for Fast and Frugal Trees	38
2.6 Large Language Models	41

2.7	Chapter Summary	43
3	Related Work	45
3.1	Ad Hoc Teamwork	45
3.1.1	Knowledge Representation	46
3.1.2	Action Selection	48
3.1.3	Adaptation to Current Teammates	50
3.2	Explanation AI	52
3.3	Knowledge Acquisition	54
3.4	Chapter Summary	56
4	Core Architecture	57
4.1	General Overview	57
4.2	Knowledge Representation	62
4.3	Reasoning	64
4.4	Agent Modeling	69
4.4.1	Ecological Rationality and Heuristics	69
4.4.2	Representational Choices and Models	69
4.5	Model Selection	72
4.6	KAT Pseudocode	75
4.7	Experimental Procedure	79
4.7.1	Environments	79
4.7.2	Agent Types	81
4.7.3	Experimental Procedure	83
4.8	Results	88
4.9	Chapter Summary	94
5	Task Anticipation and Partial Observability	97
5.1	Task Anticipation	98
5.1.1	Experimental Procedure	101
5.1.2	Quantitative Results	107
5.1.3	Qualitative Results	109
5.2	Scalability	112
5.2.1	Experimental Procedure	113
5.2.2	Results	113
5.3	Partial Observability and Communication	115

5.3.1	Experimental Procedure	116
5.3.2	Results	117
5.4	Chapter Summary	119
6	Knowledge Acquisition	123
6.1	General Overview	123
6.2	Learning From Natural Language Descriptions	125
6.3	Learning From Simulated Execution	130
6.4	Experimental Procedure	135
6.5	Quantitative Results	138
6.6	Qualitative Results	143
6.7	Chapter Summary	145
7	Explanation Generation	147
7.1	General Overview	147
7.2	Generating Relational Descriptions	148
7.2.1	Causal Questions	149
7.2.2	Contrastive Questions	153
7.2.3	Justify Beliefs	157
7.2.4	Counterfactual Questions	161
7.3	Experimental Procedure	163
7.4	Results	166
7.4.1	Quantitative Results	166
7.4.2	Qualitative Results	168
7.5	Chapter Summary	174
8	Conclusion	177
8.1	Key Results	178
8.2	Discussion	180
8.3	Directions for Future Work	182
A	Supporting Materials	185
A.1	Implementation Details	185
A.2	Answer Set Prolog Knowledge Base	186
A.3	Task Anticipation Prompts and Outputs	194
A.4	Domain Knowledge of the Incomplete Ad Hoc Agent	203

List of Figures

1.1	Screenshots from VirtualHome (Puig et al., 2018) showing a human (female in green top) and an assistive AI agent (male in blue shirt) collaborating to prepare breakfast and set up a workstation for remote work.	2
2.1	Fast and Frugal tree developed by Green and Mehr (1997) to help physicians in a Michigan hospital to decide whether a patient with severe chest pain should be admitted to an intensive coronary care unit (ICU) or to a regular ward.	36
4.1	KAT architecture combines the complementary strengths of knowledge-based and data-driven reasoning and learning to enable an ad hoc agent to collaborate with its teammates.	60
4.2	FF tree model of human behavior for the <i>grab book</i> action in the example scenario from the VirtualHome domain.	72
4.3	Screenshots from Fort Attack, Half Field Offense and VirtualHome domains.	81
5.1	Example prompt sent to the LLM to anticipate the human’s future tasks. Each prompt includes a list of possible tasks, two partially completed example routines, and a partially completed routine for the LLM to complete.	99
5.2	Execution example: using LLM without prompting strategies or external validator causes conflicts during execution, having a negative impact on performance.	110

5.3	Execution trace for task routine: [<i>Prepare breakfast, Prepare activities, Serve snacks, Clean kitchen</i>]. When an ad hoc agent is not allowed to predict and reason about the human’s actions, it may choose to execute same action(s) as the human, leading to longer plans. . . .	111
5.4	Execution example: using LLM to generate low-level actions results in poor performance.	112
5.5	Screenshot from the Fort Attack and Half Field Offense domains after introducing partial observability.	115
6.1	KAT leverages LLM, decision tree induction and prior knowledge to map natural language descriptions of actions and unexpected outcomes into new objects, actions and axioms in non-monotonic logical reasoning.	124
6.2	Part of the decision tree created to learn missing executability conditions.	134
6.3	Average number of objects, actions, and axioms learned by the ad hoc agent in three consecutive runs. The results are averaged over 10 trials, with each trial consisting of three runs (Exp2).	139
7.1	Belief tracing to explore why the ad hoc agent believed the cake was heated in step 6.	160
7.2	Part of the Fort Attack domain showing the ad hoc guard agent (green) moving to track and shoot an attacker (red).	170
7.3	Belief tracing to explore why the ad hoc agent did not grab the bread slice in step 1.	175

List of Tables

4.1	Attributes used to create the behavior models of the human agent in the example scenario from the VirtualHome domain. Number of attributes represent the <i>number of variables</i> in that attribute.	71
4.2	Attributes considered for models of other agents' behavior in Fort Attack domain. Number of attributes represent the <i>number of variables</i> in each attribute times the <i>number of agents</i>	84
4.3	Attributes used to learn teammates (offense) behavior models in the full version of the Half Field Offense domain. Number of attributes represent the <i>number of variables</i> in each attribute times the <i>number of agents</i>	86
4.4	Attributes used to learn defense agents' behavior models in the full version of the Half Field Offense domain. Number of attributes represent the <i>number of variables</i> in each attribute times the <i>number of agents</i>	86
4.5	Attributes used to create the behavior models of the human agent in the VirtualHome domain. Number of attributes represent the <i>number of variables</i> in that attribute.	88
4.6	Average percentage (%) of episodes in which guards won with handcrafted policies and learned agent models (Exp1).	89
4.7	Average percentage (%) of episodes in which guards win with previously unseen built-in policies (Exp2).	90
4.8	Prediction accuracy of learned behavior models.	90
4.9	Average shooting accuracy (%) of the ad hoc agent with previously unseen built-in policies.	91
4.10	Win percentage (%) for guards with handcrafted policies in Fort Attack domain. Model adaptation improves performance.	91

4.11	Win percentage (%) for guards with built-in policies in Fort Attack domain. Model adaptation improves performance.	92
4.12	Fraction of goals scored by the offense team in Half Field Offense domain in the limited version (2v2) and full version (4v5).	92
4.13	Prediction accuracy (%) of the learned agent behavior models in limited (2v2) version of the Half Field Offense domain (Exp4).	93
4.14	Prediction accuracy (%) of the learned agent behavior models in full (4v5) version of the Half Field Offense domain (Exp5).	93
4.15	Mean and standard deviation of number of steps and time taken by KAT architecture to achieve the goal, expressed as a fraction of these values for the baseline, alongside the average absolute values.	94
5.1	Average number of steps and time taken by the team (human, ad hoc agent) to complete task routines, with values for baselines computed as a fraction of these values for KAT in each trial; for comparison, the average absolute values are presented alongside. Task anticipation (LLM) and action prediction (FF trees) substantially improve performance.	108
5.2	Precision and recall values of the anticipated tasks with and without the LLM and the validator. A simple statistical model is not good enough and the validator plays an important role.	108
5.3	Average number of steps and time taken by the team (human, ad hoc agent) to complete tasks with prompting strategies and/or external validator; performance measure values for baselines computed as a fraction of values for KAT in each trial, with the average absolute values presented alongside.	109
5.4	Average number of steps and time taken by Team1, Team2 and Team3 to complete the task routines, with performance measure values for Teams 2-3 computed as a fraction of the values for Team 1 in each trial, showing average absolute values alongside; collaboration between agents improves performance.	114
5.5	Win percentage (%) for guards with hand-crafted policies in Fort Attack domain (Exp1). Communication addresses partial observability. .	119
5.6	Win percentage (%) for guards with built-in policies in Fort Attack domain (Exp2). Communication addresses partial observability. . . .	119

5.7	Goals scored (i.e., games won) by offense team in Half Field Offense domain under partial observability (Exp3, Exp4). KAT’s performance comparable with baseline that had no ad hoc agents in the team but used training datasets that were orders of magnitude larger.	120
6.1	Average precision and recall of learned axioms in 10 episodes; the agent accurately learns previously unknown axioms (Exp1).	139
6.2	Average number of action steps and time taken by the team to complete task sequences; represented as a fraction of the values of these measures for the baseline; reasoning with learned knowledge substantially improves performance. (Exp3)	140
6.3	Precision and recall of the learned axioms in Fort Attack domain (Exp4).	141
6.4	Precision and recall of the learned axioms in Half Field Offense domain (Exp6).	141
6.5	Win percentage of the team of guards in the Fort Attack domain with and without the learned axioms (Exp5).	141
6.6	Win percentage of the team of guards in the Half Field Offense domain with and without the learned axioms (Exp7).	142
7.1	Precision and recall of retrieving relevant literals for constructing explanations in the <i>Fort Attack</i> domain with the learned axioms (Exp1), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.	167
7.2	Precision and recall of retrieving relevant literals for constructing explanations in the <i>Fort Attack</i> domain without the learned axioms (Exp2), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.	167
7.3	Precision and recall of retrieving relevant literals for constructing explanations in the Half Field Offense domain with the learned axioms (Exp3), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.	168
7.4	Precision and recall of retrieving relevant literals for constructing explanations in the Half Field Offense domain without the learned axioms (Exp4), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.	168

7.5	Precision and recall of retrieving relevant literals for providing explanations in the VirtualHome domain (Exp5), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.	169
-----	--	-----

List of Acronyms

AHT	Ad Hoc Teamwork
AI	Artificial Intelligence
ASP	Answer Set Programming
CoT	Chain-of-thought
CR	Consistency Restoring
CWA	Closed World Assumptions
FF	Fast and Frugal
KAT	Knowledge-based reasoning and learning for Ad hoc Teamwork
LLM	Large Language Model
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning
UCT	Upper Confidence bounds for Tree

Chapter 1

Introduction

This chapter begins by introducing the Ad Hoc Teamwork (AHT) problem, an increasingly relevant challenge in multiagent systems. We discuss its practical applications and the limitations of existing approaches, outlining the key challenges that motivate this work (Section 1.1). We then define the objectives of this thesis, emphasizing its key contributions towards advancing AHT research, and outline its scope and limitations (Section 1.2). Finally, we present an overview of the chapters that follow, highlighting the contributions discussed in each chapter, and conclude with a list of publications that document the results of this work (Section 1.3).

1.1 Ad Hoc Teamwork

Consider an AI (Artificial Intelligence) agent performing household tasks in collaboration with other agents (AI, human) it has not worked with before. Figure 1.1 shows snapshots of this motivating scenario in which the AI agent (male, blue shirt) and a human (female, green top) are preparing breakfast and setting up a workstation for remote work. Both the human and the agent have a limited view of the environment at each step and do not communicate directly with each other, although each of them is aware of the state of the domain, including the location of teammates and the outcomes of their actions (e.g., change in the location of an object moved by a teammate). The human is aware of the upcoming tasks in the domain, while the agent is not. To collaborate effectively, the AI agent has to reason with different (e.g., relational, probabilistic) descriptions of prior knowledge (e.g., default statements, domain and agent attributes, and rules governing change) and uncertainty, adapting its action choices to changes in the domain and the behavior of the human. In addition, the human may



Figure 1.1: Screenshots from VirtualHome (Puig et al., 2018) showing a human (female in green top) and an assistive AI agent (male in blue shirt) collaborating to prepare breakfast and set up a workstation for remote work.

want to query and understand the decisions of the agent. These characteristics correspond to **AHT**, where an agent must cooperate with others “on the fly” without prior coordination (Stone et al., 2010).

The AHT problem arises in many practical applications in robotics, such as disaster rescue, space exploration, assistive robotics, and collaborative games, and poses challenges in knowledge representation, reasoning, and learning. These scenarios require agents to operate under strict time constraints, limited resources, and dynamic changes in the environment, team composition, and agent capabilities, making depending on joint training (as is common in many multiagent systems) often impractical.

There has been considerable research in AHT, as discussed in a recent survey (Mirsky et al., 2022). Initial methods relied on predefined protocols or plays designed for different scenarios, to select the actions of the ad hoc agent based on the current state observations (i.e., agent positions and trajectories) (Bowling and McCracken, 2005). Subsequent methods, especially those considered state of the art methods for AHT, include a key “data-driven” component in the form of probabilistic or deep network models that estimate the behavior of other agents (or agent *types*) and optimize the ad hoc agent’s actions based on policies learned from a long history of prior interactions with similar agents (Barrett et al., 2013; Rahman et al., 2021; Ribeiro et al., 2023). While effective in specific scenarios, these methods have key limitations.

First, they make it difficult to fully leverage the commonsense domain knowledge readily available in many practical domains. For example, an agent preparing breakfast may fail if it assumes that eggs are always in the fridge, whereas default knowledge

that eggs might instead be found in the pantry or inside the shopping bag would allow it to adapt successfully. Second, these methods often require large amounts of data and computational resources for learning the models and policies that govern the AI agent’s action choices, whereas practical AHT domains often impose strong resource constraints. It is difficult to gather such training datasets of different situations, and computationally expensive to build the necessary models. Third, it is challenging to deploy such models in AHT settings where previously unknown situations are common. Finally, the internal decision processes of such models can be difficult to interpret, which may limit their effectiveness in human-agent collaboration scenarios where understanding and trust are important (Anjomshoae et al., 2019; Fox et al., 2017).

These challenges motivate the need for approaches that can combine structured knowledge with data-driven adaptation. In the next section, we introduce the key research questions we aim to address in this thesis, along with the objectives, scope and contributions of the study.

1.2 Aims, Contributions and Scope

Building on the challenges highlighted in the previous section, this thesis aims to investigate how an ad hoc agent can collaborate effectively with previously unknown teammates in dynamic and resource-constrained environments. In particular, we consider the following research questions:

Research Question:

1. How to design an architecture for ad hoc teamwork that leverages the complementary strengths of **knowledge-based reasoning** and **data-driven learning** to enable effective and adaptive collaboration when performing under resource constraints?
2. How to enable ad hoc agents to describe their decisions as relational statements that make contact with human-level concepts such as beliefs and goals to promote transparency?

The limitations of existing approaches for AHT suggest several requirements that needs to be satisfied by an effective AHT architecture. First, it must support reasoning with incomplete and continuously evolving knowledge. In real-world settings,

an ad hoc agent rarely has access to a full specification of the environment or the goals and the behaviors of other agents. Instead, the agent must operate with partial information, make informed decisions based on what is currently available, and revise its knowledge and conclusions as new information becomes available. Second, the architecture must enable adaptation to previously unseen teammates using limited data. Unlike pre-coordinated teams, where agents follow predefined protocols, ad hoc agents operate in dynamic environments where teammates may have diverse behaviors, objectives, and strategies. Thus, the agent must be capable of identifying and anticipating the behaviors of other agents and use these inferences to guide its own actions to adapt and collaborate efficiently. Moreover, such behavior models need to be both data-efficient and adaptable, allowing the agent to quickly adjust when team composition changes. Third, the architecture should support anticipation of future tasks and goals in the domain to improve coordination over extended interactions. For example, in the household environment, a human may perform a sequence of interrelated tasks such as cooking, setting the table, and then serving a meal. However, an ad hoc agent may only be aware of the current task. If the agent is able to anticipate likely future tasks, it could assist proactively, improving both immediate responsiveness and long-term efficiency. Finally, the architecture should support transparency, particularly in the agent's internal decision-making process. Since ad hoc agents operate in environments where teammates' expectations and behaviors may vary considerably, it is important that their actions remain predictable and justifiable to others. Thus, the ad hoc agent must be capable of providing explanations of its decisions and beliefs, facilitating trust and transparency in human-agent collaboration. To address these limitations we propose a novel hybrid architecture for AHT, **KAT**: Knowledge-based reasoning and learning for Ad hoc Teamwork, that integrates the complementary strengths of knowledge-based reasoning and data-driven learning.

KAT uses **non-monotonic logical reasoning** (Section 2.1.1) with prior commonsense domain knowledge. Non-monotonic logical reasoning allows previously held conclusions to be revised or withdrawn when new, conflicting information arises. The agent utilizes commonsense domain knowledge in its decision-making, and make assumptions under exceptional circumstances to recover from inconsistencies. This reasoning is implemented using Answer Set Programming (ASP, Section 2.2.1). ASP is a declarative programming paradigm in which domain knowledge is encoded as logical rules, and solutions are computed as answer sets. ASP supports default reasoning and enables the agent to recover from inconsistencies by revising assumptions. We encode

this knowledge once and allow the agent to revise it over time as needed by incrementally acquiring any missing domain knowledge in terms of objects, actions and rules that govern domain dynamics.

Example 1 [*Reasoning with non-monotonic logical reasoning*]

Consider an AI agent collaborating with a human to prepare breakfast in the Virtual-Home domain. The agent’s prior knowledge includes the default statement that eggs are usually found in the fridge. However, during plan execution, the agent observes that the fridge does not contain any eggs. Rather than failing, the agent retracts this default assumption and considers alternative locations, such as the pantry or the kitchen counter, restoring consistency between its beliefs and observations and allowing it to continue planning toward the goal.

Next, to model the behavior of other agents, we draw on **ecological rationality** principles (Gigerenzer, 2020). Ecological rationality is the idea that the quality of a decision depends on how well the decision-making strategy matches the environment. Rather than assuming that more information or complex reasoning always leads to better decisions, ecological rationality argues that simple, adaptive heuristics (*i.e.*, decision-making rules that use limited information to produce effective choices) can be highly effective when they are adapted to the structure of the environment (Section 2.5). Models learned using such heuristics are computationally efficient and can be learned and updated efficiently from limited data, making them well-suited for environments where data is limited or constantly evolving such as AHT settings. Moreover, they are inherently interpretable, allowing users to understand and trust the agent’s decisions.

Example 2 [*Adapting to previously unseen teammates with ecological rationality*]

Consider an ad hoc agent collaborating with a human whose decision-making policy is initially unknown. The agent has learned a behavior model of a typical human teammate from a limited number of prior interactions. When collaborating with a new human who has different preferences and priorities — for example, one who prefers to toast bread before preparing cereal, rather than the other way around — the agent detects that its current model no longer accurately predicts the human’s actions. It then revises the model incrementally based on newly observed behavior, allowing it to adapt its coordination strategy and avoid selecting actions that conflict with or duplicate those of the human, without requiring large amounts of new training data.

In addition, KAT leverage a pretrained **Large Language Model** (LLM, Section 2.6) to anticipate abstract (high-level) future tasks and goals in the domain, as well as to interpret natural language descriptions. LLMs are pretrained neural network models that learn generic patterns from large datasets, enabling them to generate predictions for tasks where manually encoding knowledge across diverse domains and scenarios is not only difficult but impractical. To ensure relevance and accuracy, we guide and adapts these predictions based on prior domain-specific knowledge and experience.

Example 3 [*Task anticipation with LLM*]

Consider an ad hoc agent collaborating with a human on a weekday when the human is working from home. The agent is initially only aware of the current task, *prepare breakfast*. By leveraging a pretrained LLM together with the team’s historical task routines, the agent anticipates that *prepare home work-station*, *prepare coffee*, and *prepare lunch* are likely to follow. The agent can then jointly plan for the current and anticipated tasks — for example, retrieving items from the fridge for both breakfast and lunch in a single trip — reducing the total number of steps and time required to complete the day’s tasks.

Example 4 [*Knowledge acquisition*]

Consider an ad hoc agent collaborating with a human in the VirtualHome domain. The agent’s knowledge base does not yet include the action of opening an appliance, so when it needs to place a book inside a cabinet with a closed door, it is unable to compute a valid plan. During execution, the human describes to the ad hoc agent, *Agent1 cannot put the book inside the cabinet since the cabinet door is closed. Thus, Agent1 opened the cabinet*. The agent uses a pretrained LLM to translate this cue into a new executability condition and causal law, and adds them to its knowledge base with a new action, *open*. Later, while independently exploring the effects of this newly learned action in different initial states, the agent discovers through repeated simulation that switching on an appliance also fails when its door is open, an executability condition it was not directly told about. It incorporates this additional precondition into its knowledge base as well, allowing it to complete the task and accurately reason about similar appliances it encounters in the future.

Lastly, KAT’s use of knowledge-based reasoning and simple, interpretable predictive models provides the foundation for making the ad hoc agent’s internal decision making transparent. Specifically, the agent generates relational descriptions as on-demand explanations of its own behavior and that of other agents, to answer causal

(“Why did you do ...?”), contrastive (“Why did you not do ...?”), belief justification (“Why did you believe ...?”) and counterfactual (“What would have happened if ...?”) questions identified as important in work on explainable planning (Fox et al., 2017). These explanations provide users with insights into the agent’s reasoning process, enhancing transparency and promoting acceptability. Furthermore, generating these descriptions on-demand promotes computational efficiency.

Example 5 [*Explanation generation*]

Consider a scenario in the VirtualHome domain where the ad hoc agent decides to move to the fridge rather than attempting to grab the cereal from the kitchen counter. A human teammate may find this decision unintuitive and query the agent with a causal question such as *Why did you move to the fridge?*. The agent then generates a relational explanation such as *I moved to the fridge because the milk required for preparing breakfast is inside*, providing the human with a justification of the agent’s reasoning and promoting trust in the collaboration.

Within KAT, these techniques and capabilities are integrated into five components: (i) knowledge representation and reasoning, (ii) behavior modeling, (iii) task anticipation, (iv) knowledge acquisition, and (v) explanation generation. These collectively make the following contributions:

- Design and validate a novel architecture for AHT (KAT), that integrates non-monotonic logical reasoning, data-efficient behavior modeling, LLM-based generalization, interactive knowledge acquisition and explainable agency within a single framework for reliable, adaptive, transparent, and scalable collaboration in AHT settings.
- Utilize non-monotonic logical reasoning to represent and reason with prior commonsense domain knowledge, support default reasoning, and enable revision of conclusions in response to new observations, allowing the agent to operate effectively under incomplete and evolving information.
- Incrementally and rapidly learn interpretable models of other agents’ behavior from limited examples based on ecological rationality principles, enabling rapid adaptation to previously unknown teammates from less data and computation.
- Leverage a pretrained LLM to anticipate future high-level tasks to be completed by the team, automatically prompting and adapting the LLM’s output based on

domain-specific knowledge, and assigning the current and anticipated tasks as joint goals to be achieved for better team performance.

- Incrementally acquire previously unknown knowledge in the form of new objects, actions, and axioms, and cumulatively revise existing knowledge, based on LLM-based processing of natural language descriptions of actions and outcomes, and decision tree induction applied to observations, for minimal plans and improved performance for the agent and the team.
- Automatically construct on-demand relational descriptions as *explanations* of the agents decisions and beliefs, and those of other agents in the team, in response to different types of questions (e.g., causal, contrastive, and counterfactual) enhancing the interoperability of the system.

Having outlined the key contributions of the thesis, next we define the assumptions that shape the scope and limitations of our study. We discuss these assumptions in detail in the following chapters.

First, in this study we assume that all actions in the environments are deterministic. We believe it is possible to extend many of our results to stochastic action models by leveraging probabilistic extensions of ASP, and as the architecture is built on non-monotonic logical reasoning which supports default reasoning and revision of conclusions. However, we do not explore or evaluate this capability within the scope of this work. Second, the architecture supports both fully and partially observable environments. In partially observable setting, parts of the environment are hidden from the agent’s view, preventing it from directly observing the true environment state. In some domains, we assume limited communication capabilities, but we do not assume that agents who receive these messages will always comply with the communicated information. Finally, while the environments we evaluate include both discrete and continuous state and action spaces, we discretize them specifically for the reasoning component. Other components of the architecture may operate over a mix of discrete and continuous representations.

In addition to the assumptions above, we frame our approach within the standard assumptions of AHT as described in this survey (Mirsky et al., 2022). First, we assume that the ad hoc agent has no prior coordination with the other agents in the domain. This means, there is no opportunity to establish coordination mechanisms in advance, such as task distribution or joint training. The ad hoc agent may possess partial knowledge about similar teammates, acquired from an expert or previous interactions with

other agents. Second, we assume that the ad hoc agent has no control over its teammates, thus it cannot modify their policies or behavior directly. Third, we assume that the teammates of the ad hoc agent are collaborative, that they share a common goal and are working toward achieving it. While individual teammates may have additional personal objectives, we assume these do not conflict with the team’s primary objective, and they are neither deceptive nor adversarial.

1.3 Thesis Structure

This thesis is organized into chapters, each exploring a different aspect of the architecture, starting with its core design and expanding into more functionalities. Each chapter introduces the key concepts, describe the methodologies used, and analyze results that collectively contribute to addressing the challenges of AHT as discussed in Section 1.2. Below, we outline the key points of the chapters to follow.

Chapter 2: Background

Chapter 2 provides the background and foundational knowledge necessary for this thesis. We begin by introducing the core concepts of logical reasoning, declarative languages, action languages and automated planning, followed by a discussion on ecological rationality and heuristics and LLMs, which form the basis of our proposed architecture. Together, these establish the context necessary for understanding the motivations, design choices, and methodologies presented in the remainder of the thesis.

Chapter 3: Related Work

This chapter discusses the existing research literature in AHT, and previous studies related to various methods employed in the approach proposed in this thesis. We begin by discussing key strategies developed for AHT, focusing on their approaches for knowledge representation, action selection, and adaptation to teammate. We highlight major advancements in the field as well as their limitations. Next we explore the progress made in transparent decision-making through explainable AI. Finally, we conclude the chapter by reviewing research on knowledge acquisition in AI systems. Through this discussion, we situate our proposed approach within the broader research literature and identifying the gaps our work aims to address.

Chapter 4: Core Architecture

Chapter 4 introduces the core components of KAT, our proposed approach for solving the AHT problem. KAT is designed to enable ad hoc agents to collaborate efficiently with unknown teammates in dynamic environments, even in the presence of limited data, partial observability, and evolving team compositions. The architecture consists of five key components, two of which are the focus of this chapter: *knowledge representation and reasoning* component, and the *agent behavior models* component. To leverage commonsense knowledge and overcome the challenges posed by limited training data and resources, KAT integrates non-monotonic logical reasoning with commonsense domain knowledge and ecological rationality based behavior model learning.

We provide experiments to evaluate KAT’s effectiveness in comparison to state of the art data-driven AHT baselines as well as non-monotonic logical reasoning agents. Our results indicate that KAT achieves comparable or better performance, in both simple and complex scenarios, particularly where training data is scarce. We show that KAT’s higher performance not attributed to a single component but rather to the combined use of reasoning and learning, which enables greater adaptability and robustness.

Chapter 5: Task Anticipation and Partial Observability

In this chapter, we introduce the third component of KAT: *task anticipation*. KAT employ a pretrained LLM to anticipate the future tasks in the domain, allowing the agent to jointly plan for both current and future goals, improving adaptation and performance. We show that leveraging LLMs for task anticipation together with prompts that are incrementally updated using domain-specific knowledge and validators that check the anticipated tasks, improves team efficiency, allowing the agent to better align its actions with its teammates’ long-term goals.

The latter part of this chapter explores and evaluates the scalability of our architecture, particularly in settings where coordination becomes more challenging. Finally, we extend KAT beyond fully observable environments to operate under partial observability and limited communication and indicate its performance in two separate domains.

Chapter 6: Knowledge Acquisition

This chapter focuses on the fourth component of KAT: *knowledge acquisition*. We describe KAT’s approach to acquiring previously unknown knowledge in terms of objects, action and axioms (executability conditions and causal laws) while collaborating with other agents in complex domains. Specifically, KAT leverages a pretrained LLM to translate natural language descriptions of action outcomes and failures into formal Answer Set Prolog representations and use decision tree induction to incrementally revise knowledge based on observations. Acquiring this knowledge helps reduce uncertainty, refine decision-making, and enable the agent to adapt to evolving environments without requiring extensive retraining. Through experiments in three domains, we evaluate the accuracy and consistency of KAT’s acquired knowledge and indicate that incorporating newly acquired knowledge improves the agent’s and team’s performance.

Chapter 7: Explanation Generation

This chapter presents the fifth and final component of KAT: *explanation generation*, which is designed to enhance the transparency of the system. KAT’s methodology allow it to generate relational descriptions of its decisions and beliefs, and those of other agents, as explanations in response to different types of question. This allows others to understand the reasoning behind the ad hoc agent’s decisions, making the decision-making process interpretable and accountable. In our experiments, we evaluate the accuracy and consistency of the generated explanations, demonstrating KAT’s ability to explain its reasoning effectively across a variety of scenarios.

Chapter 8: Conclusion

In the final chapter, we summarize the key findings and contributions of this thesis, highlighting the impact of our proposed approach on addressing the challenges of AHT. In the discussion, we analyze the limitations of our approach, identifying aspects where further improvements are needed. We then explore future research directions aimed at addressing these limitations.

This thesis builds on the following publications, where parts of the research have appeared.

- Back to the Future: Toward a Hybrid Architecture for Ad Hoc Teamwork. In

the AAAI Conference on Artificial Intelligence (AAAI), Washington DC, USA, February 7-14, 2023 (Chapter 4).

- Knowledge-based Reasoning and Learning under Partial Observability in Ad Hoc Teamwork. In Theory and Practice of Logic Programming journal (TPLP), 2023 (Chapter 4 and 5).
- Thinking Together, Apart: Reasoning and Learning in Ad Hoc Teamwork. In the Annual Conference on Advances in Cognitive Systems (ACS), Atlanta, Georgia, USA, October 13-15, 2025 (Chapter 5).
- Reasoning with Commonsense Knowledge and Decision Heuristics for Ad hoc Human-Agent Collaboration. In the International Conference on Distributed Artificial Intelligence (DAI), London, UK, November 21-24, 2025 (Chapter 6).
- Explanation and Knowledge Acquisition in Ad Hoc Teamwork. In the International Symposium on Practical Aspects of Declarative Languages (PADL) at POPL, London, UK, January 15-16, 2024 (Chapter 6 and 7).
- Reasoning and Explanation Generation in Ad hoc Collaboration between Humans and Embodied AI. In the International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR), Texas, USA, August 11-14, 2024 (Chapter 7).

Chapter 2

Background

This chapter provides the background necessary to understand the approach proposed in this thesis. Specifically, the proposed approach draws on concepts from *knowledge representation*, *non-monotonic reasoning*, *automated planning*, and *ecological rationality*. We begin by introducing logical reasoning (Section 2.1) and discuss its role in representing domain knowledge and supporting reasoning. We then review non-monotonic logical reasoning (Section 2.1.1) which was developed to overcome the limitations of classical logic when reasoning with incomplete and evolving information. Since reasoning about knowledge has also motivated modal logics, we next review modal logic (Section 2.1.2), which provide formal mechanisms for representing agents’ knowledge and belief in multi-agent systems. We then review declarative programming languages (Section 2.2), ASP (Section 2.2.1), and action languages (Section 2.3), which together provide the theoretical and computational foundation for the architecture proposed in this thesis. To position the proposed approach within the broader planning literature, we then review major planning paradigms, including classical planning (Section 2.4.1) and planning under incomplete information (Section 2.4.2). We also briefly discuss related multi-agent reasoning techniques, including plan recognition (Section 2.4.3) and negotiation (Section 2.4.4) highlighting how these approaches differ from the objectives of our work. Finally, we discuss ecological rationality and application of heuristics (Section 2.5) with a focus on fast-and-frugal trees (Section 2.5.1), which motivate our choice of methods for modeling the behavior of other agents. We conclude the chapter with an overview of LLMs (Section 2.6), whose capabilities are leveraged in our architecture to interpret natural language descriptions and anticipate high-level future tasks in the domain.

2.1 Logical Reasoning

Logical reasoning has long provided a formal foundation for representing knowledge and deriving conclusions from it (Brachman and Levesque, 2004; Genesereth and Nilsson, 1990). In AI, logic enables an agent to describe properties of the world, reason about their consequences, and select actions that achieve its objectives. Logic-based agents usually comprise two components: a knowledge base that stores information about the domain and a reasoning mechanism that derives conclusions from this knowledge together with observations obtained from the environment. Such agents have been successfully applied to a wide range of intelligent tasks, including planning, diagnosis, and scheduling (Levesque and Lakemeyer, 2008).

In the 1950s, John McCarthy and others laid the foundation to *logic based approach to AI* by applying axiomatic method to formalize the reasoning problems in AI (McCarthy, 1959). They proposed to use a language of mathematical logic to provide a computer program with sufficient knowledge about the problem domain and use logical inference to determine which actions should be performed to achieve their goal. Since then, many researchers have been applying logic based approaches to design intelligent agents.

Early logic-based systems relied primarily on **classical logic** for representing such knowledge. Built on propositional and first order logic, classical logic offered a well-defined syntax and semantics for representing knowledge and powerful inference mechanisms. To this day, it remains a fundamental reasoning framework for domains in which the environment can be described completely and accurately.

Nevertheless, pioneers of logical reasoning soon recognized that classical logic was inadequate for representing commonsense knowledge which is often incomplete, uncertain, and defeasible (i.e., conclusions can be overridden by new information). Everyday reasoning frequently relies on default assumptions that may later need to be withdrawn when new information becomes available. Such defeasible knowledge cannot be expressed using classical logic due to its deductive and monotonic nature (once a conclusion has been logically derived, adding more premises will not invalidate that conclusion).

A classic example used in the literature to illustrate this is the reasoning that birds can fly. This can be formalized in classical logic as:

$$\forall x(Bird(x) \rightarrow Flies(x)) \quad (2.1)$$

which states that “All birds fly”. While generally true, this fails in cases such as penguins or ostriches, who cannot fly. One possible remedy is to explicitly encode these exceptions as:

$$\forall x(Bird(x) \wedge \neg Penguin(x) \wedge \neg Ostrich(x) \rightarrow Flies(x)) \quad (2.2)$$

which implies “Birds fly unless they are penguins or ostriches”. In other words, expressing this in classical logic requires explicitly specifying every possible exception to the axioms, such as penguins, ostriches, birds with broken wings, dead birds who cannot fly. As new exceptions are discovered, the logical theory must continually be modified to accommodate them. This is impractical in open-world situations, where we rarely know all possible exceptions and complete information in advance.

To overcome such incomplete knowledge situations, in real world reasoning we often make assumptions that may turn out to be wrong later on with the availability of new, more accurate information. The old (prior) conclusions will then be retracted to support the new information that appeared. For example, introducing $Bird(Tweety)$ to the above example allow us to infer $Flies(Tweety)$ indicating Tweety flies. However, if we discover later that Tweety has a broken wing, we need to retract this conclusion and devise the new conclusion that Tweety cannot fly anymore. This violates the monotonicity of classical logic, which assumes a settlement to be either true or false and adding new premises can only produce additional conclusions; it cannot invalidate conclusions that have already been derived. This inability to represent defaults, exceptions, and the revision of beliefs limits the applicability of classical logic to many real-world reasoning problems, particularly those involving incomplete or evolving knowledge. To reason effectively in such environments, we require representations that permit tentative conclusions to be withdrawn when more accurate information becomes available.

These limitations motivated two important directions of research in knowledge representation. The first led to the development of non-monotonic logics, which extend classical logic with mechanisms for reasoning by applying defaults and revising conclusions in the presence of new information. The second led to modal logics, which extend classical logic with operators for reasoning about concepts such as necessity, possibility, knowledge, and belief.

In the next section we survey the evolution of non-monotonic logical reasoning in AI.

2.1.1 Non-Monotonic Logical Reasoning

The limitations of classical logic motivated the development of **non-monotonic logics**, a family of formalisms that allow conclusions to be withdrawn when new information becomes available and supports defeasible reasoning. This ability makes non-monotonic reasoning particularly suitable for modelling commonsense reasoning, where knowledge is often incomplete and exceptions are common. Over the past several decades, a number of influential non-monotonic formalisms have been proposed. Here we briefly review the developments most relevant to this thesis, referring the reader to Reiter (1987) for a comprehensive survey.

One of the earliest attempts to represent incomplete knowledge was to extend classical logic with Closed World Assumptions (CWA) (Reiter, 1980). Under CWA, any statement that cannot be proven true from the existing knowledge was assumed to be false - effectively ‘closing’ the world and treating the available knowledge as complete. For instance, in the birds example, if we introduce $Bird(Tweety)$ but does not explicitly state whether Tweety flies ($Flies(Tweety)$), under CWA the knowledge base will conclude that Tweety does not fly, $\neg Flies(Tweety)$, treating birds whose flying ability is unknown as non-flying birds, unless specified otherwise. While this can simplify reasoning, especially by allowing implicit inference of negative information without having to explicitly represent all the negative facts, the absence of a fact in the knowledge base alone is not always sufficient to conclude its negation. In many real-world domains, failing to derive a fact may reflect incomplete knowledge rather than its falsity. Thus, CWA can lead to incorrect conclusions in some scenarios.

Another approach for reasoning with incomplete knowledge is *circumscription*, introduced by McCarthy (1980). Circumscription is a form of non-monotonic logic that minimizes the extension of predicates to infer defaults. The correctness of conclusions drawn through circumscription depends on having considered all relevant facts and applying the minimization of predicates to assume as little abnormality as possible. For instance, using circumscription, the previous example can be extended with a predicate $Ab(x)$ (i.e., standing for ‘abnormal’) to represent:

$$\forall x (Bird(x) \wedge \neg Ab(x) \rightarrow Flies(x)) \quad (2.3a)$$

$$\forall x (Penguin(x) \rightarrow Ab(x)) \quad (2.3b)$$

which states that “all birds fly unless they are abnormal”, and that penguins are abnormal. Introducing $Bird(Tweety)$ now will lead to the conclusion $Flies(Tweety)$, since

Tweety is not specified to be abnormal, nor a penguin. Thus, Tweety is considered to be normal unless there is explicit evidence indicating otherwise. If later discovered that Tweety is a Penguin and $Penguin(Tweety)$ was added to the program, Tweety becomes abnormal and $Flies(Tweety)$ is not concluded.

At the same time, Reiter (1980) proposed, *default logic*, which introduced default rules that can be applied unless they conflict with known facts. For instance, following default rule can be applied in the birds example:

$$\frac{Bird(x) : Flies(x)}{Flies(x)} \quad (2.4a)$$

$$\forall x(Penguin(x) \rightarrow \neg Flies(x)) \quad (2.4b)$$

which implies that “If x is a bird, and it is consistent to assume that x flies, then infer that x flies”. $Bird(x)$ here is the prerequisite, and $Flies(x)$ above the line is the justifications, and $Flies(x)$ below the line is the consequent of the default. This states that if $Bird(x)$ holds and there is no evidence of the justification $Flies(x)$ to be false, then infer $Flies(x)$. The default rule here acts as a mapping from incomplete information to a more complete extension of the knowledge by partly filling the gaps with plausible conclusions. Introducing, $Bird(Tweety)$ to the above, lead to the conclusion $Flies(Tweety)$. Moreover, learning Tweety is a Penguin later on, $Penguin(Tweety)$ will lead to the conclusion $\neg Flies(Tweety)$.

Another non-monotonic logic closely related to default logic is Moore (1985)’s *autoepistemic logic*, which was based on McDermott and Doyle (1980)’s proposal for non-monotonic logic. McDermott and Doyle (1980) based their non-monotonic logic on modal logics capturing the notion of belief. They modified first order logic to include an operator M which represent “is consistent”, and an inference rule, “infer Mp from the inability to infer $\neg p$ ” which states that Mp is derivable if $\neg p$ is not derivable. In other words, consider a formula is consistent if its negation is not believed.

$$\forall x(Bird(x) \wedge M Flies(x) \rightarrow Flies(x)) \quad (2.5)$$

Adding $Bird(Tweety)$ to above infers $Flies(Tweety)$, based on the belief “Tweety is a bird, and only birds inferred not to fly are unable to fly. As Tweety is not inferred not to fly, it must be true that Tweety can fly”.

Moore (1985) argued that this form of reasoning is more related to autoepistemic logic than default logic, as it reflects agent’s beliefs about its own beliefs. In default logic, reasoning proceeds as “Birds normally fly, Tweety is a bird, therefore, Tweety

probably flies”. However, in autoepistemic logic, reasoning proceeds as “if Tweety were a bird that could not fly, then I would know about it. Since I have no such knowledge, I conclude that Tweety must be able to fly”.

Autoepistemic logic extends propositional logic with the modal operator L , which represents “is believed”. Given some premises, an autoepistemic theory T is a set of formulas (axioms), and its intended models E are called stable expansions of T iff,

$$E = Cn(T \cup \{L\phi | \phi \in E\} \cup \{\neg L\phi | \phi \notin E\}) \quad (2.6)$$

where Cn is a propositional consequence operator, and E is a set of formulas which represent a set of beliefs the agent could hold after reflecting on what it believes and what it does not. Starting from T , the agent assumes that it believes exactly ϕ that are in E and assume it does not believe all others. For every formula ϕ , either $L\phi \in E$ or $\neg L\phi \in E$. If $\phi \in E$ then $L\phi \in E$; if $\phi \notin E$ then $\neg L\phi \in E$. A formula F is true in T if F belongs to all stable expansions of T . The previous example can be written using autoepistemic logic as:

$$\forall x (Bird(x) \wedge \neg L \neg Flies(x) \rightarrow Flies(x)) \quad (2.7)$$

which implies “If x is a bird, and it is not believed that x does not fly, then infer that x flies”. Adding $Bird(Tweety)$ to this, lead to the expansion E that includes $Flies(Tweety)$. Expanding it to include all beliefs in E will lead to $LFlies(Tweety)$. All beliefs not in E will lead to $\neg LFlies(Tweety)$. Considering 2.6, this will conclude $Flies(Tweety)$, Also adding $Penguin(Tweety)$ will infer $\neg Flies(Tweety)$.

These frameworks laid the foundation to *stable model semantics* proposed by Gelfond and Lifschitz (1988). Under stable model semantics, the expression “*not p*” is interpreted as “ p is not believed” when p is not known or derivable. Using stable model semantics, the birds example from before can be formulated as:

$$Flies(x) \leftarrow Bird(x), not Ab(x) \quad (2.8a)$$

$$Ab(x) \leftarrow Penguin(x) \quad (2.8b)$$

$$Bird(x) \leftarrow Penguin(x) \quad (2.8c)$$

Here $not Ab(x)$ represent the default negation under stable model semantics. Adding $Bird(Tweety)$ to the above lead to the conclusion $Flies(Tweety)$ as there is no information that allows the program to derive $Ab(Tweety)$. Also, adding $Penguin(Tweety)$ to the program later on infers $Ab(Tweety)$ from the equation 2.8b. At the same time, as

equation 2.8a is not satisfied, $Flies(Tweety)$ is not derived, and the stable model will only contain $\{Bird(Tweety), Penguin(Tweety)\}$.

Stable model semantics provided the formal foundation for ASP (Section 2.2.1), a declarative programming paradigm in which reasoning problems are encoded as logic programs and solved by computing stable models. This makes ASP particularly well suited to the work presented in this thesis, as it provides a unified framework for both representing domain knowledge and generating plans.

2.1.2 Modal Logic

Alongside the development of non-monotonic formalisms discussed above, another line of research has focused on formally representing an agent’s knowledge and belief using **modal logic**. Modal logic extends classical logic with operators that express whether a statement is true (i.e., necessary, possible, known, or believed). Several formalisms and interpretations of these operators have been proposed over time leading to a family of logics with similar rules and different symbols (Chellas, 1980; Zach, 2019).

One of the earliest formal treatment of modal logic was introducing *strict implication* by Lewis and Landford (1934). Lewis argued that “if A then B ” (i.e., $A \rightarrow B$) in classical logic, which is false only when A is true and B is false is a poor model of “if...then” in natural language, since it makes a conditional true whenever A is false, even if A and B are entirely unrelated. They instead proposed that “if A then B ” should mean that it is impossible for A to be true while B is false (i.e., necessarily, A implies B). They formalized this idea using a set of axioms and inference rules, and later developed five modal systems S1 through S5 (Lewis and Landford, 1934).

Later, Carnap (1947) proposed defining a statement as necessarily true, if it is true in every possible state description (i.e., a complete and consistent specification of the world’s basic facts), providing one of the earliest attempts to define semantics for modal logic. Although this provided an important semantic interpretation of necessity, it was later recognized that the approach does not adequately handle iterated modalities, that is, formulas containing nested modal operators such as “possibly, it is necessary that possibly A ”. Specifically, the semantics does not introduce explicit accessibility relations between possible worlds and as a result, nested modal operators could not naturally be interpreted relative to different worlds.

Later, Kripke (1963) developed relational possible-world semantics for modal logic

by introducing an accessibility relation between worlds. Specifically, a Kripke model consisted of a set of possible worlds together with a relation specifying which worlds are accessible from each world. Under this interpretation, a formula $\Box A$ is true, at world w , exactly when A is true at every world accessible from w . The semantics itself does not specify which worlds are accessible from which others; this is determined by the accessibility relation chosen for the model. By imposing different properties on the accessibility relation, one can obtain different modal systems.

Since the accessibility relation could be interpreted in different ways depending on the application, researchers reused Kripke (1963) framework by interpreting $\Box$ operator in different ways, depending on what they wanted to reason about. One of the first and most influential such reinterpretations was epistemic logic by (Hintikka, 1962) who replaced Kripke’s accessibility relation as an agent’s epistemic accessibility relation: the worlds that, given what the agent currently knows, could still turn out to be the real one. He defined a knowledge operator $K_i\phi$ (“agent i knows that ϕ ”) to be true at a world exactly when ϕ is true at every one of i ’s epistemic alternatives at that world (i.e., an agent is said to know a fact if that fact holds in every world it considers possible). The same construction, with “knows” replaced by “believes”, gives a logic of belief (doxastic logic). Epistemic logic has been widely used for reasoning about multi-agent systems, where different agents possess different information and may reason about one another’s knowledge. Fagin et al. (1995) later extended this framework to express not only what individual agents know, but also what they know about one another’s knowledge.

While epistemic logic provides a powerful framework for representing an agent’s knowledge at a particular point in time, many applications require reasoning about how knowledge changes as actions occur. This motivated the development of Dynamic Epistemic Logic (DEL) (Baltag and Moss, 2005), which extends epistemic logic with formal models of information-changing actions such as public announcements, private communications, observations, and sensing actions. These actions transform the agents’ epistemic states by modifying the accessibility relations between worlds and provide a formal framework for reasoning about how agents acquire, revise, and share knowledge during interaction.

Modal logics of knowledge have become an important foundation for reasoning under uncertainty in multi-agent systems and have strongly influenced research on epistemic planning, where planning algorithms reason explicitly about the knowledge and beliefs of multiple agents. However, their primary objective is to reason about

agents' informational states rather than to perform non-monotonic reasoning about commonsense domain knowledge. In contrast, our approach presented in this thesis represents domain knowledge using ASP, allowing an ad hoc agent to reason with defaults, revise conclusions as new information becomes available, and generate plans under incomplete knowledge about domain and other agents.

2.2 Declarative Languages

While non-monotonic reasoning and modal logic focused on extending the expressive power of logic, logic programming emerged as another direction of research to investigate how logic itself could serve as a programming language. Pioneering work by Green (1969); Hayes (1973); Kowalski and Kim (1991) and many others explored how logic could serve as a language for representing knowledge, by integrating logic with techniques from automated deduction theory and constructive logic (Baral and Gelfond, 1994). This approach became known as logic programming and led to the development of the first logic programming language, Prolog (Kowalski, 1974; Colmerauer and Roussel, 1993), which remains one of the most influential languages in artificial intelligence and knowledge representation.

Prolog restricted first order logic to Horn clauses and supported computation via linear-input resolution and backward-chaining. In Prolog, all knowledge is expressed as Horn clauses, which are either facts (atomic formulas) or rules (implications of atomic formulas) as below:

$$P_0 \leftarrow P_1 \wedge P_2 \wedge \dots P_n \quad (2.9)$$

In linear-input resolution, at each derivation step, for each resolvent, one of the parent clauses must be either an initial input clause or an ancestor of the current clause. The search strategy, backward-chaining, ensures that the reasoner starts from the current query or goal and works backwards instead of starting from facts and generating consequences.

Prolog is a declarative programming language. Declarative languages allow the user to specify what needs to be computed, rather than how to compute it. For example, a user can describe a domain in Prolog by defining its objects, attributes, relations, and rules governing domain dynamics. Once defined, this knowledge can be used by an inference engine to perform tasks such as planning, diagnosis and generate explanations.

Later versions of Prolog incorporated *negation as failure* (Clark, 1987), introducing non-monotonic features to declarative languages for handling incomplete information. Negation as failure allowed a statement “*not p*” to be assumed *true* if there is no proof of “*p*”, that is, if “*p*” cannot be derived from the current knowledge base. This allows reasoning with incomplete knowledge and closely resembles the notion of default reasoning, where conclusions may be drawn in the absence of evidence to the contrary and later withdrawn if new information becomes available. This development of negation as failure inspired ASP, a declarative problem-solving paradigm based on the stable model semantics (Gelfond and Lifschitz, 1988).

2.2.1 Answer Set Programming

ASP emerged as a powerful declarative paradigm rooted in logic programming and non-monotonic reasoning. It allowed modeling of complex problems as logic programs whose solutions correspond to its answer sets (stable models) (Gelfond and Lifschitz, 1988). This enabled complex reasoning tasks to be expressed compactly while allowing efficient solvers to perform the underlying search automatically.

ASP programs are written as collections of facts and rules, and may be expressed in languages such as Answer Set Prolog (Gelfond and Kahl, 2014). As a declarative language, Answer Set Prolog allows users to describe a domain using a set of atoms (atomic statements) of the form $p(t_1, \dots, t_n)$, where p is a predicate (relation) symbol and $t_1, \dots, t_n$ are terms (variables, object constants). A literal l , can be an atom or its negation, $\neg p(t_1, \dots, t_n)$. A rule in Answer Set Prolog takes the form of:

$$l_0 \text{ or } \dots \text{ or } l_i \leftarrow l_{i+1}, \dots, l_m, \text{not } l_{m+1}, \dots, \text{not } l_n \quad (2.10)$$

where the left-hand side forms the *head* of the rule and the right-hand side forms its *body*. The body consists of positive literals together with default-negated literals preceded by the keyword “*not*”. The symbol “*not*” represent default negation (also called negation as failure) and is read as “it is not believed that l is true”. Unlike classical negation, written $\neg l$, which explicitly asserts that a proposition is false, default negation, expresses only that l is not believed or cannot be derived from the available knowledge. Thus, when an atom is absent from a set, this does not in itself mean that its negated form is present. In other words, the absence of l from an answer set does not imply that $\neg l$ is true. The absence of both the atom and its default negation in the set, only imply that there is no support for l being *true*, not that l is believed to be *false*.

Using this, an incomplete state of knowledge can be described using a consistent but possible incomplete set of literals, supporting non-monotonic capabilities.

The disjunction “*or*” here refers to epistemic disjunction, where a statement l_0 *or* l_1 is read as “ l_0 is believed to be true or l_1 is believed to be true”. This is different from the classical disjunction $\vee$, where the formula $l \vee \neg l$ of propositional logic, is a tautology, since every proposition l is either true or false. However, in Answer Set Prolog, the statement l *or* $\neg l$, is not tautology, as it expresses the agent’s *belief*. A rational agent may remain undecided about the truth or falsity of l , neither believing l , nor its negation. Thus, Answer Set Prolog, allows reasoning with incomplete or uncertain knowledge. It uses default negation to represent lack of belief in l , without enforcing that every atom must be definitively true or false, enabling reasoning based on absence of information.

Answer Set Prolog solving is based on fast search techniques inspired by satisfiability solvers (Lifschitz, 2008). This makes ASP particularly suited for solving difficult search problems. Such problems are then reduced to computing stable models with answer set solvers used to perform the search and generate the solutions.

An answer set of a logic program represents a consistent set of beliefs with respect to the program. Each answer set contains all grounded atoms (i.e., atoms whose variables are replaced with its object constants) that are true in that model. Depending on the program, there may be multiple answer sets (representing alternative solutions) or none at all (indicating inconsistency). The valid consequences of the program are those statements that hold in all of its answer sets. Many reasoning tasks can be formulated as computing answer sets or deriving consequences from them. There are many ASP inference engines (ASP solvers) capable of computing answer sets of logic programs with millions of ground rules (Gelfond and Kahl, 2014). Notable systems include Smodels (Niemelä and Simons, 1997), Clingo (gringo+claspD) (Gebser et al., 2017), and DLV (Leone et al., 2006).

ASP has been used by an international research community to reason with incomplete knowledge with the help of modern ASP solvers that have proven to be efficient in handling large knowledge bases (Cao Tran et al., 2023). For instance, ASP has been successfully applied in decision support for space shuttle (Nogueira et al., 2001) and inference of phylogenetic trees (Brooks et al., 2007). Moreover, Answer Set Planning has emerged as an active research direction, where ASP is used to solve planning problems, which are translated to logic programs. The resulting answer sets of the logic program correspond to plans that transform an initial state of the world into a desired

goal state. Answer Set Planning has been applied across a wide range of domains, including cognitive robotics (Erdem and Patoglu, 2018), planning, and diagnostics in housekeeping robots (Erdem et al., 2012), coordination of unmanned aerial vehicles in dynamic environments (Balduccini et al., 2014), multi-agent path finding (Nguyen et al., 2017), planning with preferences (Son and Pontelli, 2004), and robot task planning (Jiang et al., 2019), among many others, as discussed by Cao Tran et al. (2023).

The action languages discussed in the following section build directly upon this idea by providing high-level formalisms for describing dynamic systems whose semantics can be translated into Answer Set Prolog programs for automated reasoning and plan generation.

2.3 Action Languages

While Answer Set Programming provides a powerful framework for representing knowledge and performing non-monotonic reasoning, it does not by itself specify how dynamic systems should be modelled. Planning requires a formal description of how actions transform the world, including their preconditions, direct effects, indirect effects, and persistence over time. Action languages were developed to provide this high-level representations, allowing dynamic domains to be described independently of the underlying reasoning engine.

Early approaches represented dynamic systems using transition diagrams, where nodes correspond to possible states of the world and edges represent the execution of actions that transform one state into another. However, for complex practical systems, transition diagrams quickly become large and impractical to construct or maintain.

To address this limitation, *action languages* were introduced as formal tools for describing transition diagrams compactly (Gelfond and Lifschitz, 1999). They are formal models of parts of natural language for describing transition diagrams of dynamic systems. Rather than explicitly enumerating every possible state and transition, an action language describes the dynamics of a domain using a system description consisting of causal laws, executability conditions, state constraints, and inertia axioms. Simply, its a collection of statements that provide a clear and compact way to capture the possible states of the system and the effects of actions. Moreover, their formal semantics can be easily translated into Answer Set Prolog representations, allowing ASP solvers to reason about dynamic domains. Given an initial state and a desired goal, computing an answer set to this representation corresponds to reasoning about the system such

as finding a sequence of actions (plans) that transforms the initial state into a goal state. Specifically, action languages provide a high-level modelling language, while ASP provides the underlying reasoning mechanism used to solve planning and diagnosis problems. Throughout this thesis, we adopt this paradigm to represent domain dynamics and compute plans.

There has been considerable research in developing action languages for single- and multiagent domains. STRIPS is one of the earliest action languages (Fikes and Nilsson, 1971) that formalized the representation of actions in terms of preconditions and effects. STRIPS demonstrated that planning problems could be specified declaratively, but lacked mechanisms for representing indirect effects, default assumptions, and complex domain constraints. Subsequent work led to the development of richer action languages, including $\mathcal{A}$ (Gelfond and Lifschitz, 1993), $\mathcal{B}$ (Gelfond and Lifschitz, 1999), $\mathcal{ADL}$ (Pednault, 1994) and PDDL (McDermott et al., 1998). These languages progressively extended the expressive power of action languages available for describing dynamic systems, supporting features such as inertia, indirect effects through state constraints, and richer domain descriptions. Different languages emphasize different modelling capabilities and are therefore been used by a wide research community in specific applications based on their expressiveness, the types of actions they can represent, and the domains for which they are best suited (Brenner and Nebel, 2009; Khandelwal et al., 2014).

This line of work was subsequently extended through languages such as $\mathcal{AL}$ (Turner, 1997), $\mathcal{AL}_d$ (Gelfond and Incezan, 2013a) and $\mathcal{ALM}$ (Incezan and Gelfond, 2016). Among these, $\mathcal{AL}_d$ provides a clear representation of dynamic domains based on actions, fluents, causal laws, and state constraints, while accommodating defaults and exceptions thus supporting non-monotonic characteristics. Moreover, $\mathcal{ALM}$ introduced hierarchical modelling, allowing actions and fluents to be defined in terms of other actions and fluents, thereby improving the scalability and maintainability of large system descriptions. The combination of action languages with ASP has enabled a wide range of reasoning tasks beyond plan generation. Since system descriptions provide an explicit model of domain dynamics, the same representation can be used for planning, diagnosis and explanation.

A large and growing body of literature has explored the use of the action language $\mathcal{AL}_d$ for developing applications in single agent systems (Gomez et al., 2020; Sridharan et al., 2019; Blount et al., 2014). As research shifted towards multi-agent domains, action languages were extended to model additional action types to capture interactions

between multiple agents. For example, action language $\mathcal{A}$ has been used for computing cooperative actions for agents in multiagent domains (Son and Sakama, 2010a,b). Cooperative actions are requests or offer actions which enable agents to request something from other agents or to establish conditions for other agents. These actions are used to form joint plans between multiple agents. Similarly, the action language $\mathcal{C}$ has been used to model benchmark multi-agent domains with only minimal extensions (Baral et al., 2010b). The need to reason explicitly about agents' knowledge and beliefs during planning has motivated the development of epistemic action languages, such as $m\mathcal{A}^*$ (Baral et al., 2022). $m\mathcal{A}^*$ model actions that change the physical environment as well as actions that can manipulate the information available to different agents (Baral et al., 2010a). For example, there are world-altering actions (actions that change the physical world), sensing actions (actions that change an agents knowledge without changing the physical world) and announcement actions (actions that publicly communicate information, changing agents beliefs). These developments established a connection between action languages and epistemic reasoning by allowing planners to reason about both changes in the world and changes in agents' knowledge.

Researchers have investigated both centralized and distributed planning architectures using action languages. In centralized planning the reasoning and planning is carried out via a single centralized system which generates plans for multiple agents (Son and Balduccini, 2018). In contrast, distributed planning approaches allow individual agents to construct plans and coordinate through communication and conflict resolution. For example, action language $\mathcal{B}$ has been combined with Prolog and Answer Set Prolog to implement a distributed multi-agent planning system that supports communication in a team of collaborative agents (Son et al., 2010). A centralized controller was used to analyze and merge initial plans generated by smaller groups and resolve conflicts. More recent research has employed $\mathcal{B}$ in distributed planning for non-cooperative or partially collaborative agents using negotiations (Son and Balduccini, 2018).

Motivated by the approaches discussed above for representing and reasoning with domain knowledge in dynamic environments, we adopt these methods to address the ad hoc teamwork problem, where agents must operate with limited knowledge of both the environment and their teammates. In particular, we use non-monotonic logical reasoning to represent and reason with commonsense domain knowledge enabling an ad hoc agent to make decisions under incomplete and evolving information and revise its conclusions as new information becomes available. We use Answer Set Prolog to rep-

resent domain knowledge and perform reasoning tasks such as planning and diagnosis, supporting robust decision-making in ad hoc teamwork.

2.4 Automated Planning

Once a domain has been described using actions, fluents, and axioms (as discussed in the previous sections), a reasoning mechanism is required to determine which actions should be executed, and in what order, to achieve a desired goal. This task is addressed by automated planning. Specifically, given an initial state, a description of the available actions, and a goal specification, a planning system computes a sequence of actions that transforms the initial state into the goal state (Ghallab et al., 2004). Many planning paradigms have been proposed over time – from classical planning, which assumes complete knowledge of a deterministic world, to approaches capable of reasoning under incomplete information and in multi-agent environments.

2.4.1 Classical Planning

Classical planning considers the simplest form of the planning problem, where the environment is assumed to be fully observable and deterministic, with no unexpected events occurring during plan execution. The planning agent has a complete and accurate model of the world, and every action has deterministic effects. Under these assumptions, planning problems are commonly represented using formalisms such as STRIPS (Fikes and Nilsson, 1971) and PDDL (McDermott et al., 1998), and solved using algorithms that treat planning as a search problem over the space of possible world states (Ghallab et al., 2004).

Early researchers used algorithms that searched this state space directly, where the planning problem is represented using a search tree whose nodes represent a world state and edges represent actions. The planner searches for a sequence of actions that transforms the initial state into a state satisfying the goal by traversing the space of reachable states. A search strategy determines the order in which states are explored during this search process. Common strategies used are forward search where the search is performed forward from the initial state, backward search where the search is performed backward from the goal, breadth-first search, depth-first search, best-first search and A* (Ghallab et al., 2004; Russell and Norvig, 2020).

Others have proposed algorithms that searched the space of partial plans instead

of world states, incrementally adding actions and ordering constraints without committing to a complete ordering of actions unless required (Sacerdoti, 1975; Tate, 1977; McAllester and Rosenblitt, 1991). This allowed independent actions to remain unordered until necessary, reducing the number of alternative plans that needed to be explored and often improving planning efficiency.

However, as Bylander (1994) showed in his work, determining whether a plan exists for a given propositional STRIPS planning problem is PSPACE-complete in general. The number of possible states can grow exponentially with the size of the planning problem, making the search space extremely large even for moderately sized planning problems and the computationally infeasible.

To improve scalability, later work introduced heuristics to guide the search (Bonet and Geffner, 2001). A heuristic is an evaluation function that estimates the cost from the current state to the goal. Rather than exploring every possible state equally, heuristics allow the planner to prioritize the most promising search directions, substantially reducing the amount of search required. For example, the Fast-Forward (FF) planning system estimates how many actions remain to reach the goal from a given state by temporarily ignoring the delete effects of actions, thus reducing the planning problem to a simpler (relaxed) problem (Hoffmann and Nebel, 2001). This relaxed version of the planning problem is then used to derive a heuristic that guides ordinary forward state-space search towards the goal.

In a different line of work, Blum and Furst (1997) introduced *Graphplan*. Instead of directly searching the state space, Graphplan first constructs a planning graph consisting of alternating layers of facts (fluents) and the actions they enable, and then searches this graph backwards to extract a valid plan. Representing the planning problem in this way often reduced the amount of search required and considerably improved planning efficiency in practice.

Others have formulated planning problem as a satisfiability problem by encoding a bounded-length planning problem as a propositional formula whose satisfying assignments correspond to valid plans (Kautz and Selman, 1992). SAT solvers have made this approach much efficient, allowing many planning problems to be solved more effectively than with explicit state-space search.

Although these approaches differ in the way they search for plans, they all assume complete knowledge of a deterministic environment. However, in many real-world problems the agents may need to reason with incomplete information and uncertainty before deciding which actions to execute. This motivated the development of planning

approaches for reasoning under incomplete information.

2.4.2 Planning under Incomplete Information

Classical planning assumes that the planner has complete knowledge of the initial state, the environment is fully observable and static, and actions have deterministic effects. In many real-world domains, however, these assumptions do not hold. The planning agent may have incomplete knowledge of the environment and its actions may have uncertain or unobservable effects. This motivated the work on planning under incomplete and uncertain information.

One of the earliest extensions of classical planning to address incomplete knowledge is conformant planning. Conformant planning assumes that the initial state is only partially known and that the agent cannot acquire additional information through sensing during plan execution. Instead the planner must generate a single plan that is guaranteed to achieve the goal regardless of which one the actual initial state is. For example, Smith and Weld (1998) extended GraphPlan (Blum and Furst, 1997) to build separate plan graphs for each possible initial state and search all graphs simultaneously with forward search until find a plan.

Bonet and Geffner (2000) formulated planning with incomplete information as a problem of heuristic search (A^*) in belief states. Belief states represent the set of all world states that are consistent with the planner's current knowledge. Rather than searching over individual world states as in classical planning, their planner searched over belief states, updating them as actions are executed and new information becomes available.

A more general form of conformant planning is contingent planning where the agent can observe the world as it acts, and it can choose different continuations of the plan depending on what it observes. This allows the planner to reduce uncertainty during execution and adapt its behavior to the actual state of the environment. Peot and Smith (1992) extended classical planners with conditional actions (i.e., actions with several possible outcomes, only one of which occurs when the action is executed) to support contingent planning. Given a goal and a set of ordinary and conditional actions, the planner builds a plan that branches at every conditional action, constructing a separate partial plan below each possible outcome to carry the agent from that outcome to the goal. Others have enabled the planner to use explicit knowledge acquire actions (subgoals) to decide which plan branch to follow and use this information to

make informed decisions (Pryor and Collins, 1996)

An alternative approach is knowledge-level planning, which explicitly represents what the agent knows rather than enumerating all possible world states. In their work, Petrick and Bacchus (2002, 2004) represented the agent’s knowledge using first-order modal logic. The actions of the agent are defined in terms of how they modify the agent’s knowledge state instead of directly modifying the physical world. For example, sensing actions observe the state of the world and can increase the agent’s knowledge, while ordinary actions can make changes to the world. Sensing actions are treated as ordinary planning actions whose primary effect is to increase the agent’s knowledge rather than change the physical world.

Researchers have extended these ideas to epistemic planning, where planning is performed over explicit representations of knowledge and belief, specially under multi-agent settings. Drawing on Dynamic Epistemic Logic (DEL, Section 2.1.2) they represent not only what an agent knows about the world, but also what it knows about other agents’ knowledge, including nested knowledge such as one agent knowing that another agent knows a particular fact. Thus, epistemic planning contain actions that aim to change the physical state of the environment as well as the epistemic state of one or more agents (Bolander and Andersen, 2011; Belle et al., 2023).

In a different line of work, researchers brought together the ideas from logic programming and planning systems together to develop a framework for encoding planning problems as logic programs with answer set semantics (Subrahmanian and Zaniolo, 1995; Dimopoulos et al., 19). Later Lifschitz (1999) introduced the term *Answer Set Planning* for this family of approaches. Given a planning problem consisting of a domain description, an initial state, and a goal, Answer Set Planning translates the problem into a logic program Π consisting of *domain-dependent* rules that describe the domain, initial state, and goal (typically compiled from an action language) and *domain-independent* rules that generate candidate action occurrences at each time step and enforce inertia (the assumption that a fluent’s truth value persists across time steps unless an action explicitly changes it, see Section 4.2 for more details). Computing answer sets of Π then amounts to computing plans, which are action sequences that transform an initial state of the world into a desired goal state (see Section 4.3 for more details). Since the underlying representation is a non-monotonic logic program, Answer Set Planning inherits ASP’s ability to represent defaults, exceptions, and incomplete information directly within the planning domain, providing a framework for reasoning under incomplete information.

2.4.3 Plan Recognition

Rather than computing a sequence of actions to achieve a goal, plan recognition seeks to infer an agent's underlying goal or plan from a sequence of its observed actions. Over the years, many techniques have been proposed to solving this task (Albrecht and Stone, 2018; Keren et al., 2020; Meneguzzi and Fraga Pereira, 2021).

Most plan recognition approaches rely on plan libraries: structured representations of possible plans, often organized hierarchically. Given such a plan library and a sequence of observed actions, the task is to identify one or more plan hypotheses that are consistent with the plan library and explain (i.e., contain) the observed actions (Kautz and Allen, 1986; Geib and Goldman, 2009; Mirsky et al., 2017). Although effective in well-defined domains, constructing complete plan libraries can be labor-intensive and the defined plan libraries can be incomplete.

Others have formulated plan recognition as a problem of probabilistic inference using Bayesian networks. Charniak and Goldman (1993) represented high-level plans and their decomposable actions as nodes in a Bayesian network, with prior probabilities set for the high-level plans. Observed actions were then treated as evidence, and standard Bayesian inference was used to update the probability of each candidate plan.

Rather than relying on a plan library, Ramírez and Geffner (2009) reformulated plan recognition as a planning problem assuming a planning domain model (represented using STRIPS or PDDL) and a set of candidate goals. The idea was that the potential goal of an observed agent is likely to be one for which the optimal plan contains the observed actions in the order they were executed. This formulation allowed existing classical planning techniques to be reused for goal and plan recognition, establishing a close connection between recognition and the planning paradigms discussed in Section 2.4. This work was later extended to compute Bayesian probabilities over candidate goals by specifying prior probability for each possible goal and defining the likelihood of the observed actions given a goal as the cost difference between the plan that optimally achieves the goal and the plan that optimally achieves the goal and is consistent with the observed actions (Ramírez and Geffner, 2010).

Later work has focused on improving the efficiency and scalability of planning-based recognition. For example, Vered et al. (2016) introduced the concept of goal mirroring, in which the recognizer estimates the observed agent's goal by comparing its behavior with the actions it would perform if it were pursuing each candidate goal itself. Building on this idea, Vered and Kaminka (2017) developed a heuris-

tic approach for online goal recognition that updated goal estimates incrementally as new observations become available, making them suitable for real-time recognitions. Later, Vered et al. (2018) combined goal mirroring with planning landmarks for goal recognition. Landmarks are properties (or actions) that cannot be avoided to achieve a goal thus must occur in every valid plan for a particular goal. Since landmarks are unavoidable, they provide informative evidence about an agent's intentions. Building on this idea, (Pereira et al., 2020) developed landmark-based heuristics and filtering techniques for goal recognition. By evaluating how well the observed actions satisfy the landmarks associated with each candidate goal, landmark based methods eliminate unlikely goals and reduce the computational effort required for goal recognition improving efficiency and accuracy.

Another line of work studied plan recognition by identifying similarity to past plans. Rather than relying on predefined plan libraries or planning models, these approaches maintain a repository of previously observed plans and retrieve those that are most similar to the current situation. For example, Kerkez and Cox (2002) proposed a case-based reasoning approach in which previously observed execution traces, consisting of sequences of world states and actions, are stored in a case base and matched against the current observations. Unlike plan-library approaches, these methods allow the repository of known behaviors to expand as new plans are observed, reducing the need to specify all possible plans in advance.

Although plan recognition provides powerful techniques for inferring the intentions of other agents, it is not the primary focus of this thesis. Instead of explicitly recognizing plans or goals, our work uses lightweight heuristic models to anticipate teammates' likely future behavior, enabling the ad hoc agent to coordinate effectively with its teammates.

2.4.4 Negotiation

While plan recognition allows an agent to infer others agent's goals from its observed behavior, negotiation allows multiple agents to coordinate by reaching agreements over task allocation, resource sharing, or commitments through explicit interaction. One of the earliest negotiation mechanisms is the Contract Net Protocol proposed by Smith (1980). It was developed as a high-level communication and control protocol for distributed problem solving, in which a manager agent announces a task, receives bids from potential contractor agents, and awards the task to the most suitable contractor.

This laid the foundation for many subsequent coordination and negotiation frameworks in multi-agent systems.

Jennings et al. (2001) surveyed the field of automated negotiation and discussed a number of approaches, including bargaining, auction- and bidding-based negotiation, and argumentation-based negotiation. Subsequent work developed game-theoretic models for negotiation between self-interested agents (Kraus, 2001) and argumentation-based approaches, where agents exchange reasons supporting their proposals rather than only offers and counter-offers (Rahwan et al., 2003).

Negotiation has also been incorporated into logic-based planning frameworks. Son and Sakama (2010a,b) introduced cooperative actions into action languages, allowing one agent to request assistance from another agent or establish conditions that enable coordinated behavior. These cooperative actions can be composed into joint plans for teams of collaborating agents. More recent work employed the action language $\mathcal{B}$ for distributed planning among non-cooperative or partially cooperative agents, using negotiation as the coordination mechanism for resolving conflicts and constructing compatible plans (Son and Balduccini, 2018).

While methods discussed above provide effective mechanisms for coordination, they generally assume that agents share compatible communication protocols and focus on communication strategies and agreement protocols which may not hold in ad hoc teamwork domains (see Section 1.2).

2.5 Ecological Rationality and Heuristics

Humans make decisions by applying logic, statistics, or heuristics, each of which is suited to solving a particular type of problem (Gigerenzer and Gaissmaier, 2011). Out of these heuristics are efficient cognitive processes, that conscious or unconscious simplify decision-making by ignoring part of the available information. By focusing only on a few key cues, heuristics enable faster, frugal, and sometimes more accurate decisions compared to more complex methods (Todd and Gigerenzer, 2012).

While it is assumed that more information and computation lead to better outcomes, humans and other animals have often relied on simple strategies or heuristics to solve adaptive problems. For example, when deciding whether it is safe to cross a busy road, where someone has no exact information about the speed, distance, and acceleration of approaching vehicles, humans, and animals rely on simple heuristics, such as avoiding crossing when an oncoming vehicle appears to grow rapidly in size,

or crossing when there is a visible gap. Even in complex situations, such as financial investment, where future conditions cannot be fully known, people use heuristics to guide decisions. While people often rely on heuristics, under certain conditions they would achieve better accuracy by following optimization strategies. Specially, when all information is available, including perfect foresight of future states and outcomes. Such worlds are referred to as small worlds (Gigerenzer and Gaissmaier, 2011). In a small world, the agent knows the exhaustive and mutually exclusive set of all possible future states and consequences associated with each action given a state.

In contrast, in many real world situations these assumptions do not hold, since the future is not fully knowable and uncertainty cannot be meaningfully reduced to probability (Katsikopoulos et al., 2021). These are referred to as large worlds. In a large world, part of the relevant information may be unknown or need to be derived from limited samples. Unlike small worlds with perfect foresight, large worlds lack a complete specification of future states and their consequences. As a result, standard expected utility maximization cannot be applied. In such uncertain environments, optimization may no longer guarantee the best outcomes or even be impossible to compute, since all relevant information is unavailable or change over time. Thus, applying such models can lead to failures, as shown by Gigerenzer and Gaissmaier (2011). Of course, such uncertainty does not apply to all real world situations. However, in the AHT problem addressed in this thesis, agents must operate under incomplete information and the conditions of a large world.

Simon et al. (1989) introduced the concept of bounded rationality to study how humans reason under such conditions of uncertainty. He argued that people behave as if they maximize their expected utility, not that they actually do. Instead, they make decisions under uncertainty by relying on aspiration levels, limited search and heuristics such as satisficing. Under satisficing, individuals aim to find a solution that is “good enough” when the optimal solution cannot be known. This involves setting an aspiration level and choosing the first option that meets or exceeds it, allowing decision-making in situations where not all options are knowable.

Research in ML also address decision-making under uncertainty, with models such as neural networks and random forests. However, there are several limitations with such approaches. Firstly, these models are complex and typically not transparent. With the increasing use of AI agents in different applications, studies show that lack of transparency could limit trust and acceptability of autonomous systems (Anjomshoe et al., 2019; Fox et al., 2017). In contrast, models based on heuristics are inherently

transparent. Secondly, while complex algorithms can outperform heuristics when large amounts of data are available, with small samples, when the future may differ unpredictably from past experience, simple heuristics have shown to outperform standard statistical methods across diverse practical domains, including finance, sports, navigation, health, and animal behavior while using same or even more information (Katsikopoulos et al., 2021; Todd and Gigerenzer, 2012; Gigerenzer, 2020).

No single heuristic or optimization strategy consistently outperforms others across all environments (Katsikopoulos et al., 2021). Instead, the key question is to identify the conditions under which heuristics outperform complex strategies, and vice versa. Simon (1990) used the scissors analogy to explain that human rational behavior is shaped by a pair of scissors, whose two blades are the structure of task environments and the computational capabilities of the actor. He argued that adaptive cognition requires both blades to fit together, and under uncertainty humans rely on a variety of cognitive tools called the “adaptive toolbox”, rather than a single universal mechanism (Todd and Gigerenzer, 2012). This adaptive toolbox contains a repertoire of heuristics that are adapted to specific classes of problems. Gigerenzer et al. (1999) later introduced the idea of Ecological Rationality (ER) to study which heuristics succeed in which environments, and how heuristics and environments co-evolve to shape one another.

Many heuristics in this toolbox have been studied by researches including the recognition heuristic, fluency heuristic, take-the-best, fast-and-frugal trees, tallying, and the 1/N rule (Todd and Gigerenzer, 2012; Gigerenzer and Gaissmaier, 2011). In this thesis, our focus will be on fast-and-frugal trees.

2.5.1 Fast and Frugal Trees

Martignon et al. (2008) introduced Fast and Frugal (FF) trees as a heuristic designed for real world classification tasks under constraints of limited time, information, and computation. A FF tree is a binary decision tree in which each internal node corresponds to a binary cue. The cues are evaluated sequentially, and every level of the tree contains at least one decision exit. Decision making stops as soon as an exit is reached, in other words only a subset of available cues may be examined for a given case leading to a faster decision. The structure of the tree specifies both the order in which cues are considered and the conditions under which a classification is made.

For example, Figure 2.1 shows an FF tree developed by Green and Mehr (1997) to

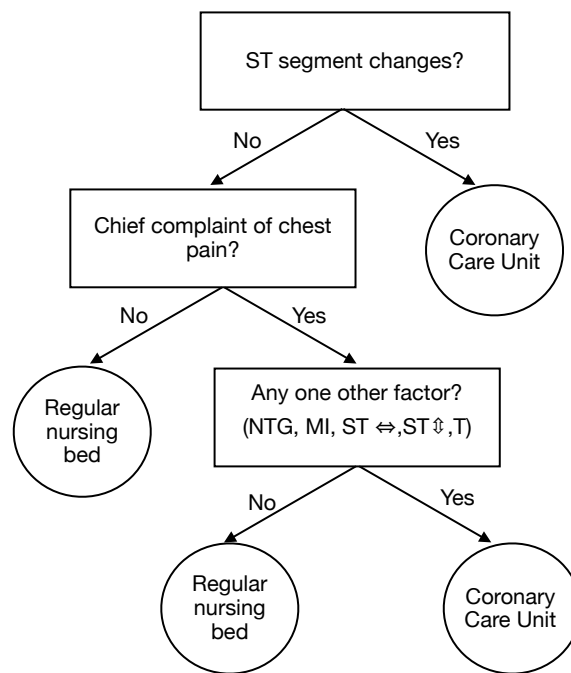


Figure 2.1: Fast and Frugal tree developed by Green and Mehr (1997) to help physicians in a Michigan hospital to decide whether a patient with severe chest pain should be admitted to an intensive coronary care unit (ICU) or to a regular ward.

help physicians in a Michigan hospital to quickly decide whether a patient with severe chest pain should be admitted to an intensive coronary care unit or to a regular ward. The tree uses at most three cues, but often reaches a decision after fewer cues, as each node in the tree has an exit that can trigger an early decision. When applying the tree, the physician considers the first cue (the root node), “is the ST segment depressed/elevated?”. If the answer to the cue is yes, the tree immediately classifies the patient to the admission of intensive coronary care unit and ignores all remaining information. If not, the decision process moves to the next node, and so on. This sequential structure imposes a priority ordering on cues: each is considered one at a time, and decisions are often made based on only a subset of all available information. Once a decision is made based on some information, no additional information can change the decision, thus as soon as a cue provides a clear decision, the process stops. Put simply, FF trees rely on a few cues that are “good enough” to make decisions quickly and efficiently, thereby embodying both frugality and speed. This contrasts with algorithms such as regression and random forests, which systematically weigh and integrate most or all

cues, allowing the influence of one cue to overturn the evidence of others.

As noted by Phillips et al. (2017), FF trees have three main advantages. First, they rely on much less information making them well suited for decision tasks where information is scarce or costly to obtain in terms of time and computation. Secondly, they explicitly ignore cue interactions, reducing the chance of fitting noise in the training data. Thus, they are robust against overfitting and capable of making reliable predictions with small or noisy datasets. Previous studies have shown that FF trees can achieve predictive accuracies comparable to complex models such as logistic regression and are thus used by experts in many fields (Martignon et al., 2008; Woike et al., 2017; Keller and Katsikopoulos, 2016; Jenny et al., 2013). Lastly, FF trees are simple and transparent. They are easy to understand by both designers and end users. This transparency makes them useful in domains where transparency in decisions is much critical.

2.5.2 Constructing Fast and Frugal Trees

FF trees are supervised learning algorithms that use only binary cues. A tree consists of nodes that correspond to cue-based questions, branches that represent the answers to those questions, and leaves that represent decisions. An FF tree with k number of cues has $k+1$ leaves (exits). Each level of the tree contains an exit, so any single cue can lead directly to a decision. All nodes except the last have a single exit, while the final node has two. The order of cues in the tree determines the order in which they are considered during decision-making. Cues are evaluated one at a time, and the process stops as soon as a cue leads to a clear decision. As a result, not all cues are necessarily examined for every decision instance. Only enough cues are processed to reach a decision.

Building an FF tree involves four main steps: (1) selecting the cues to be used; (2) determining a decision threshold for each cue; (3) determining the order of cues; and (4) determining the exit for each cue. Different algorithms have been proposed for constructing FF trees based on these steps. For example, (Martignon et al., 2003, 2008; Woike et al., 2017) proposed two construction algorithms, max and zigzag, which used the median value of numeric cues to determine the decision thresholds. The max algorithm ranked cues according to the maximum of their two validities: positive validity and negative validity. Cues with higher positive than negative validity are assigned positive exits, and cues with higher negative than positive validity are assigned neg-

ative exits. The zigzag algorithm on the other hand determined the exit direction for each node before establishing the cue order. It alternated decision exits between positive (yes) and negative (no) at each cue level, and selected the next cue based on the greatest positive validity among remaining cues for a “yes” exit or the greatest negative validity among remaining cues for a “no” exit, alternating as needed. Both algorithms have shown to achieve predictive accuracies comparable to logistic regression and standard decision trees (Martignon et al., 2008). Later, Phillips et al. (2017) proposed two new algorithms for constructing FF trees, IFAN and DFAN. In this thesis, we focus on the IFAN algorithm, which is described in the next section.

2.5.3 IFAN Algorithm for Fast and Frugal Trees

When constructing an FFtree, the IFAN algorithm (Algorithm 1) first calculates the decision threshold and decision direction for each cue. For numeric cues, the threshold value is determined by selecting twenty equally spaced values between the minimum and maximum values of the cue (Line 4). This number can be changed by the user. Each selected threshold is then paired with the two possible decision directions ($<$ and $>$) and tested on the dataset to select the threshold–direction pair that maximizes a chosen performance criterion, when the decision is made using that cue alone, ignoring all other cues (Line 6). By default, the performance criterion is balanced accuracy ($bacc$) which is defined by,

$$bacc = \left(\frac{hi}{hi + mi} \right) * 0.5 + \left(\frac{cr}{cr + fa} \right) * 0.5 \quad (2.11)$$

hi (hits) represent the positive instance correctly classified as positive (true positive), cr (correct rejections) represent the negative instance correctly classifies as negative (true negative), mi (misses) represent the positive instance incorrectly classified as negative (false negative); and fa (false alarms) represent the negative instance incorrectly classified as positive (false positive). Thus, $bacc$ represent the average of sensitivity ($\frac{hi}{hi+mi}$) and specificity ($\frac{cr}{cr+fa}$).

Next, IFAN ranks the cues by their maximum performance criterion values (Line 10). From this ranking, it selects the top max_levels cues (max_levels is user specified and the default value is 4) and discards the rest (Line 11). Using these selected cues, the algorithm then generates $2^{max_levels-1}$ candidate trees (Line 12). These trees are created keeping the cue ranking from the previous step and using all possible exit structures (each node from the root through level $max_levels - 1$ can have either a positive or a

negative exit and the final *max_levels* node have both).

Algorithm 1: IFAN Algorithm

Input: *data*: training data set for learning the tree, *C*: cues provided by the designer to be considered when building the tree, *K*: number of thresholds per cue, *max_levels*: maximum number of cues per tree

Output: Tree

```

1  $O \leftarrow \emptyset$ 
2  $\mathcal{B} \leftarrow \emptyset$ 
3 for  $c \in C$  do
4    $\mathcal{T}_c \leftarrow \{t_k | t_k \text{ equally spaced} \in [\min_c(data), \max_c(data)], k = 1, \dots, K\}$ 
5    $\mathcal{D} \leftarrow \{>, <\}$ 
6    $t_c, d_c, bacc_c \leftarrow \arg \max_{t \in \mathcal{T}_c, d \in \mathcal{D}} bacc(c_{t,d}(data))$ 
7    $O \leftarrow O \cup (t_c, d_c)$ 
8    $\mathcal{B} \leftarrow \mathcal{B} \cup bacc_c$ 
9 end
10  $C_{sorted} \leftarrow \text{sort } C \text{ using } \mathcal{B}$ 
11  $C_{top} \leftarrow \{c_1, \dots, c_{max\_levels}\} \subseteq C_{sorted}$ 
12  $Tr_{cand} \leftarrow \{t_i | t_i \text{ constructed from all exit configurations}(data, O, C_{top})\}$ 
13  $Tr \leftarrow \{t_i | t_i \text{ prune}(t_i), t_i \in Tr_{cand}\}$ 
14  $Tree \leftarrow \arg \max_{t \in Tr} wacc(t(data))$ 
15 return Tree

```

Each candidate tree is then used to classify all instances in the dataset. If a lower level node contains fewer instances than a user defined stopping parameter expressed as a percentage of total cases (10% by default), that node is removed and its exit structure is moved upward (Line 13). If the removed node was a leaf, the final lowest remaining node is forced to contain both the positive and negative exits.

Finally, for each tree, the weighted accuracy (*wacc*) is calculated with a user defined sensitivity weight (*w*, between 0 and 1). The parameter *w* determines the relative importance of sensitivity compared to specificity. For instance, when sensitivity is prioritized over specificity, *w* is set to be greater than 0.5, whereas a value of 0.5 assigns equal importance to both. *wacc* is defined by,

$$wacc = \left(\frac{hi}{hi + mi} \right) * w + \left(\frac{cr}{cr + fa} \right) * (1 - w) \quad (2.12)$$

The tree with the highest *wacc* value is selected as the final FF tree (Line 14).

The DFAN algorithm (Algorithm 2) follows a similar process to IFAN, but differs in how cue thresholds are determined: instead of computing thresholds once on the full dataset and ranking them, DFAN iteratively recalculates cue thresholds and accuracies based on the subsets of cases that emerge dynamically as the tree is constructed.

Algorithm 2: DFAN Algorithm

Input: *data*: training data set for learning the tree, *C*: cues provided by the designer to be considered when building the tree, *K*: number of thresholds per cue, *max_levels*: maximum number of cues per tree

Output: Tree

```

1   $O \leftarrow \emptyset$ 
2   $C_{order} \leftarrow \emptyset$ 
3   $data_{cur} \leftarrow data$ 
4   $C_{rem} \leftarrow C$ 
5  for  $level \in \{1 \dots max\_levels\}$  do
6       $O_{level} \leftarrow \emptyset$ 
7       $\mathcal{B}_{level} \leftarrow \emptyset$ 
8      for  $c \in C_{rem}$  do
9           $\mathcal{T}_c \leftarrow \{t_k | t_k \text{ equally spaced}$ 
             $\in [\min_c(data_{cur}), \max_c(data_{cur})], k = 1, \dots, K\}$ 
10          $\mathcal{D} \leftarrow \{>, <\}$ 
11          $t_c, d_c, bacc_c \leftarrow \arg \max_{t \in \mathcal{T}_c, d \in \mathcal{D}} bacc(c_{t,d}(data_{cur}))$ 
12          $O_{level} \leftarrow O_{level} \cup (t_c, d_c)$ 
13          $\mathcal{B}_{level} \leftarrow \mathcal{B}_{level} \cup bacc_c$ 
14     end
15      $c_{best} \leftarrow \arg \max_{c \in C_{rem}} \mathcal{B}_{level}$ 
16      $C_{order} \leftarrow C_{order} \cup c_{best}$ 
17      $O \leftarrow O \cup O_{level}[c_{best}]$ 
18      $C_{rem} \leftarrow C_{rem} \setminus \{c_{best}\}$ 
19      $data_{cur} \leftarrow \{d \in data_{cur} \mid d \text{ not classified by } (c_{best}, O_{level}[c_{best}])\}$ 
20 end
21  $Tr_{cand} \leftarrow \{t_i | t_i \text{ constructed from all exit configurations}(data, O, C_{order})\}$ 
22  $Tr \leftarrow \{t_i | t_i \text{ prune}(t_i), t_i \in Tr_{cand}\}$ 
23  $Tree \leftarrow \arg \max_{t \in Tr} wacc(t(data))$ 
24 return Tree

```

Studies in the literature have shown that FF trees constructed with IFAN are faster

and more accurate than earlier FF tree algorithms such as max and zigzag. They are also shown to achieve prediction performance comparable to state of the art machine learning models including support vector machines, regularized logistic regression, and random forests (Phillips et al., 2017).

Overall, FF trees classify instances quickly, perform well even with limited data, and remain easy to understand and explain due to their transparency. By design, they are suited for environments such as AHT settings, where faster, simple, and robust decisions are preferred over computationally expensive or opaque strategies. Motivated by these findings, we utilize FF trees to model the behavior of the other agents incrementally and from limited data in AHT domains.

2.6 Large Language Models

LLMs estimates the probability of a token or a sequence of tokens occurring in a given context (i.e., within a longer sequence of tokens). A token is the smallest unit of language modeling and may correspond to a single character, a word, or a subset of a word. By assigning probabilities to sequences, LLMs can predict the next token in a sequence and extend it to form sentences, paragraphs or even longer documents. As a result, they can appear to perform complex tasks, such as solving mathematical problems or planning, although their behavior is based on sequence prediction rather than explicit reasoning.

Most modern LLMs are based on Transformer architectures (Vaswani et al., 2017). Transformers use self-attention mechanisms, which allow the model to determine how each token in a sequence relates to others. This enables the model to capture dependencies across the entire input. Transformers contain billions of parameters (e.g., the GPT-3 model has 175 billion parameters (Brown et al., 2020)), and training them requires large amounts of data and substantial computational resources (Kaplan et al., 2020). Once trained, they can perform inference using relatively fewer resources, although the model size and complexity continue to impose significant computational costs. Another major issue with LLMs is that they can hallucinate (i.e., generate fluent but incorrect or misleading outputs) and reproduce biases present in the training data (Huang et al., 2025; Bender et al., 2021).

Despite these limitations, LLMs are effective at generating easy to understand text for a variety of target audiences, and making predictions on tasks that they are explicitly trained on. While some researchers claim that LLMs can be used for making

predictions for tasks that they were not explicitly trained on (i.e., emergent abilities), this is still unclear and under ongoing debate (Brown et al., 2020; Lu et al., 2024).

While an LLM that is trained on sufficient data may capture general properties of language, including grammar and common usage, and exhibit considerable performance in those tasks, such models are not directly optimized for specific applications. Thus, they need to be *fine tuned* to adapt to a particular task (Radford and Narasimhan, 2018; Devlin et al., 2018). In fine tuning, the LLM is updated with task-specific data to improve its performance on that particular task. In standard fine tuning all model parameters are updated. Thus, it is computationally expensive to perform for large models. To reduce this cost, parameter-efficient methods have been proposed, which update only a subset of parameters while keeping the rest fixed (Hu et al., 2022; Houlsby et al., 2019; Lester et al., 2021; Li and Liang, 2021). These approaches enable efficient adaptation of LLMs to specific tasks while maintaining performance.

Another approach to reduce computational costs is distillation, where a smaller model is trained to reproduce the behavior of a large model. This is acquired by either incorporating additional loss functions for aligning the outputs or intermediate states of the smaller model to those of the larger model (Gu et al., 2026), or using the responses from the larger model to train the smaller model (Hsieh et al., 2023). The resulting model can generate predictions faster and with fewer resources, but may achieve lower accuracy compared to the original model.

Prompt engineering is another method used to control and adapt the LLM output without modifying model parameters (White et al., 2023). By structuring the input prompt and providing examples, users can guide the LLM's behavior. This includes zero-shot prompting (no examples), one-shot prompting (a single example), and few-shot prompting (multiple examples). These methods rely on the model's ability to use patterns in the input to generate appropriate outputs.

Overall, LLMs are used effectively across a range of tasks but require significant resources and careful adaptation for specific applications. There is increasing evidence to show that they can make arbitrary decisions in novel situations (Lu et al., 2024), do not actually plan (i.e., struggle to solve problems that require multi-step plans), and are more effective when used to generate abstract guidance that is validated before being used (Guan et al., 2023; Kambhampati et al., 2024).

Motivated by these findings, we leverage a pretrained LLM to anticipate high-level future tasks in the domain to be completed by the ad hoc agent and its team, automatically prompting and adapting the LLM's output based on domain-specific knowledge

using a combination of prompt engineering techniques and a simple validator. We also use a pretrained LLM to process natural language descriptions of actions and outcomes, enabling the ad hoc agent to incrementally acquire and revise its knowledge in the form of objects, actions, and axioms governing domain dynamics.

2.7 Chapter Summary

This chapter provided the background required to understand the approach proposed in this thesis. We began by discussing logical reasoning and its extensions for representing and reasoning with incomplete knowledge: non-monotonic reasoning and modal logic. We then reviewed declarative programming languages, ASP, and action languages, which together provide the theoretical and computational foundation for the architecture proposed in this thesis. To position our approach within the broader planning literature, we reviewed classical planning and planning under incomplete information, as well as related multi-agent reasoning techniques, including plan recognition and negotiation. We then introduced ecological rationality and heuristic decision-making, and described fast-and-frugal trees as a method that performs well under limited data and is transparent by design. Finally, we reviewed Large Language Models (LLMs), whose capabilities we leverage to interpret natural language and anticipate future tasks. Together, these concepts provide the foundations for the approach presented in the following chapters.

Our architecture builds on these foundations by incorporating commonsense domain knowledge through non-monotonic logical reasoning, develop predictive models of other agents from limited examples using ecological rationality and heuristic methods, and supports incremental acquisition and revision of domain knowledge. In addition, the architecture emphasizes transparency by providing on-demand explanations of the agent’s own decisions and beliefs, as well as those of other agents.

Chapter 3

Related Work

This chapter positions this thesis within the broader context of existing research literature. We begin by discussing various approaches and strategies that have been developed to address the AHT problem, categorizing them based on their knowledge representation, teammate modeling and adaptation, and action selection mechanisms (Section 3.1). We discuss the key challenges that remain unsolved and identify gaps in the existing AHT research, establishing the foundation for the novel architecture proposed in this thesis. We also review current advancements in explainable AI and knowledge acquisition in AI systems (Section 3.2 and Section 3.3).

3.1 Ad Hoc Teamwork

AHT has received considerable attention in recent years, as discussed in a recent survey (Mirsky et al., 2022). Early work by Bowling and McCracken (2005) introducing the concept of a ‘pickup player’, an independent agent joining an already established team without prior coordination. In their approach, teams relied on predefined ‘plays’, which specified a set of roles guiding each agent’s behavior in various situations. The pickup player inferred the play that is being used by the team and its own role within that play, based on state observations (e.g., player positions and trajectories), allowing it to align its actions with the team’s behavior.

This challenge of enabling an agent to collaborate efficiently with previously unknown agents without prior coordination was later formalized as the *ad hoc teamwork* problem by Stone et al. (2010). There are three key assumptions that characterize the AHT problem:

- No prior coordination: The ad hoc agent is expected to collaborate with other

agents it has not previously worked with, and without established prior coordination mechanisms or communication protocols. While the ad hoc agent may have partial knowledge of similar types of teammates, it is not assumed to have prior experience with the specific agents it is required to work with.

- **Collaboration:** The agents have a shared goal they seek to achieve together through collaboration. Each individual agent may have additional objectives, but these do not conflict with the primary goal of the team.
- **No control over teammates:** The ad hoc agent does not have control over the other agents in the domain. It may influence the teammates' behavior through its action choices, but it cannot directly change the properties of their policies or behavior.

Given above assumptions, a large body of work has been carried out addressing different aspects of the AHT problem. Below, we group this literature into three main subtasks: mechanisms used for representing agent's knowledge (Section 3.1.1), strategies adopted by ad hoc agents for selecting their own actions based on this knowledge (Section 3.1.2), and mechanisms for adaptation (Section 3.1.3).

3.1.1 Knowledge Representation

In AHT, an architecture must incorporate knowledge about the domain and other agents to account for uncertainty caused by domain dynamics and unknown behaviors during interaction, thus enabling effective collaboration. State of the art solution for AHT generally do not integrate explicit domain knowledge as a core architectural component. Instead, these approaches focus on adaptability through data-driven learning, relying heavily on extensive prior observations to learn domain dynamics (Barrett et al., 2017; Chen et al., 2020; Rahman et al., 2021; Zintgraf et al., 2021). Rather domain characteristics such as whether an environment is discrete or continuous, or static or dynamic due to other agents' actions, are typically used by system designers to determine the most appropriate approach or algorithm to apply to solve the AHT problem. For example, a designer's understanding that an AHT environment is discrete may influence the choice of a specific action selection mechanism for the ad hoc agent.

When providing knowledge regarding potential teammates, the most common sources used are input from a human expert or the agent's own prior experience. Existing approaches generally represent this knowledge in two ways. (1) Type-based represen-

tation in which prior observations are used to construct explicit models of teammate behavior, with each type corresponding to a particular action-selection policy or behavioral pattern; (2) *Experience-based* representation, in which prior interactions are stored directly and later retrieved to guide decision making without constructing an explicit behavioral model.

Type-based methods: Type-based approaches in the state of the art of AHT use a long history of prior interactions to learn explicit representations of teammate behavior. These representations may take the form of probabilistic models, reinforcement learning policies, or deep neural network models. During interaction, the ad hoc agent tries to identify which learned type best matches the observed behavior of its teammates and uses the corresponding model to predict their future actions. For example, the PLASTIC algorithm used decision-tree models learned offline for each teammate type to probabilistically determine the ad hoc agent’s actions (Barrett et al., 2011, 2017). Others have provided the ad hoc agent with a set of user-defined types that represent different agents that it may encounter during interaction (Albrecht and Ramamoorthy, 2013). Recent methods have utilized neural network based encoders to map observations to an embedding of teammate types. For example, sequential and hierarchical variational autoencoders have been used to model changing behaviors of teammates by separating consistent behavior from short term changes (Zintgraf et al., 2021). Each teammate type is modeled to have two characters, an agent character which does not change throughout the interaction (consistent behavior) and a mental character that can change during interaction. Potential changes in team composition have also been handled by training Graph Neural Networks to consider a range of team sizes and teammate types, modeling teammates as nodes (in a graph) that can be added or removed, and using Long Short-term Memory networks for type inference (Rahman et al., 2021). More recent work has used a self-play strategy to learn a finite set of candidate teammate policies (Zand et al., 2022; Fang et al., 2024). Model-based Reinforcement Learning (RL) approaches have also been used to separately learn models of the environment and teammate behavior, where interaction data (observed states and teammate actions) were used to train a set of feed-forward neural networks as a set of teammate behavior models and transition probabilities and a reward function (Ribeiro et al., 2023; Xu et al., 2025).

Experience-based methods: In contrast to type-based methods, experience-based approaches do not construct explicit behavior models. Instead, they store interaction experiences and use similarity measures to retrieve relevant past observations when

encountering new teammates. For example, attention-based recurrent neural networks have been used to identify temporal correlations between current and previously observed states, enabling the ad hoc agent to retrieve relevant experiences and adapt its actions accordingly (Chen et al., 2020). Other work has employed a combination of KD-trees (k-dimensional trees) and arrays to store interaction experiences (Santos et al., 2021). A KD-tree is a data structure in which each node is a k-dimensional point. At each level of the tree, each node splits the space along one dimension at a time, cycling through the dimensions as the tree grows. Each nodes left/right children contain points that fall below/above the split value in that dimension. Santos et al. (2021) allowed the agent to learn a separate KD-tree for each team, with nodes storing state representations and transition id, and associated arrays storing the resulting successor states. During interaction, the agent found and retrieved the past experience state most similar to its current observation using the k-nearest neighbors algorithm and used that to guide its decision making.

While the type-based and experience-based methods summarized above offer promising strategies for modeling the behavior of teammates, they are often resource hungry, requiring substantial time, computation, and training examples to achieve the observed performance. These resources are not readily available in many practical AHT domains, posing questions about the scalability and usability of these methods. Also, these methods struggle to leverage the commonsense knowledge available in many practical application domains of AHT.

3.1.2 Action Selection

Based on the representation of knowledge (Section 3.1.1), the ad hoc agent selects its own actions to coordinate with its teammates. The action selection mechanisms used by the state of the art AHT methods can be broadly categorized into two groups: (1) Planning approaches, where the ad hoc agent's actions were determined through planning algorithms, and (2) Expert policy methods, where the ad hoc agent's actions were chosen using policies learned from experience data.

Planning methods: Many AHT architectures in the literature have used sampling-based methods, such as Upper Confidence bounds for Trees (UCT) for Monte Carlo Tree Search, for determining the actions of the ad hoc agent (Barrett et al., 2011, 2013; Albrecht and Stone, 2017; Ribeiro et al., 2023). Later work have extended UCT by incorporating history information into nodes, thereby reducing the number of distinct

states represented in the search tree enabling the ad hoc agent to make faster decisions (Yourdshahi et al., 2018). UCT has also been combined with biased adaptive play to form an online planning algorithm for AHT, with UCT estimating the joint utility function in each stage and biased adaptive play selecting the ad hoc agent’s action based on that estimate (Wu et al., 2011). Other work has used the Bellman optimality equation to compute the ad hoc agent’s actions by simulating interaction trajectories for known types of agents in the team (Albrecht and Ramamoorthy, 2013). These methods represent the teammate’s behavior using explicit behavior models (type-based) and use them to predict their future actions during interaction. With LLMs and other foundation models increasingly being considered state of the art for different problems in AI and robotics, recent work has leveraged LLMs for ad hoc agent’s action selection. For example, an LLM based hierarchical planner has been used to generate a sequence of sub-skills (*e.g.*, GoToRoom, PickUpObject) to be executed by the ad hoc agent. The ad hoc agent then used pre-trained execution policies obtained in advance to execute each sub skill (Liu et al., 2024).

Expert policy methods: Over time, AHT has primarily been posed as a learning problem, and many approaches have relied on expert policies to select the actions of the ad hoc agent. These policies are often obtained prior to interaction, using offline experience data or online RL training. For example, the PLASTIC algorithm used Fitted Q-Iteration to learn a set of policies for selecting the actions of the ad hoc agent using offline experience data (*i.e.*, states, actions, rewards, and next states) (Barrett and Stone, 2015; Barrett et al., 2017). A separate policy is learned for each teammate type, and during execution, the appropriate policy is selected based on the observed teammate type. Later work used attention-based recurrent neural networks, with one attention network learned for each teammate type. By measuring the similarity between the past and current teammates, they allowed the ad hoc agent to adapt its actions based on its estimate of the type of agents it is interacting with (Chen et al., 2020). The PLASTIC-Policy under Adversarial Selection (PPAS) framework used a Deep Q-Network to learn a policy for different teams, and combined the output from the learned policies using a greedy weighted majority rule to determine the ad hoc agent’s actions with non-stationary teammates (Santos et al., 2021). Model-based RL methods have also been used to determine the actions of the ad hoc agent (Ribeiro et al., 2023; Xu et al., 2025). These approaches rely on *experience-based* method to train the ad hoc agent’s policy.

It is often difficult to learn policies that capture every possible scenario the ad hoc

agent may encounter. On the other hand, type-based planning methods adapt to unseen team compositions by reasoning at the level of types of individual agent types. Considering this, some architectures have used a combination of type based methods and expert policy methods. For example, the learned models of different types of agents have been used to anticipate the effects of other agents' actions on the ad hoc agent's rewards in order to compute the policy governing the ad hoc agent's actions (Albrecht et al., 2015). Others have conditioned the ad hoc agent's policy, learned using proximal policy optimization, on its belief about teammates using sequential and hierarchical Variational Autoencoders (Zintgraf et al., 2021). Later work have employed self-play to learn a cooperation policy for the ad hoc agent alongside a set of candidate teammate policies (Zand et al., 2022; Fang et al., 2024).

While recent approaches for action selection in AHT are increasingly sophisticated, ranging from type based planning methods to deep neural network based policies, they often rely on larger training datasets and long learning periods to achieve generalizable performance. Moreover, their internal mechanisms governing the decision made are often opaque, limiting the ability to understand, justify and audit the automated decisions made by the agents.

3.1.3 Adaptation to Current Teammates

The components discussed above form the foundation of an ad hoc agent architecture. In other words, once the ad hoc agent has acquired initial knowledge about its domain/potential teammates and established a method for selecting its own actions, it has the core components necessary for AHT. However, since ad hoc agents need to collaborate with its teammates whose behaviors may initially be unknown, they need the ability to adapt their behavior during interactions. To support this, architectures in literature incorporate different adaptation mechanisms.

One of the most commonly used methods for such adaptation is that the ad hoc agent updates its beliefs about the other agent's types based on observations during interaction. For example, Bayesian inference has been used to update beliefs on the likelihood of a teammate belonging to a particular type based on the interaction history (Barrett et al., 2011; Albrecht and Ramamoorthy, 2013; Albrecht and Stone, 2017). Others have used Gibbs sampling to support scalable belief updates over teammate types (Zand et al., 2022). Similarly, Barrett et al. (2017) updated the ad hoc agent's beliefs over which policy to use for coordinating with each team using poly-

nomial weights algorithm from regret minimization. They measured the similarity between current and previously stored transition data to find the team that gives the least difference between expected and observed states. Later work extended this approach to train one attention neural network per teammate type, and extracted the temporal correlations between past and current sequence of states to produce a metric that measured the similarity between past and new teammates based on a trajectory rather than a single transition (Chen et al., 2020). Other work which stored the team models as KD trees used K-Nearest Neighbors over KD tree to compute the euclidean distance between the observed and the expected state from stored data, thus computing the similarity between the teams (Santos et al., 2021). To detect changing behaviors of teammates, researchers have used convolutional neural networks where change point detection is posed as an image classification task and agents shifting to a new behavior type is interpreted as a break in the image representation, thus identifying the change point (Ravula et al., 2019). Others have used hierarchical variational autoencoders to model teammates, and perform posterior inference over latent teammates representations using Gaussian approximations (Zintgraf et al., 2021).

When the existing type space lacks a suitable model for current teammates, researches have explored learning new models on the fly. For example, Barrett et al. (2011) used state-action observations from a single episode to build new models of current teammates during interaction. TwoStageTransfer algorithm has been used to learn a new model by integrating previous teammates models with limited observations of new teammates. TwoStageTransfer determine which teammate type model from the existing models is the most similar to the new teammate’s behavior based on the likelihood that both would choose the same action in the same state, and transfer knowledge from that model to the new one (Barrett et al., 2017).

Some researchers have investigated the use of communication mechanisms for adaptation in AHT settings. For example, communication messages has been used to broadcast observation and suggestions with teammates (Barrett et al., 2014). Sending these messages incurred a cost, and the team goal was to maximize the total payoffs minus communication costs. Other work has used communication to query a teammate about its hidden goal. By assigning values to queries, the agent decided when and what to communicate to minimize the total execution cost, identifying when querying is beneficial within a domain (Mirsky et al., 2020; Macke et al., 2021). Recent work has designed ad hoc agents to broadcast their state, capabilities, and intentions upon joining a team. The ad hoc agent then utilize information received from teammates’

messages with the help of an LLM to complete their planning (Liu et al., 2024). However, these approaches assume teammates possess communicative capabilities, which may not hold in all AHT settings.

While most AHT approaches assume tasks to be known beforehand and focus on uncertainty about teammates, some work have explored ambiguity in task identity. For example, Melo and Sardinha (2016) modeled a fully cooperative matrix game where the ad hoc agent must infer the current task from a set of possible tasks, through interactions observed with a teammate following a bounded rational best-response model. Others have modeled AHT under partial observability, where given a set of possible tasks described as a Partially Observable Markov Decision Process (POMDP) model the agent used Bayesian online prediction to map the history of observations to a belief over possible tasks and teammates (Ribeiro et al., 2022). Recent work has also used LLM based subtask planner to generate the next candidate subtask for the ad hoc agent using information obtained through communication messages with teammates. The ad hoc agent broadcasted the subtasks, and used the responses from teammates to assess the feasibility and select the most suitable task for execution (Liu et al., 2024).

These approaches provide different mechanisms for adapting to teammates during interaction. However, many rely on assumptions such as the availability of sufficient prior data, known task structures, or communication capabilities, which may not hold in all AHT settings.

3.2 Explanation AI

The increasing use of AI methods in safety-critical and high-impact applications has led to considerable interest in providing transparency in the operation of such methods (Miller, 2019; Rudin et al., 2022). This has led to the development of Explainable Artificial Intelligence (XAI), a field concerned with producing explanations that enable users to understand, trust, and interact with AI systems. Below we briefly discuss some work in XAI that is highly relevant to the approach proposed in this thesis and refer the reader to Barredo Arrieta et al. (2020) for a comprehensive survey.

Several taxonomies have been proposed to classify explanation methods. A common distinction is *interpretable models*, whose reasoning processes are directly understandable, and *post hoc explanation* methods, which generate explanations for otherwise opaque models after predictions have been made (Guidotti et al., 2018; Barredo Arrieta et al., 2020).

Many modern machine learning models, particularly deep neural networks, fall into the second category, where they often achieve state-of-the-art predictive performance, but typically operate as black boxes, making it difficult for users to understand why a particular prediction or decision was made (Adadi and Berrada, 2018; Guidotti et al., 2018). However, the explanations produced by post hoc methods may not necessarily reflect the actual reasoning process that led to the model’s prediction (Rudin et al., 2022).

On the other hand, symbolic AI systems naturally support transparency since their reasoning processes are represented explicitly through logical rules and inference chains. For example, an early expert systems, MYCIN, could justify its recommendations by tracing the sequence of rules that led to a conclusion (Shortliffe et al., 1975; Clancey, 1983). Later, Halpern and Pearl (2005) proposed a formal framework for explaining why events occur based on causal models. Pearl and Mackenzie (2018) further discussed the importance of causal reasoning and how causal relationships can be used to explain the decisions made by intelligent systems. Others have considered the recipient’s prior knowledge for generating user-centric explanations (Ehsan et al., 2024).

Explainability has long been recognized as an important aspect of automated planning - Explainable AI Planning (XAIP). Unlike many machine learning systems, classical planners explicitly represent actions, goals, preconditions, and effects, allowing their decisions to be explained in terms of the planning model itself (Fox et al., 2017; Chakraborti et al., 2021). However, simply presenting a plan is not necessarily sufficient to explain the planner’s behavior, particularly when the user possesses an incomplete or inaccurate understanding of the planning model (Fox et al., 2017; Chakraborti et al., 2017). Thus, Fox et al. (2017) outlined a range of questions that planning systems should be able to answer, including contrastive questions concerning why one action or plan was selected over an alternative. This subsequently motivated the development of model-based explanation techniques in planning.

Early work focused on explaining why particular actions appeared in a plan and how they contributed to achieving the desired goal (Fox et al., 2017). Later work shifted towards model reconciliation, in which explanations are viewed as modifications that reconcile differences between the planner’s model and the user’s mental model. For example, Chakraborti et al. (2017) formalized explanations as the minimal set of model changes required to make the planner’s solution appear optimal from the user’s perspective. Rather than merely describing the generated plan, these explanations identify which assumptions held by the user must be revised to understand the

planner’s reasoning.

In a different line of work, researchers have designed architectures that embed the principle of *explainable agency* (Langley, 2019; Langley et al., 2017) and build on a theory for explanation generation (Sridharan and Meadows, 2019), enabling an agent to construct relational descriptions as explanations in response to different types of questions. Later work has adapted this theory to support transparency in the decisions made by agents in scene understanding tasks (Mota et al., 2021).

Others have explored how planners based on PDDL can generate explanations to justify its chosen plan is no worse than an alternative plan proposed by the user (Borgo et al., 2018). Symbolic knowledge representations have been combined with probabilistic reasoning to support planning and diagnostics in open worlds on a mobile robot platform, and generate explanations for task failures using diagnosed information (Hanheide et al., 2017). Subsequent work has used ASP for knowledge representation and for generating explanations for “why” type questions, explaining why a set of literals is, or is not, an answer set (Fandinno and Schulz, 2018). Explanation generation within ASP has been represented in multiple ways, including dependency graphs between literals and rules (Pontelli et al., 2009; Schulz and Toni, 2016; Cabalar et al., 2014; Trieu et al., 2021; Alviano et al., 2023), and propositional formulas (Viegas Damásio et al., 2013).

Motivated by these findings we seek to design architectures that incorporate interpretable models (Rudin, 2019) and automatically support process-level explanations based on knowledge that is evolving over time. As Goodman and Flaxman (2017) argue, automatically computed decisions will only be acceptable in the future if they are explainable in ways that humans can understand. The work discussed above demonstrates that beyond simply exposing the internal workings of an algorithm, explanations are intended to help users understand system behavior, justify individual decisions, detect errors, and calibrate their trust appropriately (Miller, 2019; Hoffman et al., 2018; Barredo Arrieta et al., 2020).

3.3 Knowledge Acquisition

The formalisms discussed in Section 2.1 often assume that the agent has some prior knowledge about the problem/domain (i.e., a knowledge base consisting of facts and rules) that it can use to reason over. Building this knowledge base by hand is labor-intensive, slow and requires a person who knows both the domain and the logical

formalism. Moreover, such knowledge bases can be incomplete and often needs to be revised over time. Thus, many methods have been developed to support knowledge acquisition.

Early work studied incremental revision of first-order logic representations of action operators when planning knowledge was incomplete (Gil, 1994). Later approaches combined non-monotonic logical reasoning with decision-tree induction and relational reinforcement learning to identify new rules for answer set programs (Sridharan et al., 2017). Others have leveraged a small number of real world trials to enable robots to learn symbolic representation of preconditions and effects of actions (Konidaris et al., 2018). Recent approaches have developed systems based on inductive learning to acquire new domain knowledge represented in the form of Answer Set Prolog programs (Law et al., 2018, 2020).

Researchers have also used standard natural language processing tools, such as part of speech tagging and WordNet synsets to process natural language descriptions and map them to actions and causal laws (Sridharan and Meadows, 2018). Natural language instructions have also been translated into sequences of transferable symbols that can be grounded in a novel map for mobile robot planning (Gopalan et al., 2020).

More recent approaches use LLMs to help produce this knowledge base. One line of work uses an LLM to translate natural language sentences into logic formulae (LLM-to-logic translation). For example, Lalwani et al. (2025) used LLM for converting natural language sentences into FOL formulae through a multi-step process involving few-shot prompting, decomposing sentence into smaller parts, identifying relationships between subcomponents, and identifying real-world relationships between these sub-components (background knowledge) and finally combine all this information to obtain the FOL formulae. Other work instead fine-tuned a 7-billion-parameter LLaMA model on a dataset of natural-language/FOL pairs specifically for this task (Yang et al., 2024). Others have leveraged the few-shot generalization ability of LLMs to provided the LLM with detailed instructions about the grammar of multiple symbolic languages (e.g., Prolog, FOL) and few demonstrations as in-context examples to identify key entities, facts, and rules present in the problem statement, and then translate these elements into symbolic language following the defined grammar (Pan et al., 2023).

Motivated by above work, we leverages a pretrained LLM to translate natural language descriptions of action outcomes and failures into formal Answer Set Prolog representations in terms of objects, actions and axioms, and use decision tree induction to incrementally revise knowledge based on observations.

3.4 Chapter Summary

This chapter reviewed existing literature on the AHT problem. We discussed different approaches based on how they represent knowledge, select actions of the ad hoc agent, and adapt during interaction. While these methods have achieved notable success, discussion above highlights several limitations of existing AHT approaches.

Firstly, many methods rely heavily on large amounts of prior interaction data to learn the models of teammates and policies for action selection, which may not be available in practice. Secondly, they do not explicitly incorporate structured commonsense domain knowledge, which could support more efficient reasoning and adaptation. Moreover, adapting to new teammates or changing behaviors often requires either extensive retraining or strong assumptions about the availability of prior models or communication. Thirdly, the decision-making processes of many existing methods, particularly those based on deep learning, are difficult to interpret. This limits the ability to understand, justify, and trust the behavior of the ad hoc agent.

These limitations motivate the need for AHT architectures that can incorporate commonsense domain knowledge, operate effectively with limited data, support efficient adaptation, and provide transparency in decision-making. Addressing these challenges is the focus of this thesis.

Chapter 4

Core Architecture

This chapter introduces our proposed solution to the AHT problem, KAT. KAT combines knowledge-based reasoning and data-driven learning to design and develop an ad hoc agent, which leverages the complementary strengths of each other to overcome their limitations. The chapter begins by providing an overview of the KAT architecture, highlighting the role of each key component before discussing them in detail in the following sections (Section 4.1). We describe the agent’s methodology for knowledge representation (Section 4.2), and reasoning (Section 4.3), followed by the learning component, which enables the agent to develop predictive behavior models of the other agents in the domain (Section 4.4). Next, we describe the procedure the agent uses to identify and adapt to different types of agents based on its existing knowledge (Section 4.5). We describe the system’s main control loop, providing pseudo code to demonstrate how the components are connected and how information is exchanged throughout the agent’s decision-making process (Section 4.6). Next, we detail the evaluation of KAT, beginning with the experimental setup (Section 4.7), followed by its performance across different tasks, domains, and agent configurations (Section 4.8).

4.1 General Overview

Figure 4.1 provides an overview of the KAT architecture presented in this thesis. KAT consists of five main components: *knowledge representation and reasoning*, *learned agent behavior models*, *task anticipation*, *knowledge acquisition* and *explanation generation*. The core architecture described in this chapter, focuses on the first two components: *knowledge representation and reasoning* and *learned agent behavior models* (represented as A in the Figure 4.1). The remaining components are discussed in the

subsequent chapters as: *task anticipation* (B in Figure 4.1) in Chapter 5, *knowledge acquisition* (C in Figure 4.1) in Chapter 6, and *explanation generation* (D in Figure 4.1) in Chapter 7. To illustrate these components, consider the following example scenario from the VirtualHome domain, which will be revisited throughout this thesis to indicate KAT's functionalities.

Example 6 [*Human-AI agent collaboration scenario from VirtualHome*]

Consider an AI (ad hoc) agent collaborating with a human and other AI agents to complete daily household tasks in a home environment. Examples of such tasks include *prepare breakfast*, *prepare coffee*, *set up the home work-station*, *pack bag* and *clean kitchen*. Figure 1.1 shows snapshots of these agents collaborating to perform such tasks in the VirtualHome domain. An external task generator generates the set of tasks to be completed on any given day and assigns these tasks to all agents one by one as each previous task is completed. Completing a task requires a sequence of actions to be computed and executed by members of the team without any direct communication between them¹. Each agent receives valid observations of the domain state. For example in the VirtualHome domain a state observation take the form of a graph, where nodes correspond to objects and edges represent relations between them. An object node for the kitchen table in the graph includes attributes such as:

$\{id : 129, category : Furniture, class_name : kitchentable, position : [-6.00999928, 0.029, 2.52]\}$

along with additional information such as rotation, size, and properties. Similarly, an object node for a cake is represented as:

$\{id : 175, category : Food, class_name : cake, position : [-5.70279932, 0.9955, 2.9469]\}$

Then the relationship between these objects is encoded as an edge of the form:

$\{from_id : 175, to_id : 129, relation_type : ON\}$

This state information is sent as input to all other agents, as well as the ad hoc agent's component A in Figure 4.1 together with the task assigned (e.g., task = *prepare breakfast*). The agents use this information to select and execute its actions to interact with the environment. Actions in the VirtualHome include moving to locations, picking up or placing objects, turning appliances on or off, and opening or closing them. For

¹Communication is an important means to share information. However, since communication is not readily available in many practical AHT domains, we do not include it in our example scenario.

example, to prepare breakfast, the team of agents will prepare and bring items such as cereal, milk, and toast to the table using actions such as *move*, *grab*, *switch on*, and *open*. The output of component A is the selected action of the ad hoc agent.

While each agent is aware of the current task assigned to the team by the task generator, they are not aware of the future tasks that will be assigned to them on that day. Thus, the ad hoc agent uses its task anticipation component (B in Figure 4.1) to anticipate the high-level future tasks that will be assigned to the team. The tasks completed so far on the given day is send as input to component B in Figure 4.1. The ad hoc agent uses this input together with historical data (task sequences from previous days) to construct a prompt to the LLM. The generated prompt is then sent to the LLM and the tasks anticipated by the LLM are verified using the external verifier before being returned as output back to the component A. This output (e.g., prepare coffee) is then considered by the ad hoc agent as a possible future task and used together with the current task to perform joint planning. For example, in addition to planning for the current task of preparing breakfast, the agent now plan for both preparing breakfast and coffee.

Since the decision-making policies of the other agents are unknown to the ad hoc agent, it assumes that its teammates will have access to the same information about the domain state, and predicts the actions that each teammate will execute over the next few steps using the learned agent behavior models (A in Figure 4.1). These models are learned by the agent using limited experience from its previous interactions with similar agents. The input is the state observations and the output is the action predictions of the other agents. For example, in the VirtualHome domain, the predicted actions of other agents may be represented as: human agent action — grab cereal.

These predicted actions are then integrated to the knowledge base of the ad hoc agent together with the anticipated tasks, current assigned task, and the state observations. The ad hoc agent’s prior commonsense knowledge base consists of relational descriptions of some domain attributes (e.g., arrangement of places), agent attributes (e.g., location of an agent), default statements (e.g., eggs are usually in the fridge), axioms governing actions and changes, e.g., the agent is aware that it cannot hold more than two objects at a time and that it cannot pick up objects that are not in a location next to it or can only move to a location nearby. The ad hoc agent then reasons over this knowledge base to compute its plan. Thus, the output from component A is the selected action of the ad hoc agent. Additionally, the human agent occasionally describes an agent’s actions (e.g., “Agent 2 cannot put the cake inside the microwave since the

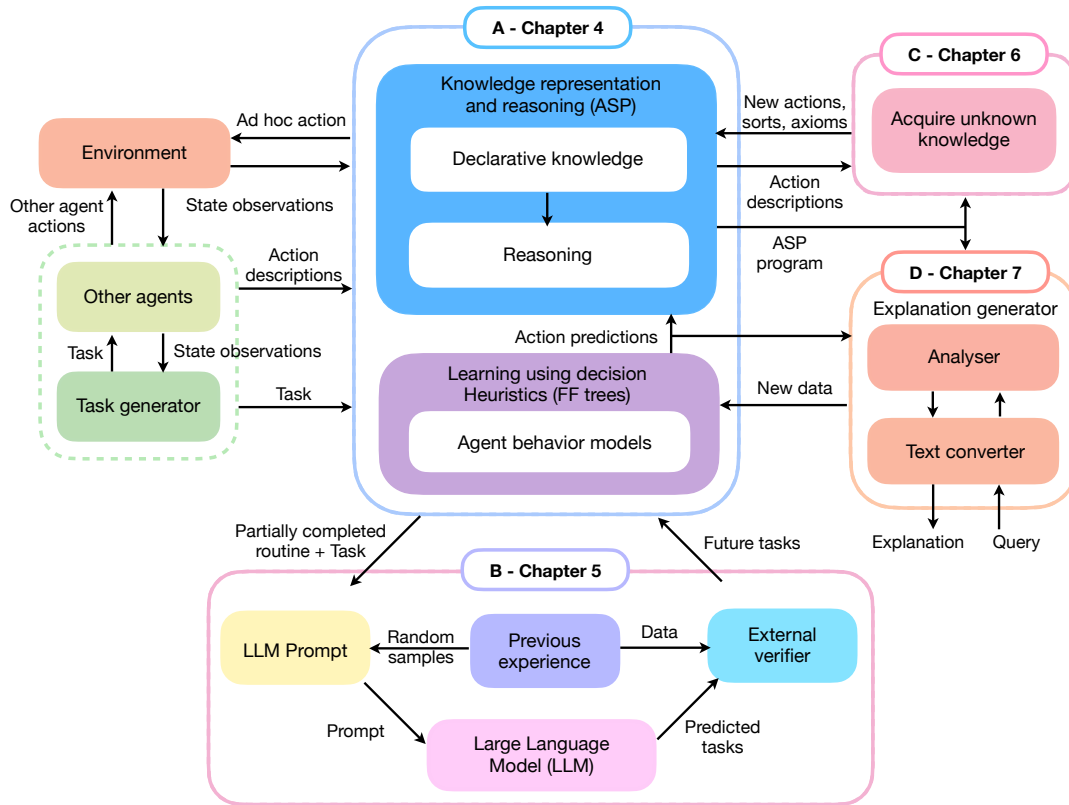


Figure 4.1: KAT architecture combines the complementary strengths of knowledge-based and data-driven reasoning and learning to enable an ad hoc agent to collaborate with its teammates.

microwave’s door is closed”), which is sent as input to component C in Figure 4.1, together with the current knowledge base. This input is used by the ad hoc agent to learn previously unknown domain knowledge in the form of objects, actions, and axioms. The ad hoc agent also uses state observations obtained during plan execution to learn axioms via decision tree induction. Both of these knowledge acquisition mechanisms are handled by the knowledge acquisition component C. Furthermore, the ad hoc agent provides relational descriptions as explanations of its decisions and beliefs, and those of other agents, in response to different types of questions, via the explanation generation component (D in Figure 4.1). The inputs to the component D are the query, the current knowledge base, and the action predictions of the other agents. The output of component D is the explanation generated by the agent.

We formalize the AHT problem scenario from above as a tuple $\langle E, S, A, T, Act, G, H \rangle$ where:

- E is the environment;
- S is the finite state space of E ;
- A is the *ad hoc agent* that uses the KAT architecture to plan and execute actions in E ;
- $T = \{t_1, \dots, t_n\}$ is the set of teammates (human or AI agents) whose decision-making policies are *unknown* to the ad hoc agent A a priori;
- $Act = Act_a \cup Act_{exo}$ is the joint action space, where Act_a is the set of actions available to A and Act_{exo} is the set of exogenous actions executed by teammates in T ;
- G is the set of possible tasks in E . At each episode, an external task generator dispatches a sequence $G_e = \langle g_1, \dots, g_m \rangle$, $g_i \in G$, to all agents one task at a time; and
- $H = \langle H_I, H_G \rangle$ is the history maintained by A , where $H_I^t = \langle (s_0, a_0^1, \dots, a_0^n), \dots, s_t \rangle$ records observed states and teammate actions up to time t , and $H_G = \{G_1, \dots, G_k\}$ records task sequences from previously completed episodes.

Each episode begins at time $t = 0$ in state $s_0 \in S$ and proceeds as follows.

1. At $t = 0$ and $s = s_0$, the external task generator produces a sequence of tasks $G_e = \langle g_1, \dots, g_m \rangle$, $g_i \in G$, for the episode and dispatches these to all agents one at a time as each task is completed.
2. At each step t , every agent receives a valid observation of the current domain state $s_t \in S$.
3. Each teammate uses (g_i, s_t) to independently determine and execute their individual actions $a \in Act_{exo}$ in E , collaborating towards the shared goal without communicating with A .
4. When only the current task g_i is known to A , it uses the *task anticipator* to anticipate (high-level) future tasks $g_j \in G_e, j > i$ in E . The task anticipator uses a pretrained LLM together with the completed task set $G_c \subset G_e$, the current task g_i , and the task history H_G to predict a set of future tasks $G_f = \{g_j \mid j > i\}$. The predictions are validated and adapted based on H_G . The A then jointly plans for the current task g_i and the first task from the predicted future task list $g_j \in G_f$.

5. At each step, A uses (s_t, g_i) together with learned *agent behavior models* to anticipate the likely near-future actions of the other agents A'_{exo} in E . These models are initially learned offline from limited prior state action observations and revised incrementally during execution based on real time observations of the agent behavior.
6. A selects its actions $a_t \in Act_a$ by reasoning over its *knowledge base* $\Pi(\mathcal{D}, \mathcal{H})$, the predicted other agents actions A'_{exo} , the current state s_t , and its goals (g_i, g_j) .
7. When the human agent produces an action description, A uses the *knowledge acquisition* component to learn previous unknown domain knowledge from the description in the form of objects, actions, and axioms and merges them into $\Pi(\mathcal{D}, \mathcal{H})$. Axioms are also learned from state observations via decision-tree induction. Note that these descriptions are treated as one-way environmental signals, not as direct communication from a teammate.
8. On demand, A uses the *explanation generation* component to answer queries about its own action choices, plans, and beliefs as well as other agents actions. A *text converter* parses the query to extract question type and relevant literals and pass this information to an *analyzer* that retrieves the corresponding axioms and literals from $\Pi(\mathcal{D}, \mathcal{H})$ and return them to the *text converter* to be inserted into a response template to produce a human-readable explanation. Any new experience encountered during this process is used to update the agent behavior models.

The agents do not exchange messages about their intentions or share their plans during task execution. Each agent acts independently, only relying on its own observations of the domain state s_t and the current task assignment g_i . When the human provides the action descriptions, they are treated as *environmental signals* available to the ad hoc agent for knowledge acquisition, not as a bidirectional communication channel between teammates. In the next section, we describe the first component of KAT: *knowledge representation and reasoning*.

4.2 Knowledge Representation

In KAT architecture, any given domain's transition diagram is described using an extension of the action language $\mathcal{AL}_d$ (Gelfond and Incelesan, 2013b). The domain repre-

sensation comprises a system description $\mathcal{D}$, a collection of statements of $\mathcal{AL}_d$, and a history $\mathcal{H}$. $\mathcal{D}$ has a sorted signature Σ which consists of basic sorts, domain attributes (statics, fluents) and actions described in terms of these basic sorts.

For example, Σ in our example scenario from the VirtualHome domain includes basic sorts such as *object*, *appliance*, *ad_hoc_agent*, *human*, and *step*, which represent household items, agents and steps for temporal reasoning. These sorts are arranged hierarchically, with some sorts being subsorts of the others (e.g., *apple* is a sub-sort of *food* that is a sub-sort of *graspable*, a sub-sort of *object*). Statics in Σ are domain attributes whose values cannot be changed, such as *next_to(location, locations)* that encode the relative arrangement of places in the VirtualHome domain.

The fluents in Σ can be *inertial*, which obey the laws of inertia and can be changed by the ad hoc agent's actions. If no such action occurs, then the value of inertial fluents remains unchanged. In contrast, *defined* fluents are not subject to inertia laws and cannot be directly caused by the ad hoc agent's actions. For example, inertial fluents in the VirtualHome domain include:

$$\begin{aligned} &at(ad_hoc_agent, location) \\ &in_hand(ad_hoc_agent, graspable) \\ &switched_on(appliance) \end{aligned} \tag{4.1}$$

which describe the location of the ad hoc agent, objects the ad hoc agent is holding in its hands, and the appliances that are switched on by the agent, respectively. The value of these attributes changes as a direct consequence of executing agents' specific actions. Defined fluents of the VirtualHome domain include:

$$\begin{aligned} &agent_at(external_agent, location) \\ &agent_hand(external_agent, graspable) \end{aligned} \tag{4.2}$$

which encode the location of the external (i.e., non-ad hoc) agents, and the objects they are holding, respectively. The ad hoc agent has no control over the value of these fluents, although the ad hoc agent's actions can influence the value of these fluents.

The actions in the VirtualHome domain include *agent_actions* that are performed by the ad hoc agent:

$$\begin{aligned} &grab(ad_hoc_agent, object) \\ &switch_on(ad_hoc_agent, appliance) \\ &move(ad_hoc_agent, location) \end{aligned} \tag{4.3}$$

which encode an agent's ability to grab, switch on and move to places. It also includes exogenous actions (*exo_actions*):

$$exo_grab(external_agent, object) \quad (4.4)$$

$$exo_switch_on(external_agent, appliance)$$

which encode the actions performed by the other agents in the domain.

Given a specific Σ , the domain's dynamics are described using three types of axioms: causal laws, state constraints, and executability conditions. In the VirtualHome domain, this includes statements such as:

$$open(A, E) \text{ \textbf{causes} } opened(E) \quad (4.5a)$$

$$\neg at(A, L1) \text{ \textbf{if} } at(A, L2), L1 \neq L2 \quad (4.5b)$$

$$\text{\textbf{impossible} } grab(A, O) \text{ \textbf{if} } on(O, E), \neg opened(E) \quad (4.5c)$$

where Statement 4.5(a), a causal law, implies that an agent (A) executing the $open(A, E)$ action causes an appliance E to be opened; Statement 4.5(b), a state constraint, implies that an agent (A) cannot be in two places ($L1, L2$) at the same time; and Statement 4.5(c), an executability condition, prevents the ad hoc agent (A) from trying to grab an object (O) from an appliance (E) with a closed door.

The history $\mathcal{H}$ of the domain is a record of observations of fluents at particular time steps, i.e., $obs(fluent, boolean, step)$, and of action execution at particular time steps, i.e., $hpd(action, step)$. It also includes initial state defaults, i.e., statements in the initial state that are believed to be true in all but a few exceptional circumstances (e.g., *cereal* is usually found inside the *kitchen_cupboard*). This representation does not assign numerical values to degrees of belief in these defaults, but rather supports reasoning with defaults and any specific exceptions.

Similar knowledge representations are used in the other domains evaluated in this thesis. Examples of knowledge representations can be found in the appendix and in our code repository (Dodampegama and Sridharan, 2026).

4.3 Reasoning

To reason with knowledge in our KAT architecture, the $\mathcal{AL}_d$ description is translated to a program $\Pi(\mathcal{D}, \mathcal{H})$ in CR-Prolog (Balduccini and Gelfond, 2003), an extension to Answer Set Prolog that supports consistency restoring (CR) rules. $\Pi(\mathcal{D}, \mathcal{H})$ contains

the Answer Set Prolog translation of statements from $\mathcal{D}$ and $\mathcal{H}$, inertia axioms, reality check axioms, closed world assumptions for defined fluents and actions, helper relations to reason over time steps, and helper axioms that define goals and guide planning and diagnosis.

Answer Set Prolog is based on stable model semantics in which the beliefs of an agent are represented by one or more stable models corresponding to self-consistent sets of conclusions derived from the agent's program (Section 2.1.1), and represents constructs difficult to express in classical logic formalisms. It encodes concepts such as *default negation* and *epistemic disjunction*, i.e., unlike “ $\neg a$ ” that states *a is believed to be false*, “*not a*” only implies *a is not believed to be true*, and unlike “ $p \vee \neg p$ ”, “*p or $\neg p$* ” is not tautologous. Each literal is true, false, or unknown, and the agent only believes that which it is forced to believe. Answer Set Prolog supports non-monotonic logical reasoning, i.e., the ability to revise previously held conclusions, which is essential for agents operating in practical domains with incomplete knowledge and noisy observations. The CR rules allow the agent to make assumptions (e.g., that a default statement does not hold) under exceptional circumstances to recover from inconsistencies, e.g., if the cereal is not observed inside the kitchen cupboard (its default location) during plan execution, the agent retracts the default assumption that the cereal is in the cupboard and considers alternative hypotheses that restore consistency between its beliefs and observations. Formally, we use the following fragment of CR-Prolog to represent default statements and CR rules:

$$val(f(\bar{x}), y, 0) \leftarrow val(body(d(\bar{x})), 0), not\ ab(d(\bar{x})) \quad (4.6a)$$

$$ab(d(\bar{x})) \stackrel{+}{\leftarrow} val(body(d(\bar{x})), 0) \quad (4.6b)$$

Statement 4.6a is a default statement, and Statement 4.6b is a CR rule. The CR rule indicates that if the agent believes the value of the body of the rule to be true, then it *may possibly* believe its head. This possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program. Statement 4.6a encodes that fluent $f(\bar{x})$ takes value y at step 0 unless the default $d(\bar{x})$ is known to be abnormal (i.e., when $d(\bar{x})$ does not hold under the current circumstances). In such scenarios, the CR rule can be applied as an exception to restore the consistency of the program. In other words, Statement 4.6b is applied only when assuming $f(\bar{x}) = y$ leads to a contradiction in the program. In such cases, the CR rule is triggered to retract the default assumption and restore the consistency of the program. In the previous example, if the cereal is not observed in the kitchen cupboard (its default location), the CR

rule retracts the default assumption and allows the agent to consider alternative locations for the cereal, restoring consistency between its beliefs and observations. While this process is considered in the context of non-monotonic logical reasoning as the construction of a stable model of the agent's beliefs at each step, it is also related to the process of postdiction studied in cognitive science, in which a person's interpretation of an earlier event is revised based on information obtained afterward (Shanahan, 1993).

The helper relations, $holds(fluent, step)$ and $occurs(action, step)$ implies that a particular fluent is true at a given step, and a particular action is part of a plan at a particular step. The inertia axioms take the form,

$$\begin{aligned} holds(F, I+1) &\leftarrow holds(F, I), \text{ not } -holds(F, I+1), I < n \\ -holds(F, I+1) &\leftarrow -holds(F, I), \text{ not } holds(F, I+1), I < n \end{aligned} \quad (4.7)$$

which state that in the absence of explicit evidence indicating a change, the value of an inertial fluent remains unchanged in the next step. Reality check axioms ensure that the agent's expectations remain consistent with its actual observations.

$$\begin{aligned} &\leftarrow obs(F, true, I), -holds(F, I) \\ &\leftarrow obs(F, false, I), holds(F, I) \end{aligned} \quad (4.8)$$

The closed world assumptions take the following form,

$$\begin{aligned} -occurs(A, I) &\leftarrow \text{not } occurs(A, I) \\ -holds(F, I) &\leftarrow \text{defined_fluent}(F), \text{ not } holds(F, I) \end{aligned} \quad (4.9)$$

which allow the agent to assume unknowns are false, simplifying the reasoning and making it faster by reducing the number of possibilities to consider. This is especially useful in large, complex domains like VirtualHome, which have many objects, containers, and locations. For example, there are $\approx 10^{25}$ states in the VirtualHome domain with just one ad hoc agent and one human.

Furthermore, it includes helper axioms to define goals and drive planning and diagnosis. The ad hoc agent may need to prioritize different goals at different times (e.g., on a day when the human is working from home, preparing breakfast takes higher priority than packing a bag for shopping). All reasoning tasks, i.e., planning, diagnostics, and inference are then reduced to computing *answer sets* of Π subject to some criteria (e.g., minimize costs) and extracting the action sequence; we do so using the SPARC

system (Balai et al., 2013). Example SPARC programs can be found in the appendix and in our code repository (Dodampegama and Sridharan, 2026).

Example 7 [*Reasoning in VirtualHome*]

Consider a scenario in which the goal of a team comprising a human and an ad hoc (assistive AI) agent is to prepare breakfast. To achieve this, the agents need to heat the poundcake, cook the cutlets, and bring them together with the water glass to the table. The Answer Set Prolog definition of the goal is:

$$\begin{aligned} \text{goal}(I) : & \text{--holds(heated(poundcake),I), holds(on(poundcake,kitchentable),I),} \\ & \text{holds(cooked(cutlets),I), holds(on(cutlets,kitchentable),I),} \\ & \text{holds(on(waterglass,kitchentable),I).} \end{aligned}$$

The ad hoc agent first uses a learned behavior model to predict the likely next action of the human. It then translates these predicted actions together with the current state observation into Answer Set Prolog literals. For example, if the model predicts that the human will grab the water glass, this is converted to the exogenous action *exo_grab(human, waterglass)*. Similarly, a state observation such as the poundcake being on the kitchen cupboard at step 0 is converted to *holds(on(poundcake, cupboard), 0)*. Once the action predictions and state observations have been translated into Answer Set Prolog literals, the agent solves the resulting program to compute an answer set that describes a potential world in which the desired goal is achieved over a sequence of time steps. This answer set contains literals such as:

$$\begin{aligned} & \text{occurs(move(ad_hoc_agent, fridge),0),} \\ & \text{occurs(open(ad_hoc_agent, fridge),1),} \\ & \text{occurs(grab(ad_hoc_agent, cutlets),2),} \\ & \dots \\ & \text{holds(in_hand(ad_hoc_agent, cutlets),3)} \end{aligned}$$

in which literals of the form *occurs(A, T)* indicate the action *A* to be executed by the agent at a specific time step *T*, e.g., the agent is to move to the fridge and fetch cutlets from it, and literals of the form *holds(F, T)* indicate the state of the world at specific time step, e.g., the agent is expected to be holding the cutlets at step 3. From this answer set, the agent extracts the literals that denote the plan to be executed by it, e.g., *move(ad_hoc_agent, fridge)*, *open(ad_hoc_agent, fridge)*, *grab(ad_hoc_agent, cutlets)*,

and other actions (not shown here), which, in conjunction with the actions it expects the human to execute, will enable the team to achieve the desired goal. This action sequence is then executed in the VirtualHome domain, with the corresponding observations and outcomes being added to $\mathcal{H}$.

In addition, heuristics are encoded to direct the search for plans. For example, the following statements encourage the ad hoc agent to select actions that minimize the total cost when computing action sequences to achieve a particular goal:

$$\begin{aligned} total(S) \leftarrow S = \text{sum}\{C, A : \text{occurs}(A, I), \text{cost}(A, C)\} \\ \#minimize\{S @ p, S : total(S)\}. \end{aligned} \quad (4.10)$$

Also, by assigning different costs to different actions and their outcomes, we can incorporate a preference specification to the program and identify the optimal answer sets with respect to these preferences.

As mentioned earlier, even with just one ad hoc agent and one human in the VirtualHome domain, the number of possible states can reach $\approx 10^{25}$, making it computationally expensive to compute plans with multiple steps. Similar challenges arise in many related domains, which are inherently complex due to the large number of objects and locations. To ensure scalability, we build on prior work on a refinement-based architecture (Sridharan et al., 2019) to enable the ad hoc agent to represent and reason at two resolutions. Specifically, a fine-resolution description ($\mathcal{D}_F$) is defined as a refinement of a coarse-resolution description ($\mathcal{D}_C$), with the agent now able to reason about aspects of the domain that were previously abstracted away. In our example scenario, the domain is organized into abstract regions in $\mathcal{D}_C$, with each region being refined in $\mathcal{D}_F$ into smaller regions that are components of the larger region, e.g., kitchen in $\mathcal{D}_C$ comprises kitchen table, kitchen counter, refrigerator in $\mathcal{D}_F$. Similarly, an object in $\mathcal{D}_C$ can comprise different parts in $\mathcal{D}_F$. The signature Σ_F of $\mathcal{D}_F$ is created first by expanding Σ_C of $\mathcal{D}_C$ to include the new sorts, actions, fluents, and statics. This enables the ad hoc agent to automatically *zoom to*, i.e., choose the relevant part of the relevant description based on the goal, and to transfer information between descriptions.

A common criticism of reasoning methods is that they need comprehensive domain knowledge. However, most of the steps for encoding the available knowledge can be automated, and the effort involved in encoding prior knowledge is much less than that needed to train purely data-driven systems (Sridharan and Meadows, 2018). Furthermore, Answer Set Prolog has been used by an international research commu-

nity to reason with *incomplete knowledge* (Erdem and Patoglu, 2018), and architectures that embed key principles such as refinement have indicated the ability to reason with available knowledge and revise it incrementally over time (Section 6). Moreover, modern Answer Set Prolog solvers have been shown to be efficient for large knowledge bases (Erdem and Patoglu, 2018).

4.4 Agent Modeling

Ad hoc agents are often deployed in dynamic environments where the decision-making strategies or policies of teammates are initially unknown and may evolve over time. In such settings, reasoning with just prior domain knowledge that can be incomplete or inconsistent can lead to poor performance. Thus, in addition to prior knowledge, KAT enables the ad hoc agent to reason with models that predict the action choices of other agents that are learned and revised rapidly. Specifically, the ad hoc agent uses the ER principle (Gigerenzer, 2020), which is based on Herb Simon’s definition of Bounded Rationality (Simon, 1956) and the algorithmic theory of heuristics (Gigerenzer and Gaissmaier, 2011), to quickly learn and update predictive behavior models of its teammates (or opponents).

4.4.1 Ecological Rationality and Heuristics

ER explores decision-making “in the wild”, i.e., under open world uncertainty with the space of possibilities not fully known, and characterizes behavior as a joint function of internal cognitive processes and the environment. It prioritizes *adaptive satisficing*: in the absence of comprehensive knowledge, true optimal decisions may be unknowable and not just hard to compute. Thus, heuristics (e.g., tallying, sequencing, fast and frugal methods) are used to ignore part of the information to make decisions more quickly, frugally, and/or accurately than sophisticated methods with many more free parameters (Gigerenzer and Gaissmaier, 2011). This approach has been shown to provide better performance than more sophisticated methods in practical applications (Gigerenzer, 2016).

4.4.2 Representational Choices and Models

Based on ER, KAT enables the ad hoc agent to select relevant attributes and learn models of the other agents’ behavior incrementally and from limited data. Specifically,

the agent learns an ensemble of FF trees to predict the behavior of each teammate (or type of teammate). Each FF tree provides a binary choice for a particular action, and the number of leaves in a tree is limited by the number of attributes (Katsikopoulos et al., 2021). Each level of the tree contains an exit, allowing the agent to make quick decisions based on available data. The agent makes decisions efficiently by evaluating the more informative attributes and stopping as soon as the first option exceeds an aspiration level, and choosing that option.

To construct the FF trees, we first allowed the ad hoc agent to interact with a teammate following a simple hand-crafted policy. The teammate selected its actions based solely on state observations and the assigned goal (task), and did not model the ad hoc agent’s actions or incorporated any preferences or priorities in its decision making. The ad hoc agent collected 3000 traces of the teammate’s action choices and the corresponding domain states. This data is used to construct the initial versions of the FF tree. To identify the most informative attributes from this data, we applied the statistical attribute selection method ANOVA to the collected data. ANOVA evaluates the statistical relationship between each candidate attribute and the teammate’s observed action choices by computing an F-statistic comparing the variation in attribute values across different action choices to the variation observed within each action choice. The resulting F-scores were used to rank the attributes according to their relevance to the teammate’s decision-making behavior (i.e., attributes with higher F-scores considered more relevant to the teammate’s decision-making behavior). The highest-ranked attributes were retained and the less informative attributes were discarded. Table 4.1 shows the final attributes selected by this procedure.

These selected attributes were then used as the input to the *ifan* algorithm (Algorithm 1, Section 2.5.3) to construct the FF trees (Phillips et al., 2017). Recall that the *ifan* algorithm further ranks the selected attributes according to their informativeness and incrementally constructs the FF tree using this ordering. Attributes with higher predictive performance (*bacc* in our case) are placed closer to the root of the tree, while less informative attributes appear at deeper levels. At each level, the algorithm provides an exit condition that allows an immediate decision whenever the corresponding attribute value satisfies the aspiration threshold. Figure 4.2 shows an FF tree learned by the ad hoc agent to model the behavior of its teammate in the VirtualHome domain for the action of *grab book*. It illustrates what the FF tree has learned to pay attention to when predicting whether the human will grab the book. It first checks the human’s x,y position, which serves as a proxy for proximity to relevant objects or regions in

<i>Description of the attribute</i>	<i>Number</i>
Immediate two previous actions of the agent	2
Position of the agent (x, y, z)	3
Orientation of the agent (x, y, z)	3
Objects associated with the goal	1
Any objects in the hand of the agent	1
Any objects in the hand of the remaining agents	1
Current and previous tasks	2
Flags (day of the week, going to office, guests expected)	3

Table 4.1: Attributes used to create the behavior models of the human agent in the example scenario from the VirtualHome domain. Number of attributes represent the *number of variables* in that attribute.

the environment (e.g., if the human is not in a region of the environment relevant to grabbing the book, the tree exits immediately with a prediction of *human action* $\neq$ *grab book* without evaluating any further attributes). If the location of the human is consistent, the tree then considers the human’s orientation, which further refines the estimate of where the human is directed and what object they are likely interacting with next. Finally, the tree checks the current task, which defines the set of objects/activities relevant at that moment. The same pose may lead to entirely different actions depending on the task, for example, preparing breakfast or setting up a work-station. Only if all these conditions are satisfied the FF tree will predict the next human action to be *grab book*. Since each FF tree provides a binary choice, we build an ensemble of trees and combine their outputs using a simple decision tree to produce the final action prediction for a given state. Furthermore, consistent agreement (or disagreement) between observations and model predictions can trigger model selection or revision of the existing models, as described in the next Section 4.5, allowing the ad hoc agent to quickly adapt to changes in the teammate’s behavior. We followed the same process to select attributes and train models in the other domains evaluated in this thesis, as discussed in Section 4.7.

State of the art methods for AHT often attempt to optimize performance in different (known or potential) situations by learning models offline from many (e.g., hundreds of thousands or millions of) examples. However, it is intractable to obtain such labeled examples of different situations in complex domains, and the learned models are truly

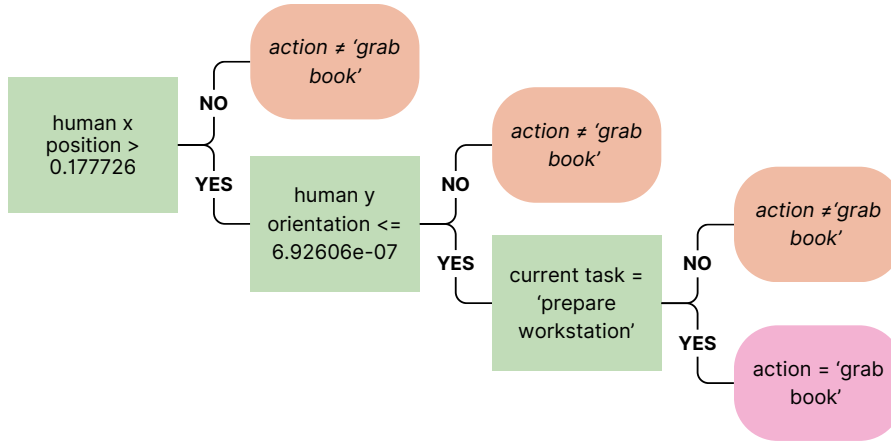


Figure 4.2: FF tree model of human behavior for the *grab book* action in the example scenario from the VirtualHome domain.

useful only if they can be learned and revised rapidly during run-time to account for previously unknown situations. On the other hand, FF tree based predictive models can be learned and revised incrementally and rapidly. Unlike many sophisticated methods for AHT, they adapt quickly without needing large amounts of data and make decisions fast using simple heuristics, making them well suited to the AHT setting.

4.5 Model Selection

While the example scenario in the VirtualHome domain involves an ad hoc agent collaborating with a single teammate, real world settings are often more complex. The ad hoc agent may need to coordinate with multiple teammates, and those teammates may include different types of agents whose behavior may change over time. In such dynamic settings, using static prediction models could be insufficient. Thus, KAT enables the ad hoc agent to respond to changes in the teammates' behavior by automatically revising the current model, switching to a relevant model, or learning new models.

Specifically, the ad hoc agent identifies and adapts to different types of teammates using its existing knowledge and recent observations. The agent compares the observed action choices of other agents with the outputs of the previously learned behavior models. Based on these observations, it selects the most likely teammate model and adjusts its planning and coordination strategies accordingly. This allows the agent to adapt effectively to previously unseen agent behaviors, enhancing collaboration and perfor-

mance.

For example, Algorithm 3 describes our approach for selecting a suitable model in the context of predicting the pose (position and orientation) of agents. First, for each other agent (teammate), the ad hoc agent predicts their actions under the current state using a relevant subset of behavior prediction models (Lines 1-4). It also stores the actual actions taken by these agents (including the ad hoc agent) at that step (Line 5).

It then periodically compares the existing models' predictions with the observed action choices of each agent over a sliding window of (Lines 7-10). If the predicted actions differ from the actual action choices, the agent simulates the effect of both the predicted and actual actions in a temporary environment, initialized with the state observed at the beginning of the sliding window, and records the resultant state (Lines 12-23). The agent then uses a graded strategy to compute the error, penalizing differences in orientation less than differences in position (Lines 24), and updates each model's score accordingly (Line 25). The agent then selects the model whose predictions best matches the observed behavior for subsequent use and revision (Line 28). If none of the models provide a good match over multiple steps, this acts as a trigger to learn a new model.

Next, we provide a brief argument for the soundness of the Algorithm 3. The algorithm uses a sliding window of few steps (three in our experiments) to balance responsiveness and stability. This prevents the agent from oscillating between models due to short-term prediction errors and provides a better balance. A window of one step would make the selection highly responsive but unstable, causing the agent to switch between models too frequently before settling on the correct one. The algorithm initialize the model scores to 100 at the beginning of each episode and penalize them based on prediction errors. If the scores of all models fall below a certain threshold (50 in our experiments), the algorithm triggers learning of a new behavior prediction model for the teammate. This ensures that the model scores never reach zero, and guarantees that the agent always has at least one usable model. During the period before a new model is available, the agent uses the model with the highest current score (i.e., the existing model whose predictions are most consistent with the observed teammate behavior) to generate predictions. When multiple models have similar scores, the algorithm selects the first model with the highest score.

Algorithm 3: Model Selection

Input: $\mathcal{A}$: other agents; $\mathcal{M}$: subset of behavior models; $\mathbf{A}_o$: actual action choices of each agent in current round of the game; $\mathbf{s}$: current state observation; *scores*: initial values (100) assigned to each agent-model combination, *step*: environment step number, *window*: number of steps before updating the models.

Output: model: selected model for each agent.

```

1  for  $i \in \{1, \dots, \mathcal{A}\}$  do
2      for  $m \in \mathcal{M}$  do
3           $A_{pred}[i, m, step] \leftarrow m(\mathbf{s})$ 
4      end
5       $A_{act}[i, step] \leftarrow \mathbf{A}_o[i]$ 
6  end
7  if  $step \bmod window = 0$  then
8      if  $A_{pred} \neq \emptyset$  and  $A_{act}[i] \neq \emptyset$  then
9          for  $i \in \{1, \dots, \mathcal{A}\}$  do
10             for  $m \in \mathcal{M}$  do
11                 if  $A_{pred}[i, m] \neq A_{act}[i]$  then
12                     Initialize environment( $n = |\mathcal{A}| + 1, S_{init}$ )
13                     for  $j = 0$  to  $window - 1$  do
14                          $s \leftarrow$  state of environment
15                          $s' = T(s, A_{pred}[i, m, j] + A_{act}[-i, j])$ 
16                     end
17                      $l_{pred}, o_{pred} \leftarrow s'$ 
18                     Initialize environment( $n = |\mathcal{A}| + 1, S_{init}$ )
19                     for  $j = 0$  to  $window - 1$  do
20                          $s \leftarrow$  state of environment
21                          $s'' = T(s, A_{act}[j])$ 
22                     end
23                      $l_{act}, o_{act} \leftarrow s''$ 
24                      $penalty \leftarrow abs(l_{act} - l_{pred}) + abs(o_{act} - o_{pred})/10$ 
25                      $scores[i, m] = scores[i, m] - penalty$ 
26                 end
27             end
28              $model[i] \leftarrow \arg \max_{m \in \mathcal{M}} (scores[i, m])$ 
29         end
30     end
31      $S_{init} \leftarrow \mathbf{s}$ 
32 end

```

The penalty function is designed to reflect the relative impact of different types of prediction errors. For example, errors in position are penalized more than errors in orientation as incorrect position predictions have a higher impact on the agent's planning in most domains. This is extended to incorporate domain-specific penalties in different domain. For example, in a domain where shooting action is available (Fort Attack in Section 4.7.1), incorrectly predicting a shooting is penalized more than incorrectly predicting an agent's location, given the higher impact of such errors.

In the following section, we present the main control loop of KAT, illustrating how the core components introduced in this chapter interact with each other and exchange information to enable the ad hoc agent's decision-making process.

4.6 KAT Pseudocode

Algorithm 4: Control Loop

Input: $\mathcal{N}$: number of trials; O_0 : initial object configuration; $\mathcal{A}$: other agents; $\mathcal{P}$: other agents' policies; $\mathcal{T}$: tasks for given day; $\mathcal{M}$: behavior models of other agents.

```

1 for  $i = 0$  to  $\mathcal{N} - 1$  do
2   Initialize environment( $n = |\mathcal{A}| + 1, O_0$ ), load  $\mathcal{P}, \mathcal{T}$ 
3    $\mathbf{s} \leftarrow$  state of environment
4    $step = 1$ 
5   while  $\mathcal{T}$  is not empty do
6      $task, \mathcal{T} \leftarrow \mathcal{T}[0], \mathcal{T}[1:]$ 
7     while  $\neg(\mathbf{s} \models goal(task))$  do
8        $\mathbf{A}_o \leftarrow \{\mathcal{P}_j(\mathbf{s}, task) \mid j \in \mathcal{A}\}$ 
9        $\mathbf{a}_{ah} \leftarrow selectAdHocAgentAction(\mathbf{s}, \mathcal{A}, \mathcal{M}, task)$ 
10       $\mathbf{A} = \mathbf{A}_o \cup \mathbf{a}_{ah}$ 
11       $\mathbf{s}' = T(\mathbf{s}, \mathbf{A})$ 
12      modelSelection( $\mathcal{M}, \mathbf{A}_o, \mathbf{s}, step$ )
13       $\mathbf{s} = \mathbf{s}'$ 
14    end
15     $step = step + 1$ 
16  end
17 end

```

Algorithm 5: Select Ad Hoc Agent Action

Input: $\mathbf{s}$: current state of the environment; $\mathcal{A}$: other agents; $\mathcal{M}$: behavior models of other agents; $task$: current task.

Output: $\mathbf{a}_{ah}$: selected action for the ad hoc agent.

```

1  $\mathbf{A}_p \leftarrow \{\mathcal{M}_j(\mathbf{s}) \mid j \in \mathcal{A}\}$ 
2  $initial\_state\_fluents \leftarrow \{f \mid f \text{ fluent literal representing } \mathbf{s}\}$ 
3  $exo\_actions \leftarrow \{a \mid a \text{ action representing } \mathbf{A}_p\}$ 
4  $goal \leftarrow \{g \mid g \text{ fluent literal representing } task\}$ 
5  $\Pi(\mathcal{D}, \mathcal{H}) = \text{load base Answer Set Prolog program}$ 
6  $fluents, exo\_actions, goal, sorts \leftarrow \{l \mid l \text{ is relevant,}$ 
    $l \in initial\_state\_fluents \cup goal \cup exo\_actions \cup \Pi(\mathcal{D}, \mathcal{H})\}$ 
7  $\Pi'(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup sorts \cup fluents \cup exo\_actions \cup goal$ 
8  $answer\_set \leftarrow \text{SPARC}(\Pi'(\mathcal{D}, \mathcal{H}))$ 
9  $actions \leftarrow \{a \mid a \in answer\_set\}$ 
10  $plan \leftarrow \text{order}(actions)$ 
11  $\mathbf{a}_{ah} \leftarrow plan[0]$ 
12 return  $\mathbf{a}_{ah}$ 

```

The overall control loop of KAT is described in Algorithm 4. First, the VirtualHome environment is set up with the initial configuration of the objects (e.g., milk inside the fridge instead of kitchen cupboard, or cereal inside the kitchen cupboard instead of kitchen rack) and the number of agents in the environment is specified. The other agents' policies $\mathcal{P}$, which are unknown to the ad hoc agent, and the tasks $\mathcal{T}$ that need to be completed within the given day are also loaded at the same time (e.g., the tasks for the day are prepare breakfast, prepare workstation, prepare coffee, prepare lunch, and clean kitchen) (Line 2). This determines the initial state $\mathbf{s}$ of the environment. Next, the first task of the day is assigned to the agent team for joint execution (Line 6). In each task, the other agents' actions are identified based on the current state, task, and their policies $\mathcal{P}$ (Line 8). The ad hoc agent's action is computed by reasoning with domain knowledge and learned models ($\mathcal{M}$) of other agents' behaviors (Line 9; Algorithm 5). The selected actions are then executed in the simulated environment to receive the updated state (Lines 10-11). The observed state information is used to incrementally learn and update $\mathcal{M}$ using Algorithm 3, e.g., when observations do not match predictions (Line 11). The updated state is used for the next step (Lines 13). This process is continued until all the tasks are achieved.

In Algorithm 5, the ad hoc agent first uses the learned behavior models ($\mathcal{M}$) of the other agents to predict their next action in the current state (Line 1). While Algorithm 5 demonstrates this procedure at a single time step, this can be repeated over multiple steps to predict and consider the other agent's actions over several future steps. The agent then converts the current state information, predicted actions of other agents and current task into Answer Set Prolog literals (Lines 2-4). For example, consider a scenario in the VirtualHome domain where the agent observes that the cake is on the kitchen table at step 0. This information is converted into an Answer Set Prolog fluent literal of the form *holds(on(cake,kitchentable),0)* by the ad hoc agent (Line 2). Similarly, the predictions from the behavior model for the human agent are converted to exogenous actions in Answer Set Prolog. For example a prediction of the *human will pick up the cake at step 0* is represented as *occurs(exo_grab(human,cake),0)* (Line 3). Finally, the task specification is mapped to the corresponding goal literals, for example, the task *prepare breakfast on a weekday* is converted to *holds(on(cereal, kitchentable), I)*, *holds(on(breadslice, kitchentable), I)*, *holds(on(bananas, kitchentable), I)*, indicating that the agent must prepare and place these items on the table (Line 4). The agent then loads the base Answer Set Prolog program, which encodes the agent's general knowledge of the domain (sorts, actions, fluents, and axioms as described in Section 4.2, Line 5).

Given the complexity of the VirtualHome domain, with its many objects, appliances, and locations, considering the entire state information would significantly increase the reasoning process of the agent unnecessarily complex and slower. Thus, the ad hoc agent uses the information on current state, anticipated actions of other agents and task to compute the relevant objects and regions in the domain that need special attention (Line 6) (e.g., if a task does not involve any objects from the bedroom, information related to that region can be excluded from the decision-making process. Similarly, any objects unrelated to the current task can be disregarded). This information is used to determine the sorts (objects, appliances, locations) and fluent that are relevant to current task, determining the level of abstraction to be used for reasoning, and to prioritize a goal (e.g., getting to a suitable region) to be added to the Answer Set Prolog program (Line 6). The updated Answer Set Prolog program is then solved using the SPARC system to compute its answer sets (Lines 7-8). The corresponding answer set provides the next action to be executed by the ad hoc agent (see Example 7). From this answer set, the agent extracts the literals that denote the plan to be executed by it (Line 9). These actions are then ordered considering the time step in the action,

and the first action of the plan is returned to Algorithm 4 as the selected action of the ad hoc agent.

Next, we provide a brief argument for the soundness of Algorithm 4 and 5. In Algorithm 4, the task list ($\mathcal{T}$) is guaranteed to be non-empty as it is checked for null prior to execution. Only one task is assigned to each agent as the previously assigned task is completed. Thus, the agent is never required to priorities between the assigned tasks. As the human agent selects its actions independently based on its observations, without depending on the ad hoc agents action, the loop is guaranteed to terminate, ensuring that each task will eventually be completed regardless of the ad hoc agent's performance.

In Algorithm 5, the SPARC program may not return an answer set (plan) for the ad hoc agent due to several reasons. First, this could be due to the maximum number of steps defined for the plan is insufficient to complete the current task. For example, the defined maximum number of steps is 20, but the agent needs 22 steps to reach its goal. In this case, the algorithm incrementally increases the step limit and reruns the SPARC program until either a valid answer set is found or a user-defined upper step limit is reached. If no answer set is found even after reaching the upper limit, the agent reduces the current goal to a subset of its current conditions. For example, if the task requires heating poundcake, cooking cutlets and bringing them to the table together with a glass of water:

$$\begin{aligned} \text{goal}(I) : & \neg \text{holds}(\text{heated}(\text{poundcake}), I), \text{holds}(\text{on}(\text{poundcake}, \text{kitchentable}), I), \\ & \text{holds}(\text{cooked}(\text{cutlets}), I), \text{holds}(\text{on}(\text{cutlets}, \text{kitchentable}), I), \\ & \text{holds}(\text{on}(\text{waterglass}, \text{kitchentable}), I). \end{aligned}$$

the agent will break it down to sub tasks and only focus on part of the goal (e.g., heating and bringing the poundcake to the table), and ignore the sub task associated with cutlets and water glass for the time being.

Second, SPARC may time out due to the large number of candidate answer sets in complex domains. In this case, the algorithm resets the step counter and reruns the SPARC program by incrementally increasing the step counter. If still no answer set is received (e.g., due to an inconsistency in the environment) the agent executes the immediate next action from the previously computed answer set, which is stored at each step. If this action fails (since the environment may have changed after the previous answer set was generated, it is possible that the next action is no longer valid under the current state), the algorithm increments a fail counter and discards the stored answer

set. In such cases, the agent falls back to a domain-specific safe action selection implemented in Python. This is a simple implementation that allows the agent to select a safe action under the current domain state. For example, in the soccer domain (Half Field Offense in Section 4.7.1), the agent will dribble if it has possession of the ball or move toward the goal if it does not have the control of the ball. This ensures that the agent continues to function, even though sub-optimally, rather than failing abruptly. If the selected action continues to fail due to persistent inconsistencies in the environment, the algorithm accepts the action as failed after repeated unsuccessful attempts, and hand over the task to another teammate (e.g., human), allowing the overall task to continue toward completion.

The exogenous actions predicted by the behavior models for the teammates are validated against the current state by the algorithm before adding them to the SPARC program. This ensures that the predicted actions are consistent with the current domain state. Any predictions inconsistent with the current state are discarded. When multiple answer sets are returned by SPARC, the first answer set is selected.

The complete code can be found in our code repository (Dodampegama and Sridharan, 2026). The components described in this chapter *knowledge representation and reasoning* and *learned agent behavior models* form the foundational version of our KAT architecture. Thus, in the next section, we present the experimental setup used to evaluate the performance of our base architecture.

4.7 Experimental Procedure

To evaluate the effectiveness of KAT, we designed and conducted a series of experiments as described below. We begin this section by introducing the set of environments used throughout this thesis to evaluate different components of our KAT architecture (Section 4.7.1). We then describe the different agent types deployed in each environment, highlighting their roles and behaviors (Section 4.7.2). Finally, we outline the experimental procedure for evaluating the base architecture, including the performance criteria and baselines used (Section 4.7.3).

4.7.1 Environments

We implemented and evaluated our architecture across three distinct environments, each selected to assess performance of KAT under varying task complexities. Below,

we provide a detailed description of each environment.

Fort Attack: *Fort Attack* is a simulated benchmark domain for multiagent collaboration (Deka and Sycara, 2021). Figure 5.5a shows a screenshot from this domain, where three guards (in green) are trying to protect a fort from three attackers (in red). An episode of the game ends if:

- (a) an attacker reaches the fort;
- (b) all members of the attacker team are killed; or
- (c) guards manage to protect the fort for a sufficient time period (200).

The state space of the domain is continuous. For reasoning, we discretize it into a 20×15 grid. The action space is discrete: at each step, an agent can choose to move in any of the four cardinal directions with a fixed velocity, turn clockwise or counter-clockwise, do nothing, or shoot; if an opponent agent is within the shooting range, it would be eliminated. The domain is fully observable, i.e., each agent can observe the state of other agents at each step. The domain provides four types of built-in policies for guards and attackers, as described in Section 4.7.2. All other agents except the ad hoc agent will be following these policies.

Half Field Offense: This simulated 2D soccer domain (Figure 5.5b) is a complex benchmark for multiagent and AHT methods (Hausknecht et al., 2016). The domain consists of an offense team trying to score a goal against a defense team. An episode of the game ends when:

- (a) offensive team scores a goal;
- (b) ball leaves field;
- (c) defensive team captures the ball; or
- (d) maximum episode length (500) is reached.

Half Field Offense supports two continuous state space abstractions: low, high; we use the high-level feature set and discretize the state space into a 20×20 grid for reasoning. There are three abstractions of the action space: primitive, mid-level, and high-level; we use a combination of mid-level and high-level actions. The domain is fully observable, ensuring noise free perception. However, the actions are noisy, causing differences between the intended and actual outcomes. For example, occasional failures in actions, such as missed kicks, can be observed. There are two versions of



Figure 4.3: Screenshots from Fort Attack, Half Field Offense and VirtualHome domains.

the domain: (i) limited version: two offense agents and two defense agents (including goalkeeper); (ii) full version: four offense agents, five defense agents (including goalkeeper). Similar to prior AHT methods, agents other than the ad hoc agent are selected from teams created in the 2013 RoboCup 2D simulation league competitions, which are described in Section 4.7.2.

VirtualHome: VirtualHome is a realistic physics-based 3D simulation environment for multiagent collaboration, where AI agents collaborate to complete household tasks (Puig et al., 2018). Figure 4.3c shows a snapshot from the domain where three agents are setting up a work-station. We implemented an external task generator in the environment to produce a sequence of tasks to complete within an episode. Each episode continued until the agent team completed all the tasks for the given day. The domain features a continuous state space and consists of multiple rooms, with each room consisting of day-to-day objects required for performing household activities. The agents can interact with the environment through high-level actions, e.g., moving to places, picking up or placing objects, switching appliances on or off, and opening or closing appliances. Completing a task requires a sequence of such actions to be computed and executed by members of the team without any direct communication between them. Policies used by agents other than our ad hoc agent are described in Section 4.7.2.

4.7.2 Agent Types

In this section, we describe the policies followed by the agents other than the ad hoc agent in the three domains introduced earlier.

Fort Attack: We used two sets of policies for the agents other than the ad hoc agent following our KAT architecture: *handcrafted policies* and *built-in policies*. Hand-

crafted policies were manually designed rule-based strategies that encode simple behaviors without any learning from data or logical reasoning over domain knowledge (e.g., force field-based movement strategies or spread and shoot behaviors implemented as programmatic rules). Built-in policies, by contrast, are data-driven policies provided with the domain that are learned by the agent from a large dataset of examples using graph neural networks. The key distinction between the two is therefore the means by which they are constructed: handcrafted policies reflect explicit human design choices encoded as programmatic rules, whereas built-in policies are complex policies learned automatically from data. We briefly describe these policies below.

Handcrafted Policies:

- *Policy1:* guards spread and actively pursue the attackers to shoot, attackers spread and approach fort.
- *Policy2:* guards stay close to the fort and shoot, attackers spread and shoot their opponents.

Built-in Policies:

- *Policy220:* guards place themselves in front of the fort and shoot continuously; attackers try to approach the fort.
- *Policy650:* guards try to block the fort; attackers try to sneak in from all sides.
- *Policy1240:* guards spread and shoot the attackers; attackers sneak from all sides.
- *Policy1600:* guards spread and move from the fort if needed; some attackers approach and shoot at the guards to divert their attention, while other attackers wait outside the guards' shooting range for the right moment to sneak in.

Ad hoc agent represented a guard agent in this domain and used the proposed KAT architecture to select its actions.

Half Field Offense: The offensive team agents in this domain were based on five teams: *Helios*, *Gliders*, *Cyrus*, *Axiom*, *AUT*; that are created for the 2013 RoboCup simulation league competitions. For the defenders, we used *Agent2D* agents, whose policy was derived from *Helios*. The strategies of these agents were trained using data-driven (probabilistic, deep, reinforcement) learning methods. We selected these policies as they have been used in previous AHT studies, enabling direct comparison

with existing work. The ad hoc agent, following our KAT architecture, represented an offense team member in each game episode.

VirtualHome: We modeled the teammate in this domain as a human agent collaborating with the ad hoc agent. The human agent was based on an Answer Set Prolog program that did not model its teammates’ actions, i.e., the human agent selected actions only based on its own knowledge and observations, without explicitly expecting support from the other agents. Since the human cannot predict the ad hoc agent’s actions, the Answer Set Prolog program governing the human’s action choices did not include any axioms for reasoning about these actions. Furthermore, the human’s Answer Set Prolog program encoded certain preferences, priorities, and capabilities that are not initially known to the ad hoc agents but may be captured over time in the models they learn to predict the behavior of the human (e.g., when preparing breakfast, the human toasted the bread first before preparing the cereal). The ad hoc agent used the KAT architecture for selecting its actions.

4.7.3 Experimental Procedure

This section details the experimental procedure used to evaluate the base architecture of KAT, including the performance measures and baselines.

Fort Attack

In the Fort Attack domain, we evaluated KAT architecture using two experimental setups,

- **Exp1**, in which all agents except the ad hoc agent followed the handcrafted policies; and
- **Exp2**, in which all agents except the ad hoc agent followed the built-in policies.

To simulate learning from limited examples, we first collected state observations and action choices of other agents by running the handcrafted policies in the Fort Attack domain. We then trained an ensemble of FF trees for different agent types only using 10000 of examples and the attributes from Table 4.2. These learned behavior prediction models were then used by the ad hoc agent to predict the actions of other agents up to two steps into the future.

To evaluate the ad hoc agent’s ability to adapt to previously unseen behaviors, the agent was not provided any prior experience or models of the agents following the

<i>Description of the attribute</i>	<i>Number</i>
Immediate previous action of the agent	1
Position of the agent (x,y)	12
Distance from agent to center of field	6
Agents' polar angle with center of field	6
Orientation of the agent	6
Distance from agent to fort	6
Distance to nearest attacker from fort	1
Number of attackers not alive	1

Table 4.2: Attributes considered for models of other agents' behavior in Fort Attack domain. Number of attributes represent the *number of variables* in each attribute times the *number of agents*.

domain's built-in policies. In **Exp2**, the other agents followed these built-in policies, which the ad hoc agent had never encountered before. The ad hoc agent used the models trained from handcrafted policy data to predict the teammates' actions.

In **Exp3**, we extended this setup by allowing the ad hoc agent to revise these models on the fly, using the model selection algorithm described in Section 4.5. This enabled the ad hoc agent to incorporate new data observed during interaction with the previously unseen agents.

Each episode in the Fort Attack domain consisted of two teams: three guards, three attackers. The ad hoc agent represented one of the guard agents. Prior commonsense domain knowledge of the ad hoc agent included relational descriptions of some domain attributes (e.g., location of agents), agent attributes (e.g., shooting range), and axioms governing change such as, an agent can only shoot others within its shooting range. The ad hoc agent automatically selected and pursued different goals depending on their priorities toward the overall objective of ensuring the fort's safety, e.g., shoot an attacker in the critical region near the fort, or move to cover an unprotected region:

$$goal(I) \leftarrow holds(agent_shot(A), I), attacker(A), holds(critical_region(A), I) \quad (4.11)$$

$$goal(I) \leftarrow holds(in(ah_agent, X, Y), I), holds(free_region(X, Y), I) \quad (4.12)$$

If an attacker entered a predefined critical region near the fort (i.e., the distance from the attacker to fort is below a predefined value) the agent prioritized the first goal. Otherwise, the agent aimed to spread around the fort keeping a constant distance in

x,y directions with the other guard agents. During planning, the ad hoc agent selected actions to minimize the total cost of achieving the selected goal using statements 4.10 and axioms defining the cost of actions (e.g., the agent was penalized for moving too close to other guards exceeding a minimum distance, and for moving away from the attackers).

For comparison, we considered three baselines without our ad hoc agent:

- **Base1** in **Exp1** with all agents following handcrafted policies;
- **Base2** in **Exp2** with all agents following built-in policies; and
- **GPL**, a state of the art AHT method based on graph neural networks (Rahman et al., 2021) as an extension of **Exp2**.

In **Exp3** (for evaluating the model selection algorithm from Section 4.5) we used a team of agents based on a mix of policies described earlier (under **Exp1** and **Exp2**); specifically, agents other than our ad hoc agent randomly chose a policy from the mix of available policies. The baselines for this experiment were:

- **Base3:** other agents followed a random mix of handcrafted policies. The ad hoc agent did not revise the learned behavior models or use the model selection algorithm.
- **Base4:** other agents followed a random mix of handcrafted policies. The ad hoc agent used a model selection algorithm without a graded strategy to compare the predicted and actual actions, i.e., fixed penalty assigned for action mismatch in Line 6 of Algorithm 3.
- **Base5:** other agents followed a random mix of built-in policies. The ad hoc agent did not revise the learned behavior models or use the model selection algorithm.
- **Base6:** other agents followed a random mix of built-in policies. The ad hoc agent used the model selection algorithm without a graded strategy to compare predicted and actual actions, i.e., fixed penalty assigned for action mismatch in Line 6 of Algorithm 3.

The **performance measures** included the win percentage of the guards' team as the ad hoc agent was part of the guards team, prediction accuracy of the learned behavior models, and fraction of times a shooting guard eliminated an attacker. For each experiment described above, we performed 150 episodes. For GPL, we took the average results from their supplementary material.

<i>Description of the attribute</i>	<i>Number</i>
Position of the agent (x, y)	8
Goal opening angle	2
Proximity to the nearest opponent	2
Position of the ball (x, y)	2

Table 4.3: Attributes used to learn teammates (offense) behavior models in the full version of the Half Field Offense domain. Number of attributes represent the *number of variables* in each attribute times the *number of agents*.

<i>Description of the attribute</i>	<i>Number</i>
Position of the agent (x, y)	10
Position of the ball (x, y)	2

Table 4.4: Attributes used to learn defense agents' behavior models in the full version of the Half Field Offense domain. Number of attributes represent the *number of variables* in each attribute times the *number of agents*.

Half Field Offense

Similar to the initial phase in the Fort Attack domain, we used handcrafted policies encoding basic behavior of the agents in the Half Field Offense domain and collected observations of states before and after each transition in the episode. Since the actions of other agents are not directly observable in the Half Field Offense domain, they were computed from the observed state transitions. To evaluate the ability to learn from limited data, we only used data from 300 episodes for each type of agent to create the FF tree models for behavior prediction, which were then revised (as needed) and used by the ad hoc agent during reasoning. Table 4.3 and Table 4.4 list the attributes used to create these models.

We evaluated KAT's performance with each built-in external team, (i.e., all offense agents other than the ad hoc agent were based on one randomly selected external team in each episode) using two experimental setups,

- **Exp4**, limited version, i.e., two offense players (including ad hoc agent) against two defense agents (including goalkeeper).
- **Exp5**, full version, i.e., four offense players (including ad hoc agent) played

against five defense agents (including goalkeeper).

The ad hoc agent’s prior commonsense knowledge included domain attributes (e.g., location of the ball), agent attributes (e.g., ball shooting angle), and axioms describing domain changes such as whether the agent had a better shooting opportunity, or whether the defending team members are too close to the agent. The agent automatically selected and prioritized different goals at different times, e.g., attempt to score when in close proximity to the goal, or advance the ball towards it:

$$goal(I) \leftarrow holds(scored_goal, I) \quad (4.13)$$

$$goal(I) \leftarrow holds(in(ball, X, Y), I). \quad (4.14)$$

A suitable goal was selected and included in the Answer Set Prolog program automatically during run-time.

As the baselines for **Exp4-Exp5**, we used recent (state of the art) AHT methods: PPAS (Santos et al., 2021), and PLASTIC (Barrett et al., 2017). These methods considered the same external agent teams mentioned above, allowing us to compare our results with the results reported in their papers. We also evaluated against a baseline that only used non-monotonic logical reasoning with prior knowledge but without any behavior prediction models (i.e., the ad hoc agent was unable to anticipate the actions of other agents).

In the Half Field Offense domain, the key **performance measure** was the fraction of games that ended with the offense team scoring a goal since the ad hoc agent was a member of the offense team. We also measured the accuracy of the learned behavior models. We conducted 1000 episodes for each experiment described above.

VirtualHome

As described in Section 4.4, each predictive model in the VirtualHome domain was an ensemble of FF trees based on attributes in Table 4.5 and only 100 prior traces of human actions and domain state.

To evaluate KAT, we first designed **Exp6** by constructing 720 different configurations in the domain, each with a different arrangement and status of objects in the initial conditions, e.g., bread on the kitchen table instead of the counter, microwave open instead of closed. We then randomly chose 100 of these configurations and measured the ability of the agent team to achieve the shared goal. As the focus of this chapter is to evaluate the core components of our architecture (i.e., knowledge representation

<i>Description of the attribute</i>	<i>Number</i>
Immediate two previous actions of the human	2
Immediate two previous objects human interacted	2
Position of the agent (x, y, z)	3
Orientation of the agent (x, y, z)	3
Distance from human to the kitchen table	1
Distance from human to the kitchen counter	1
Number of objects on the kitchen table	1

Table 4.5: Attributes used to create the behavior models of the human agent in the VirtualHome domain. Number of attributes represent the *number of variables* in that attribute.

and reasoning, agent behavior models), we limit our experiments here to a single collaborative task of *preparing breakfast*. In Chapter 5, we introduce the task anticipation component, and extend these experiments to support and reason about multiple tasks. The results presented here form the foundation for the capabilities explored in the next chapter.

As the baseline for **Exp6**, we used an Answer Set Prolog-based agent whose actions were determined by an Answer Set Prolog program that has the same knowledge as the ad hoc agent but does not consider the models predicting human behavior.

The **performance measures** were the number of steps (i.e., plan length) and the total time taken by the team (i.e., human and ad hoc agent) to complete the task. We also measured the accuracy of the behavior prediction model learned for the human agent by the ad hoc agent.

4.8 Results

In this section, we report and analyze the results of the experiments described in Section 4.7 under each environment. These experiments were conducted to evaluate the core components of our KAT architecture, *knowledge representation and reasoning*, and *learned agent behavior models* (A in the Figure 4.1), that were discussed in this chapter. We begin with the results of the Fort Attack domain.

<i>Policy</i>	<i>With ad-hoc agent</i>	<i>Without ad-hoc agent (Base1)</i>
Policy1	90%	85%
Policy2	67%	62%

Table 4.6: Average percentage (%) of episodes in which guards won with handcrafted policies and learned agent models (Exp1).

Fort Attack

Table 4.6 shows the average percentage of episodes won by the guards team in the Fort Attack domain in **Exp1**. When one of the guards was replaced with the ad hoc agent using KAT, there was a significant improvement in the number of episodes in which the guards won, compared to **Base1**, where all agents used the handcrafted policies. Given that the ad hoc agent used models of other agents’ behavior learned from limited training examples of handcrafted policies, the higher percentage of wins is a positive outcome.

Table 4.7 summarizes the results of **Exp2**. We observed that the team of guards with our ad hoc agent, using the behavior models trained from the handcrafted policies, was able to adapt to the previously unseen built-in policies of the Fort Attack domain. In particular, there was a statistically significant improvement in the percentage of wins for the team of guards for these policies. The percentage of episodes in which the team of guards won was rather small for policy 1600, although there was a significant improvement when our ad hoc agent was one of the guards (7% from 2%). This policy was particularly challenging for the guards as some attackers tried to draw the guards away from the fort and shoot them, while others stayed back and waited for an opportunity to sneak in unnoticed. These results indicate that our architecture enables adaptation to different teammate and opponent types, and indicate the benefits of reasoning and learning guiding each other.

Table 4.8 shows the prediction accuracy of the behavior models learned by the ad hoc agent for the guards and attacker agents following the handcrafted policies in the Fort Attack domain. The accuracy of the models ranged from 60 – 87%, i.e., there were errors. For example, in Guard Policy2 (60%), guards stayed closer to the fort and shoot the attackers when in range. This led them to use the “stay idle” action as frequently as the “shoot” action. Since both actions involved the agent not moving and staying in one location, the learned model sometimes confused them. Furthermore, since the

<i>Policy</i>	<i>With ad-hoc agent</i>	<i>Without ad-hoc agent (Base2)</i>
Policy 220	85%	77%
Policy 650	35%	20%
Policy 1240	70%	52%
Policy 1600	7%	2%

Table 4.7: Average percentage (%) of episodes in which guards win with previously unseen built-in policies (Exp2).

<i>Agent Model</i>	<i>Accuracy</i>
Guard Policy1	85.5%
Guard Policy2	60.0%
Attacker Policy1	86.9%
Attacker Policy2	85.2%

Table 4.8: Prediction accuracy of learned behavior models.

guards rarely used the “going up” action, it was represented by very few instances in the training data, making it difficult for the model to learn a reliable prediction for this action. However, when used together with other components of our architecture, these models were sufficient to improve the performance of the team of guards compared with baselines without our ad hoc agent, as shown in Tables 4.6–4.7. These results indicate the benefits of not over-fitting on the training set and allowing rapid revision of the learned models.

Further to the above results, we also measured the fraction of times a guard agent’s shooting eliminated an attacker, i.e., *shooting accuracy*. We compared our ad hoc guard agent with an ad hoc guard agent using GPL, and a guard using **Base2**, in the context of built-in policies; both of whom used policies based on orders of magnitude larger number of training samples. Agents other than the ad hoc agent in a team were selected randomly from the mix of available built-in policies in each episode. As illustrated in Table 4.9 our ad hoc guard’s average shooting accuracy was significantly better than that of the ad hoc guard agent using GPL or the guard agents using the built-in policies of the Fort Attack domain. Also, the GPL-based agent was trained for 60×160000 steps, whereas our ad hoc agent reasoned with prior domain knowledge and predictive models learned from 10000 samples to adapt to previously unseen policies.

<i>Policy</i>	<i>Shooting accuracy</i>
Ad hoc agent	34%
Base2	10%
GPL	30%

Table 4.9: Average shooting accuracy (%) of the ad hoc agent with previously unseen built-in policies.

<i>Experiment</i>	<i>Win</i>
Without model selection (Base3)	63%
When using direct comparison (Base4)	68%
With model selection algorithm (KAT)	73%

Table 4.10: Win percentage (%) for guards with handcrafted policies in Fort Attack domain. Model adaptation improves performance.

Next, Table 4.10 summarizes the results of using our model selection algorithm in **Exp3**. When the other agents followed the handcrafted policies and the model selection mechanism was not used by the ad hoc agent (**Base3**), the team of guards had the lowest winning percentage. When the ad hoc agent used the model selection algorithm with a fixed penalty assigned for any mismatch between predicted and actual actions (**Base4**), the performance of the team of guards improved. When the ad hoc agent used KAT’s model selection method (Algorithm 3), the winning percentage of the team of guards was substantially higher than the other two options. These results indicated that *KAT’s adaptive selection of the behavior prediction models improved performance*.

Further results of **Exp3** are summarized in Table 4.11. We observed that KAT enabled the ad hoc agent to adapt to previously unseen teammates and opponents that used the Fort Attack domain’s built-in policies, based on the model selection algorithm and the online revision of the behavior models learned from the hand-crafted policies. KAT provided the best performance compared with not using any model adaptation or selection (**Base5**), and when model selection assigned a fixed penalty for action mismatch (**Base6**).

<i>Experiment</i>	<i>Win</i>
Without model selection (Base5)	47%
When using direct comparison (Base6)	45%
With model selection algorithm (KAT)	55%

Table 4.11: Win percentage (%) for guards with built-in policies in Fort Attack domain. Model adaptation improves performance.

<i>Version</i>	<i>KAT</i>	<i>Reasoner</i>	<i>PPAS</i>	<i>PLASTIC</i>
Limited (2v2)	79%	67%	80%	80%
Full (4v5)	30%	26%	20%	20%

Table 4.12: Fraction of goals scored by the offense team in Half Field Offense domain in the limited version (2v2) and full version (4v5).

Half Field Offense

We next describe the results from the Half Field Offense domain. Table 4.12 summarizes results of **Exp4** and **Exp5**; which evaluated KAT’s performance against a baseline that had the ad hoc agent only reasoning with prior knowledge, (i.e., without any learned models predicting the behavior of other agents), and two data-driven state of the art AHT baselines (PPAS, PLASTIC). These data-driven baselines required orders of magnitude more training examples and did not support reasoning with prior domain knowledge. KAT consistently outperformed the reasoning only baseline in both the limited and full versions of the Half Field Offense domain, achieving a significantly higher fraction of goals scored (i.e., games won) by the offense team. Next, the fraction of goals scored by the team of offense agents including KAT was comparable with the goals scored by the data-driven baselines for the limited version, and substantially better than goals scored by the baselines for the full version. These results emphasized the importance of combining structured reasoning with learned models, especially in complex multi-agent environments.

The prediction accuracy of the learned behavior models created for the limited version (**Exp4**) and full version (**Exp5**) of the Half Field Offense domain are summarized in Tables 4.13 and 4.14 respectively. These behavior models were learned for the agents other than the ad hoc agent using data from 300 episodes (for each external

<i>Agent Type</i>	<i>Accuracy</i>
Helios	78.2%
Gliders	83.2%
Cyrus	69.5%
Aut	72.4%
Axiom	76.2%
Agent2D	79.8%

Table 4.13: Prediction accuracy (%) of the learned agent behavior models in limited (2v2) version of the Half Field Offense domain (Exp4).

<i>Agent Type</i>	<i>Accuracy</i>
Helios	86.0%
Gliders	66.4%
Cyrus	77.6%
Aut	67.7%
Axiom	73.6%
Agent2D	71.9%

Table 4.14: Prediction accuracy (%) of the learned agent behavior models in full (4v5) version of the Half Field Offense domain (Exp5).

agent type). This translated to orders of magnitude fewer training samples than the few hundred thousand training samples used by state of the art data-driven methods that do not reason with domain knowledge. The prediction accuracy varied over a range for the different agent types. Although the accuracy values were not very high, the models could be learned and revised quickly during run-time; also, as shown in Table 4.12, these models resulted in good performance when the ad hoc agent also reasoned with prior knowledge.

VirtualHome

In this section, we present and analyze the results of our experiments in the VirtualHome domain. First, Table 4.15 summarizes the results of **Exp6**. Since the plan length and task completion time can vary substantially between trials based on the configuration, we computed the values of the performance measures for KAT as a fraction (i.e., ratio) of those for the baseline in each trial. The average of these ratios is reported in Table 4.15. When the ad hoc agent used KAT, the number of steps and the time taken to achieve the task reduced compared to the baseline.

Moreover, the prediction accuracy of the behavior models learned by the ad hoc agent for the human agent using the ensemble of FF trees was **85.19%** and the agent was able to revise these models to both improve accuracy and adapt to changes in the human’s behaviors over time.

<i>Architecture</i>	<i>Steps</i>	<i>Time</i>	<i>Steps (raw)</i>	<i>Time (raw)</i>
KAT	0.89 ± 0.11	0.90 ± 0.19	13.5	143.4
Baseline	1 ± 0.05	1 ± 0.04	15.4	160.2

Table 4.15: Mean and standard deviation of number of steps and time taken by KAT architecture to achieve the goal, expressed as a fraction of these values for the baseline, alongside the average absolute values.

4.9 Chapter Summary

This chapter introduced KAT, a novel hybrid architecture for AHT, which combined knowledge-based reasoning and data-driven learning to enable an ad hoc agent to reason with structured knowledge and predict the behavior of unfamiliar teammates. We discussed two key components of our architecture, (1) *knowledge representation and reasoning* which enabled the agent to perform non-monotonic logical reasoning with prior commonsense domain knowledge; (2) *learned agent behavior models* an ensemble of FF decision trees learned by the agent from limited examples to model the behavior of other agents. We also discussed how the agent rapidly identified the need for and revised these learned models, to better adapt to its teammates.

Through experiments conducted in three benchmark domains for AHT: Fort Attack, Half Field Offense and VirtualHome, we indicated that incorporating both knowledge based reasoning and heuristic based learning enabled the ad hoc agent to adapt and collaborate efficiently using limited training data, outperforming baselines that completely relied on logical reasoning or data driven learning. Based on the above findings, we make the following key observations:

- **Observation 1:** *KAT enables adaptation to different teammate and opponent types* – Supported by all domains (**Exp1, 2, 4, 5 and 6**).
- **Observation 2:** *KAT supports incremental learning of models of other agents' behavior from limited training examples* – Supported by all domains (**Exp1, 2, 3, 4, 5 and 6**).
- **Observation 3:** *KAT's performance is comparable or better than state of the art baselines in different scenarios while requiring much less training*

- Supported by Fort Attack and Half Field Offense domains (**Exp2, 4 and 5**).

These observations support Research Question1 from Section 1.2, and establish the foundation for the extensions introduced in the coming chapters, where we incrementally incorporate additional components to support extended functionalities and evaluate their contributions.

Chapter 5

Task Anticipation and Partial Observability

The previous chapter introduced the base architecture of KAT that combined knowledge-based and data-driven reasoning and learning to enable an ad hoc agent to perform non-monotonic logical reasoning with prior commonsense domain knowledge and an ensemble of fast and frugal decision trees learned from limited examples to model the behavior of other agents. We evaluated the base architecture’s performance in three key multi-agent environments with varying complexities under full observability and demonstrated that our ad hoc agent can adapt to previously unseen teammates and opponents, rapidly revise the learned models, perform comparable or better than baselines including state of the art data-driven methods that require much larger training datasets and only non-monotonic logical reasoning methods. This chapter builds on that foundational architecture and extends it to address partial observability, task ambiguity, and scale to multiple ad hoc agents. We begin by introducing the third component of KAT: *task anticipation*, which enables the ad hoc agent to predict high-level future tasks in the domain (Section 5.1). Next we extend our architecture to support multiple ad hoc agents collaborating together and evaluate the scalability of the architecture where coordination is more challenging (Section 5.2). Finally, we explore KAT’s performance under partial observability with/without limited communication capabilities (Section 5.3). Each section also includes a detailed evaluation of the functionality it introduces, presenting their experimental setup and results.

5.1 Task Anticipation

In practical domains, agents can encounter multiple possible goals, creating uncertainty on which task to prioritize. For example, in the human-AI agent collaboration scenario described in Example 6, the agents can pursue a range of tasks such as “make breakfast”, “set up a workstation” and “clean dishes”. While the human agent may be aware of these upcoming tasks, the ad hoc agent by itself is only aware of the current task at each step. Thus, KAT enables the ad hoc agent to use the generic knowledge of similar situations in a pretrained Large Language Model (LLM) to anticipate the high-level future tasks in the domain, revising and adapting the LLM’s output based on domain-specific knowledge and experience.

Note that task anticipation here is related to but distinct from the well-established research area of goal recognition (Keren et al., 2020; Meneguzzi and Fraga Pereira, 2021). In classical goal recognition, the objective is to infer an agent’s current (hidden) goal or intentions from a partial sequence of observed actions (behavior). However, our task anticipation component assumes the current task is known, and instead predicts the **next (future)**, not-yet-assigned tasks that are likely to be assigned to the agent team once the current task is completed. Also we draw on generic knowledge encoded in a pretrained LLM rather than inferring over observed action traces.

Frameworks based on “foundation” models have indicated impressive capabilities in solving various problems in robotics and AI, but there is evidence to show that they (by themselves) are not a good choice for planning in practical domains as they can make arbitrary decisions in novel situations (Lu et al., 2024). They have shown to be more effective when used as translators between natural and domain-specific languages (Guan et al., 2023), and to generate high-level (i.e., generic or abstract), guidance that is validated externally before being implemented by suitable planning subroutines (Kambhampati et al., 2024).

Motivated by these findings, KAT incorporates the third component, *task anticipator* to enable an ad hoc agent to use an LLM to anticipate the high-level future tasks (e.g., *prepare dinner*) that are likely to be assigned once the current task is done. Recall that in the absence of the LLM, the agents are informed about the tasks to be executed one at a time. When the LLM is included in the architecture, each ad hoc agent anticipates the task likely to be assigned once the current task is done. By considering the current and anticipated tasks as joint goals, the ad hoc agent can prepare for upcoming tasks while completing the current task, e.g., fetch some ingredients from the fridge for

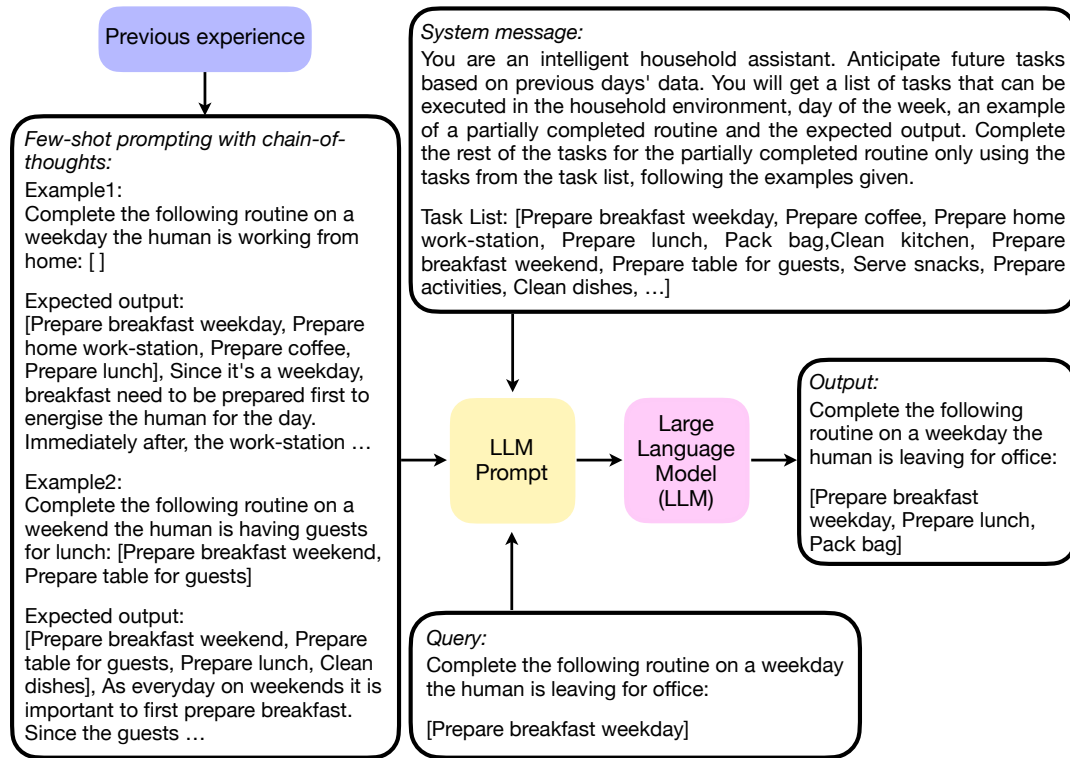


Figure 5.1: Example prompt sent to the LLM to anticipate the human's future tasks. Each prompt includes a list of possible tasks, two partially completed example routines, and a partially completed routine for the LLM to complete.

making lunch while fetching eggs for cooking breakfast. We experimentally demonstrate in Section 5.1.1 that this strategy of targeting joint goals improves the team's performance.

Figure 5.1 shows an example input prompt to the LLM by the ad hoc agent in the VirtualHome domain. The ad hoc agent uses a combination of three prompting strategies to interact with the LLM:

1. **Adopting persona:** A specific role or character is assigned to guide the LLM's responses to be more (contextually) consistent with the assigned role.
2. **Few-shot prompting:** The prompt includes a few examples of the expected output in specific situations, guiding the use of pretrained knowledge.
3. **Chain-of-thought (CoT):** The prompt includes a step-by-step reasoning process that can be followed to arrive at an answer, leading to more accurate responses.

The prompt comprises both predefined and dynamically generated components. The 'system message' (Figure 5.1) is predefined and used to guide the LLM to adopt the

persona of a household assistant as the two agents are collaborating in a household environment. It also instructs the LLM about the current objective, *i.e.*, to complete the partially completed routine by selecting potential future tasks from the provided list of tasks within the VirtualHome domain. The ‘few-shot’ prompting approach is used to include two task routines that are automatically selected at random from previous days’ data in the prompt. These routines can be at different stages of completion, *e.g.*, one with no completed tasks and another with a partially completed routine. Next, CoT prompting is used to explain the reasoning behind each task in the few shot examples. For example, in Figure 5.1, a sample routine like ‘prepare breakfast weekday, prepare home workstation, prepare coffee, and prepare lunch’ is explained with CoT method as ‘Since it is a weekday, breakfast needs to be prepared first to get the human ready for the day, after which the workstation should be set up to ensure the human can be productive during the day ...’. These explanations can be manually provided by the system designer or generated using the LLM itself. Finally, the system message, few-shot examples with CoT explanation, and the current query (*i.e.*, partially completed or empty task routine for the given day) are combined and provided as the input prompt to the LLM. Examples of the prompts generated and the corresponding LLM outputs can be found in the Appendix A.3.

Throughout this thesis, $LLM(P)$ denotes a single inference call to the GPT-4o mini language model, where P is the complete prompt constructed for the task. The prompt may comprise any combination of system messages, few-shot examples, retrieved context, user queries, or other task-specific instructions. The function returns the textual response generated by the model:

$$r = LLM(P) \quad (5.1)$$

In above $P = \text{system message} + \text{few short examples} + \text{query}$, and r is the output in response to the prompt. When the LLM provides the output response, an *external validator* parses the LLM’s output to check whether the tasks are feasible and in a reasonable order. Since LLMs are trained with large volumes of text from many domains, the anticipated tasks may not be feasible or may not match the actual needs in our domain. Thus the validator uses domain-specific knowledge (and task priorities or preferences, if any) from previous experience (*e.g.*, recent observations) to revise and reorder these tasks. Specifically it filters historical data using domain-relevant contextual features (*e.g.*, preparing lunch is not necessary if the human is eating out) and compares the filtered data with LLM’s output of expected tasks. This helps the ad hoc

agent eliminate tasks that are invalid or irrelevant, and reorder tasks according to the human preferences, e.g., in Section 5.1.2 the ad hoc agent prioritizes preparation of the workstation for work over preparing coffee. The validator is intentionally simple, making only the minimal modifications necessary. These validated high-level anticipated tasks then serve as joint goals to be achieved. Since this list of validated tasks can change over time, KAT enables the ad hoc agent to consider one anticipated (future) task along with the current task, along with predicted actions of teammates over the next few steps (using the learned behavior models), to plan a sequence of actions to be executed. We show in Section 5.1.2 that performance is much better with the LLM-based anticipation than without it.

In the following sections, we describe the experimental procedure used to evaluate the current architecture (i.e., *base architecture and task anticipation*; Section 5.1.1) and present the results (Section 5.1.2).

5.1.1 Experimental Procedure

In our experiments, we used KAT’s core architecture described in Chapter 4 with GPT4o mini (OpenAI and team, 2024) for high-level task anticipation. We used the LLM with default parameters (temperature = 1.0) without any domain-specific fine tuning. While the use of online LLMs such as GPT4o mini in practical deployments may raise considerations related to internet connectivity, latency, and privacy, our architecture queries the LLM only a limited number of times per episode for high-level task anticipation rather than at every step, making the latency introduced by API calls minimal. The average cost per LLM call remained low (i.e., approximately \$0.000042 for the input (approximately 394 tokens) and \$0.000016 for the output (approximately 31 tokens)). Furthermore, the online LLM could be easily substituted by a locally hosted LLM in settings where connectivity or privacy is a concern without requiring any changes to the rest of the architecture.

We conducted experiments in the VirtualHome domain (Section 4.7.1) with teammate policies selected from Section 4.7.2. All agents received the same observations of the domain at each step, which they used to plan their respective actions. There was no direct communication between them. The task generator provided a sequence of tasks for any given day. These sequences involved realistic, complex household tasks such as *prepare breakfast*, *prepare home-workstation*, *make coffee*, and *prepare lunch*. Completing each task required multiple interdependent actions involving finding and

retrieving objects from specific locations; placing objects in target locations; interacting with appliances by opening or closing them; and turning appliances on or off for cooking, cleaning, and other purposes. The sequence of tasks generated by the task generator were assigned to the agents over time based on the successful completion of the previous task; the agents were not aware of the complete sequence and received one task at a time.

When the ad hoc agent received a task, it followed the procedure described in Section 5.1 and Figure 5.1 to: (1) automatically generate the prompt; (2) query the LLM to anticipate the high-level future tasks likely to be assigned once the current task was completed; and (3) validate the LLM anticipated tasks by filtering out invalid or irrelevant tasks and reordering them (see Section 5.1.3 and Appendix A.3 for execution examples). The validated tasks were then mapped to Answer Set Prolog literals (goals), with the current task and the next anticipated task set as joint goals for the ad hoc agent. For example, consider a scenario where the current task is *prepare breakfast*. This task maps to the following Answer Set Prolog goal literals (as described in Example 7):

$$\begin{aligned} \text{goal}(I) : & \text{--holds}(\text{heated}(\text{poundcake}), I), \text{holds}(\text{on}(\text{poundcake}, \text{kitchentable}), I), \\ & \text{holds}(\text{cooked}(\text{cutlets}), I), \text{holds}(\text{on}(\text{cutlets}, \text{kitchentable}), I), \\ & \text{holds}(\text{on}(\text{waterglass}, \text{kitchentable}), I). \end{aligned}$$

The task anticipated by the task anticipation component in this scenario as the next task is *prepare workstation*. This task maps to the following goal literals:

$$\begin{aligned} \text{goal}(I) : & \text{--holds}(\text{switchedon}(\text{computer}), I), \\ & \text{holds}(\text{on}(\text{book}, \text{livingroom_desk}), I), \\ & \text{holds}(\text{on}(\text{cellphone}, \text{livingroom_desk}), I). \end{aligned}$$

The ad hoc agent then combines these two goals to form the following joint goal:

$$\begin{aligned} \text{goal}(I) : & \text{--holds}(\text{heated}(\text{poundcake}), I), \text{holds}(\text{on}(\text{poundcake}, \text{kitchentable}), I), \\ & \text{holds}(\text{cooked}(\text{cutlets}), I), \text{holds}(\text{on}(\text{cutlets}, \text{kitchentable}), I), \\ & \text{holds}(\text{on}(\text{waterglass}, \text{kitchentable}), I), \\ & \text{holds}(\text{switchedon}(\text{computer}), I), \\ & \text{holds}(\text{on}(\text{book}, \text{livingroom_desk}), I), \\ & \text{holds}(\text{on}(\text{cellphone}, \text{livingroom_desk}), I). \end{aligned}$$

Once the goals were set the agent proceeded to planning and followed the pipeline described in Algorithms 4 and 5. During planning, the ad hoc agent used the learned behavior prediction models to predict each teammate’s actions for two future steps (Section 4.4). The initial version of these predictive models were built using 1000 examples of prior traces of actions and domain state and revised on the fly based on observations. As described in Section 4.6, the predicted actions from the models were then mapped to exogenous actions that were added to the Answer Set Prolog program along with initial state information and refined sorts. The Answer Set Prolog program of the ad hoc agent included additional axioms for reasoning about these predicted actions of each teammate. As a result, the ad hoc agent’s plan sometimes anticipated preconditions of some intermediate steps to be satisfied by a teammate’s actions, even though the teammate did not always execute that action. The ad hoc agent hence had to respond to unexpected action outcomes and domain states. As described in Section 4.7.2, the human’s actions were determined primarily by the current state with the current task as the goal.

For **Exp1**, we designed six distinct daily task routines representing realistic household activities that the agents can collaborate to complete as a team. Since the VirtualHome domain does not provide predefined tasks, we first defined a set of individual tasks based on the available domain actions. Examples of these tasks include *Prepare breakfast weekday*, *Prepare home work-station*, and *Prepare coffee* (Example 6). We then combined these tasks into six distinct daily routines that represent realistic sequences of household activities. For example, one routine consisted of *Prepare breakfast weekday*, *Prepare home work-station*, *Prepare coffee*, *Prepare lunch*, representing a weekday when the human is working from home, and another consisted of *Prepare breakfast weekend*, *Prepare table for guests*, *Prepare lunch*, *Clean dishes*, representing a weekend when guests are expected. We then generated 100 task sequences by sampling from these predefined routines and measured the ability of a team comprising a human agent and an ad hoc agent (following the procedure described above) to complete these tasks. **Performance measures** were the number of steps and time taken to successfully complete all the tasks assigned for a given day. As the baselines for **Exp1** we used:

- **Base1**: which used LLM for anticipating future tasks, but did not use the behavior models to predict human’s actions.
- **Base2**: which did not use the LLM to anticipate future tasks, but used behavior

models to predict the human's actions.

- **Base3**: which did not use the LLM for task anticipation or behavior models to predict the human's actions.
- **Base4**: which used the LLM for directly computing sequences of actions for specific tasks.

This choices of baselines was motivated by the fact that we could not find any existing framework for AHT that supported all the capabilities of our architecture that we wanted to evaluate. This choice also allowed us to conduct **ablation** studies that experimentally evaluated the contribution of each key component of KAT. We also compared with a purely knowledge-based baseline (**Base 3**) and a purely data-driven baseline (**Base 4**). We created **Base4** such that it uses the LLM to directly generate a sequences of actions to complete a given task. In this setup, the prompt to the LLM constructed by the ad hoc agent consisted of the following:

- System message guiding the LLM to adopt the persona of a household assistant and instructing it to generate the next action to execute for achieving the assigned goal.
- Details of the available actions in the VirtualHome domain and their intended purpose (e.g., *move(agent, location)*: move the agent to an adjacent location; *put(agent, object, surface)*: place an object on a surface).
- Action Feasibility Rules:
 1. *Movement Limitation* (Critical): You must only move to adjacent locations. Adjacent locations are explicitly defined using the *next_to* relationships. If the destination is not adjacent, the move action is prohibited. Always check adjacency before predicting a move.
 2. *Object Location*: You must be in the same location as an object to perform any action on it (e.g., grab or put).
 3. *Carrying Limit*: You cannot hold more than two objects at a time. When holding two objects, actions like open, close, switchon, or switchoff require putting down at least one object first.
 4. *Appliance Safety*: For safety, you should not open appliance doors when they are switched on.

5. *Avoid Conflict*: If the human is holding an object, assume they will handle the tasks involving that object. You must not attempt to grab, or interact with the object in the humans possession. Instead, focus on completing other parts of the goal.

- Information about adjacent locations in the domain (e.g., *next_to(counter, kitchentable)*), emphasizing the fact that the agent can only move between the defined adjacent places.
- Current world state (location of the agents, objects, and appliances, each appliance’s state, information about the objects held by each agent).
- Description on current task assigned to the agent by the task generator.
- Immediate past actions of all the agents including the ad hoc agent.
- Any contextual information about the day (e.g., weekday or not; human working from home; guests are expected).
- An illustrative example of action selection.
- Query asking the LLM to predict the next action required to achieve the task.

The LLM’s action choice was assigned as the **Base4** ad hoc agent’s action. As a recovery mechanism, we corrected errors in the LLM output up to three times per trial. For example, if the LLM’s action involved grabbing an object without moving to the appropriate location, we provided feedback explaining why this choice was incorrect and allowed the LLM to predict another action for that step. We then measured the number of steps and time taken by the agent team to complete the previously selected 100 task routines.

One key consideration of our evaluation was to measure how well the ad hoc agent anticipated the correct next task, as the tasks in our domain follow realistic daily household routines with natural dependencies and priorities (e.g., prepare breakfast before lunch, set up the workstation before preparing coffee on a workday). Thus, we did not use a random task generator as a baseline since it would produce unrealistic and sometimes infeasible task sequences and would not meaningfully test the agent’s ability to anticipate the correct next task.

Since the actual time taken and the number of actions required to complete tasks can vary substantially based on the task, the average of these values over the individual

trials may not be meaningful. We instead *ran paired trials and computed performance measure values for the baselines as a fraction of these values for KAT in each trial*. We then reported the average of these ratios.

Next, for evaluating the importance of the validator, in **Exp2**, we computed the precision and recall of the tasks anticipated by the LLM in *Exp1* (over the selected 100 task routines), before and after applying the validator. We also computed the precision and recall of a simple statistical model that replaced the LLM and anticipated the agent’s next task based on past experience. Note that this statistical model serves a similar purpose to a random baseline, representing a naive approach to task anticipation based purely on historical frequency without any reasoning, and thus provides a meaningful lower bound than pure randomness.

Further, for evaluating the impact of different prompt engineering techniques and the validator on the LLM’s task anticipation and team’s overall performance, in **Exp3**, we randomly selected 20 task routines and repeated the procedure from *Exp1* with and without the combination of prompting methods described before. This allowed us to assess how each of these techniques influenced the effectiveness of task prediction and coordination. We used four baselines:

- **Base5**: none of the prompting strategies or validator was used.
- **Base6**: used few-shot prompting (i.e., two examples with expected answers) without an external validator.
- **Base7**: used CoT prompting without an external validator.
- **Base8**: used external validator but no prompt-engineering techniques were used.

As the **performance measures** we measured the number of steps and time taken for the human-agent team to complete the tasks for the day. The number of steps provided a direct measure of the efficiency of the team’s collaboration, i.e., a team that coordinates well will require fewer actions to complete the same set of tasks. The total time taken complements this measure, as two plans with the same number of steps could take different amounts of time depending on the distances agents need to travel in the environment. Note that the total time here corresponds to the time taken to execute the selected actions in the VirtualHome domain and does not include planning or other computation times. These measures also captures the benefits of joint planning across tasks. For example, an agent that picks up objects relevant to a future task on its route while completing the current task saves both steps and time that would otherwise be

spent returning to those locations later. Thus, these measures reflect not only the efficiency of individual actions of the ad hoc agent but also its ability to plan ahead and coordinate effectively with its teammate.

In the next sections we present the results of these experiments. We first present the quantitative results (Section 5.1.2) followed by some qualitative results.

5.1.2 Quantitative Results

Table 5.1 summarizes the results of **Exp1**. When the ad hoc agent reasoned with anticipated tasks from the LLM and predicted human agent actions using the behavior models, it provided the best observed performance with lowest number of action steps and least amount of time taken to complete task routines. When the ad hoc agent used task anticipation but did not use the behavior prediction models (**Base1**), the number of steps and time taken to complete tasks increased as the inability to predict the actions of teammates may lead the ad hoc agent to execute the same actions as a teammate, hindering collaboration. These results emphasize the importance of using the behavior prediction models.

Next, when the ad hoc agent used the behavior models to predict the human agent's actions, but did not anticipate and plan for future tasks (**Base2**), the performance worsened, with a further increase in the number of steps and the time taken to complete the tasks. Planning actions jointly for the current task and the anticipated (next) task often saves time and effort. For example, when the agent visited the bedroom to retrieve a board game for the guests, it also picked up snacks from the kitchen on the way. Making two separate trips for these tasks extended the length and duration of the plans. These results indicate the impact of planning for joint goals has on performance. Also, when the ad hoc agent did not use task anticipation or the behavior prediction models (**Base3**), the performance worsened further. Finally, **Base4** which used the LLM to directly anticipate the actions of the ad hoc agent showed the worst observed performance with the highest number of steps and time taken to complete the tasks. These results indicate that reasoning with prior knowledge and predictive models improves team performance and using LLM-based anticipated tasks as joint goals improves performance compared to doing one task at a time. It further suggests that using LLMs to predict a sequence of low-level actions to complete tasks results in poor performance. Table 5.2 shows the results from **Exp2**, where we computed the precision and recall values of the tasks anticipated by the LLM before and after applying the validator. We

<i>Architecture</i>	<i>Steps</i>	<i>Time</i>	<i>Steps (raw)</i>	<i>Time (raw)</i>
KAT (anticipate tasks, predict actions)	1.0	1.0	26.8	361.0
Base1 (anticipate tasks)	1.1	1.1	29.3	385.8
Base2 (predict actions)	1.3	1.2	34.1	443.0
Base3	1.4	1.4	37.5	487.4
Base4 (LLM predict low-level actions)	1.5	1.5	39.7	520.0

Table 5.1: Average number of steps and time taken by the team (human, ad hoc agent) to complete task routines, with values for baselines computed as a fraction of these values for KAT in each trial; for comparison, the average absolute values are presented alongside. Task anticipation (LLM) and action prediction (FF trees) substantially improve performance.

<i>Architecture</i>	<i>Precision</i>	<i>Recall</i>
Simple statistical model	0.75	0.76
LLM without validator	0.78	0.76
LLM with validator	0.99	0.78

Table 5.2: Precision and recall values of the anticipated tasks with and without the LLM and the validator. A simple statistical model is not good enough and the validator plays an important role.

observed marked improvement in precision after applying the validator to the LLM’s output. The errors in the LLM’s outputs were substantially reduced by the validator. The recall values did not change substantially as the validator did not introduce new tasks; it only reordered the tasks that are out of order and removed the irrelevant tasks. On the other hand, the simple statistical model that anticipated future tasks based only on past experience resulted in precision and recall of 0.75 and 0.76 respectively. These results indicate that using just the historical data is insufficient for task anticipation, and using the validator to correct the LLM’s output based on prior knowledge and experiences leads to better results. Table 5.3 shows the results from **Exp3**, which explored the use of different prompting strategies and the validator with the LLM to anticipate future tasks. We again observed a significant improvement in performance, e.g., a lower number of steps and time to complete tasks with the external validator and a combination of prompting methods. In particular, using the validator to adapt

<i>Architecture</i>	<i>Steps</i>	<i>Time</i>	<i>Steps (raw)</i>	<i>Time (raw)</i>
KAT (all prompting, with validator)	1.00	1.00	27.5	372.7
Base5 (no prompting, no validator)	1.21	1.15	33.1	427.7
Base6 (few-shot prompting, no validator)	1.17	1.18	32.2	441.1
Base7 (chain-of-thoughts, no validator)	1.16	1.16	31.7	430.8
Base8 (no prompting, with validator)	1.05	1.04	28.9	387.5

Table 5.3: Average number of steps and time taken by the team (human, ad hoc agent) to complete tasks with prompting strategies and/or external validator; performance measure values for baselines computed as a fraction of values for KAT in each trial, with the average absolute values presented alongside.

the LLM’s output to the domain had a significant impact on performance. These results indicate that using prompt engineering techniques, incrementally updated prompts and validators improve LLM task anticipation and team’s performance.

5.1.3 Qualitative Results

This section provides some execution traces as a qualitative evaluation of the capabilities described in Section 5.1.

First, Figure 5.2 shows an execution example where the ad hoc agent used the LLM to anticipate future tasks with and without the prompting strategies and validator. The example was set on a weekday where the human was working from home and no guests were expected. The correct task routine in this context was: *Prepare breakfast, Prepare home work-station, Prepare coffee, Prepare lunch*. When the ad hoc agent queried the LLM without the prompt engineering techniques or validator (Section 5.1), the anticipated task list was different from the expected output. The prompt to the LLM without using the prompt engineering strategies is shown in Figure 5.2. The LLM output was [*Prepare breakfast, Prepare coffee, Prepare home work-station, Pack bag*]. This output failed to align with the human preferences and priorities as:

- Higher priority was assigned to making coffee than setting up the workstation. This would delay the human for work and lead to coffee not being hot when needed.
- Packing the bag was an unnecessary task as the human was not leaving the house,

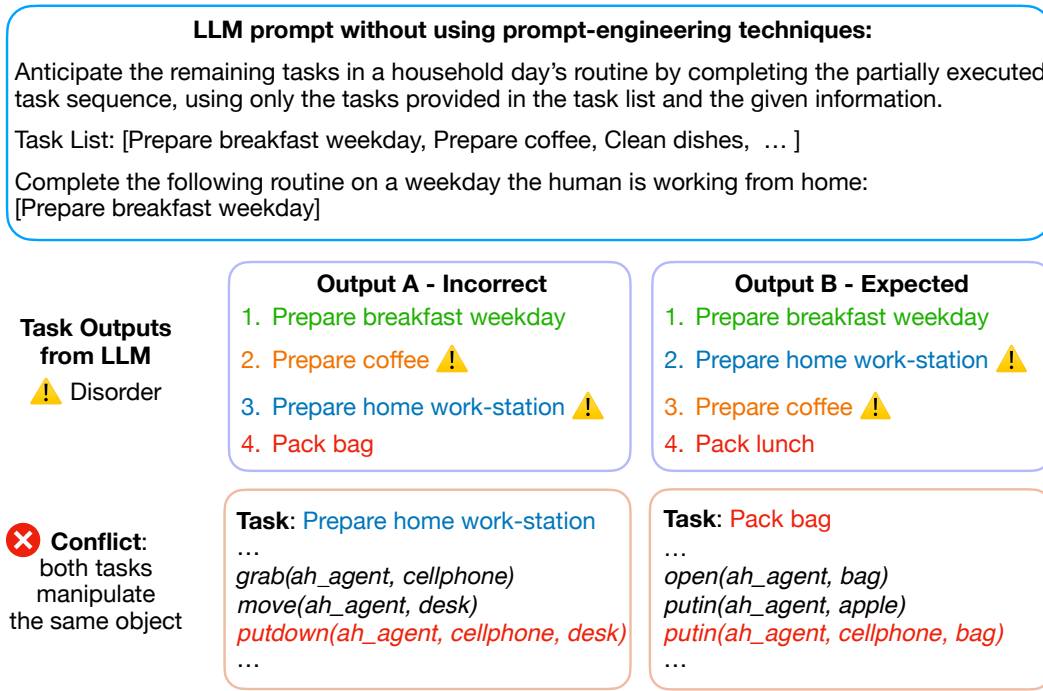


Figure 5.2: Execution example: using LLM without prompting strategies or external validator causes conflicts during execution, having a negative impact on performance.

and would have been filtered out by the validator.

On the other hand, when the ad hoc agent used the prompt engineering strategies and the validation strategy, the prompt to the LLM was automatically generated while incorporating context, as described in Section 5.1. The LLM's output was [Prepare breakfast, Prepare home work-station, Prepare coffee, Prepare lunch]. This matched the expected routine. i.e., making breakfast and setting up the workstation were considered high priority tasks, and irrelevant tasks such as *pack bag* were filtered out by the validator. These results indicate the importance of using a combination of prompting techniques and the external validator.

We observed similar situations when extending the setup to three agents, one human and two ad hoc agents, collaborating on a weekend when guests were expected. The correct task routine in this context was: *Prepare breakfast, Prepare table for guests, Prepare lunch, Clean dishes*. When the first ad hoc agent queried the LLM with the prompting strategies but without the validator, the anticipated task list by the LLM was *Prepare breakfast, Prepare table for guests, Prepare lunch, Serve snacks*. For the second ad hoc agent the LLM output was *Prepare breakfast, Prepare table for*

With predictive behaviour models - Steps: 28	Without predictive behaviour models - Steps: 34
... human [grab] cupcake agent [grab] boardgame ... human [move_to] coffeetable agent [move_to] desk2 human [grab] candybar agent [move_to] tvstand ... human [move_to] kitchentable agent [grab] cutlets ... human [move_to] tvstand agent [put] milk on microwave human [move_to] desk2 agent [move_to] desk1 human [put] cereal on desk2 agent [put] cutlets on desk1	... human [grab] cupcake agent [grab] cupcake ... human [move] coffeetable agent [move_to] coffeetable human [grab] candybar agent [grab] candybar ... human [grab] cutlets agent [grab] cutlets ... agent [put] cutlets on desk1 agent [move_to] desk2 agent [put] cereal on desk2

Figure 5.3: Execution trace for task routine: *[Prepare breakfast, Prepare activities, Serve snacks, Clean kitchen]*. When an ad hoc agent is not allowed to predict and reason about the human’s actions, it may choose to execute same action(s) as the human, leading to longer plans.

guests, Prepare lunch, Serve snacks, Clean dishes. When the validator was used, the outputs to both agents were refined by incorporating context. Since the human usually did not require snacks after lunch, the task *Serve snacks* was removed. For the first agent the refined task list was *Prepare breakfast, Prepare table for guests, Prepare lunch*. For the second agent the task list was *Prepare breakfast, Prepare table for guests, Prepare lunch, Clean dishes*; since *Clean dishes* was a valid task, it was retained by the validator. This example further suggests the importance of using the validator.

Additionally, Figure 5.3 compares two plans executed by a team comprising a human and an ad hoc agent for completing a different set of tasks: *[Prepare breakfast, Prepare activities, Serve snacks, Clean kitchen]*, with and without the behavior prediction models (Section 4.4). In the first plan, when the ad hoc agent used the behavior prediction model to predict the future actions of the human, the team successfully completed all tasks for the given day in just 28 steps. On the other hand, when the ad hoc agent did not use behavior prediction models, it often selected the same actions as the human for any particular task, leading to unnecessary delays in completing the tasks. For example, in the second plan the agent frequently selected the same action as the human—simultaneously picking up the cupcake, candy bar, and cutlets, introducing redundant behavior and prolonging task execution. As a result, the overall plan was extended to 34 steps. These results indicate that using the behavior prediction models enables the ad hoc agent to coordinate efficiently by avoiding action conflicts with the

In the current state, the human is holding both cereal and bananas, which means they cannot handle any additional objects. Since the goal involves switching on the computer and placing the note book and cellphone on the living room desk, I need to focus on actions that can help achieve this goal. The computer is located at the living room desk, and I cannot carry any items for the human while the human is holding items.

Since there's no action feasible for me to grab or put anything while the human is busy, one action I can perform immediately is to move to the living room desk to prepare for placing the items once the human is free.

I will predict the next action based on this reasoning. Next action: *move(agent, living room desk)*

Figure 5.4: Execution example: using LLM to generate low-level actions results in poor performance.

human.

When the ad hoc agent used the LLM to directly output action sequences for specific tasks (**Base8**), prompts were constructed as described in Section 5.2.2. Figure 5.4 shows part of the LLM output during the task routine: [*Prepare breakfast, Prepare home work-station, Prepare coffee, Prepare lunch*]. The action output from the LLM *move(agent, living room desk)* violated the ‘Movement Limitation (Critical)’ rule in ‘Action Feasibility Rules’ (Section 5.2.2), which constrained the agent to only move to locations adjacent to its current location. This example indicates that the LLM may not respect constraints even when they are provided as input, and highlights that an LLM is not designed for computing plans for non-trivial tasks; thus using an LLM to directly output a sequence of low-level actions to complete tasks can lead to poor performance. The exact prompts provided to the LLM, together with the corresponding outputs for the examples discussed above are presented in Appendix A.3.

Having explored how KAT leverages the generic predictive capabilities of an LLM to anticipate the future tasks in the domain, enabling joint planning and improved collaborative performance, in the next section, we examine how KAT supports scalability and extends to domains with multiple ad hoc agents.

5.2 Scalability

As the number of agents in a team increase, the complexity of decision-making also increases, making it difficult for the agents to collaborate effectively. Thus, KAT use refinement when reasoning with prior knowledge and revision of predictive behavior models to aid scalability. As described in Section 4.5, KAT supports dynamic revision of FF tree models using limited data, enabling rapid adaptation to new teammates. The agent only maintains a small library of possible teammate types and supports fast re-

vision of those models facilitating scalable collaboration. Moreover, we build on prior work in our group on a refinement-based architecture (Sridharan et al., 2019) to enable the ad hoc agent to represent and reason at different resolutions. Specifically, as described in Section 4.3 and Section 4.6, the ad hoc agent automatically selects and uses the goal and part of the domain description relevant to tasks at hand (i.e., only consider the objects and locations relevant to the current goal, without needing to consider all the location and objects in the domain). This significantly reduces inference complexity, enabling faster decision-making, and the architectures capability to scale to larger teams and domains. In the following sections, we describe the experimental setup for evaluating KATs scalability and discuss its results.

5.2.1 Experimental Procedure

To evaluate KAT’s scalability beyond a single ad hoc agent setting, we conducted a series of experiments that gradually increased the team size by introducing additional ad hoc agents from one (one ad hoc agents, one human) to three (three ad hoc agents, one human). Specifically, we repeated **Exp1** from Section 5.1.1 while increasing the number of ad hoc agents. We specifically introduced additional ‘ad hoc agents’ who adapt on the fly making adaptation and collaboration more challenging. Recall that with just two agents (one ad hoc agent, one human) the domain has $\approx 10^{25}$ possible states, and this number increases exponentially with the number of AI agents. Each ad hoc agent learned the initial behavior models for its teammates with 1000 data samples. At the beginning, each agent only had the behavior model it had previously learned for the human agent. It revised this model on the fly using current observations to predict the actions of the other agents thus adapting and improving its predictions during interaction. We conducted 100 trials for each team by randomly sampling the task sequences from the six predefined routines.

As the **performance measures** we recorded the number of steps and time taken by the agent teams to complete the tasks. Again we computed this as a fraction of the values from Team1 since the number of action steps required to complete tasks can vary substantially based on the tasks under consideration.

5.2.2 Results

Table 5.4 summarizes the performance of **Team1** (human, one ad hoc agent), **Team2** (human, two ad hoc agents) and **Team3** (human, three ad hoc agents) in completing

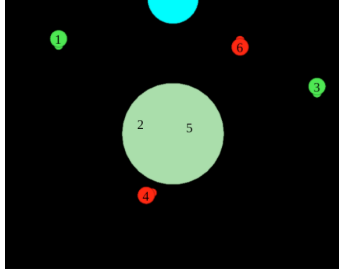
<i>Team</i>	<i>Steps</i>	<i>Time</i>	<i>Steps (raw)</i>	<i>Time (raw)</i>
Team1 (human, ad hoc agent)	1.0	1.0	26.8	361.0
Team2 (human, two ad hoc agents)	0.8	0.9	22.8	329.4
Team3 (human, three ad hoc agents)	0.7	0.8	19.5	304.8

Table 5.4: Average number of steps and time taken by Team1, Team2 and Team3 to complete the task routines, with performance measure values for Teams 2-3 computed as a fraction of the values for Team 1 in each trial, showing average absolute values alongside; collaboration between agents improves performance.

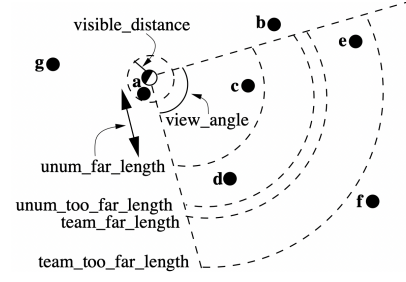
the same set of 100 task routines. As the number of ad hoc agents increases, task completion becomes more efficient: Team2 outperformed Team1 by requiring fewer steps and less time to complete the tasks, while Team3 showed further improvements over Team2. Note that as the number of agents increases, the size of the joint state space also increases exponentially, which would normally be expected to make coordination harder and planning more expensive. However, these results show that task completion improves rather than worsens with team size, showing that having more agents available to jointly work towards current task, together with the design choices in our architecture to enable each ad hoc agent to reason independently and efficiently, using refinement-based reasoning to restrict reasoning to only the part of the domain relevant, and using simple learned models improves performance.

Moreover, the average Answer Set Prolog planning time per step for the ad hoc agent, computed over 10 games (approximately 300-350 steps in total) for each team size were, 0.44 for Team1, 0.49 for Team2 and 0.45 for Team3. The planning time remained largely stable as team size increased, rather than increasing substantially with the number of agents being reasoned about. These results support our claim that refinement-based reasoning contributes to the scalability of the architecture, emphasize the importance of efficient collaboration among agents and indicate that KAT supports scalability to multiple agents.

In the next section, we explore how KAT performs under partial observability, especially how its reasoning and collaboration capabilities are affected when agents make decisions based on incomplete information.



(a) Fort Attack; Attackers inside the forest region are hidden from the view of other guards except the ad hoc agent.



(b) Half Field Offense; Visible range of the ad hoc agent (Stone, 2000).

Figure 5.5: Screenshot from the Fort Attack and Half Field Offense domains after introducing partial observability.

5.3 Partial Observability and Communication

In practical AHT domains, any single agent cannot observe the entire domain and communication is a scarce resource. In Chapter 4 we evaluated our architecture under full observability. In this section we extend the original Fort Attack and Half Field Offense domains to support partial observability and explore KAT's capabilities under partial observability with limited communication.

Specifically, in the Fort Attack domain, we introduced a *forest* region where attackers can hide from the view of the two guards other than the ad hoc agent, giving them an opportunity to secretly approach the fort (Figure 5.5). The ad hoc agent has visibility of the forest region and decides when to communicate with its teammates, e.g., when: (a) one or more attackers are hidden in the forest; and (b) one of the other guards is closer to the hidden attacker(s) than the ad hoc agent (attacker is within the shooting range of the other guard). The associated reasoning can be encoded using statements such as:

$$\text{holds}(\text{in_forest}(A), I) \leftarrow \text{holds}(\text{agent_loc}(A, X, Y), I), \text{forest}(X, Y), \quad (5.2a)$$

$$\text{not holds}(\text{shot}(A), I)$$

$$\neg \text{occurs}(\text{communicate}(AHA, G, A), I) \leftarrow \text{not holds}(\text{in_range}(G, A), I). \quad (5.2b)$$

$$\text{holds}(\text{shoots}(G, A), I + 1) \leftarrow \text{occurs}(\text{communicate}(AHA, G, A), I) \quad (5.2c)$$

where Statement 5.2(a) defines when an attacker is hidden inside the forest; Statement 5.2(b) encodes that communication is used only when a hidden attacker is within the range of a teammate (i.e., guard agent); and Statement 5.2(c) describes the ad hoc

agent's belief that a teammate receiving information about a hidden attacker will shoot it, although the teammate acts independently and may choose to ignore this information. In addition to the above, the domain description also includes axioms that calculate the distance between guards (including ad hoc agent) and the attackers. These axioms are used to choose the guard that is nearest to the attacker. If there are multiple guards satisfying these conditions, the ad hoc agent may only communicate with the guard closest to the hidden attacker(s) if possible. If not then the ad hoc agent will broadcast this information to all the guards.

Since there is no guarantee that an agent receiving this information will actually shoot the attacker, as each agent acts independently and may choose to ignore the communicated information, we use the agents' predictive behavior model to estimate whether the selected guard agent is likely to take the shot. If the model predicts that the guard will not take an action, we rerun the Answer Set Prolog planner by disabling the communication actions and take action to eliminate the attacker and protect the fort.

Next, in the Half Field Offense domain, we represent partial observability using the built-in ability to limit each agent's perception to a specific distance (3.0; total pitch size = $105 * 68$) and viewing cone ($90deg$) relative to the agent. Specifically, each agent is only able to sense objects (e.g., other agents, ball) within its viewing cone; objects outside its viewing cone are not visible. The ad hoc agent only reasoned with visible objects; and no additional communication actions were implemented.

5.3.1 Experimental Procedure

This section details the experimental procedure used to evaluate KAT architecture under partial observable settings.

Fort Attack

Following the same experimental setup from Section 4.7 in the Fort Attack domain, each game episode had three guards and three attackers, with our ad hoc agent replacing one of the guard agents. We then evaluated our KAT architecture using two experimental setups:

- **Exp1**, in which other agents followed the hand-crafted policies (Section 4.7.2); and
- **Exp2**, in which other agents followed the built-in policies (Section 4.7.2).

As the baselines for our experiments, we used two baselines where:

- **Base1:** in **Exp1**, other agents followed hand-crafted policies and ad hoc agent did not use any communication actions.
- **Base2:** in **Exp2**, other agents followed built-in policies and the ad hoc agent did not use any communication actions.

For each experiment, we performed 150 trial episodes and recorded the win percentage of the guards team as the key **performance measure**.

Half Field Offense

In the Half Field Offense domain, we followed the same experimental setup as Section 4.7 and we used the six external agent teams, *Helios*, *Gliders*, *Cyrus*, *Axiom* and *AUT* as the offense team; and *Agent2d* team as the defense team (Section 4.7.2). We used two versions of the game:

- **Exp3**, limited version, i.e., two offense players (including ad hoc agent) against two defense agents (including goalkeeper); and
- **Exp4**, full version, i.e., four offense players (including ad hoc agent) played against five defense agents (including goalkeeper).

All offense agents other than the ad hoc agent were based on one randomly selected external team in each episode. As the baselines, we used the six external agent teams. We conducted 1000 episodes for each experiment.

The key **performance measure** was the fraction of games in which the offense team scored a goal.

5.3.2 Results

Below, we present the results of our experiments across the Fort Attack and Half Field Offense domains, highlighting the ad hoc agent’s ability to collaborate under conditions of limited observability.

Fort Attack

Table 5.5 summarizes the results from **Exp1**, where the agents collaborated under partial observability, with and without communication (**Base1**). The agents other than

the ad hoc agent used the Fort Attack domain’s hand-crafted policies in this experiment. When the communication actions were enabled for the ad hoc (guard) agent, the winning percentage of the team of guards was substantially higher than the winning percentage of the team of guards when they could not use the communication actions. **Policy2** was a particularly challenging scenario as both guards and attackers can shoot, which justified the lower (overall) winning percentage.

Next, the results from **Exp2** are summarized in Table 5.6. We observed that the winning percentage of the team of guards (for policies 650, 1240, and 1600) was comparable or higher when communication actions were enabled compared to when they were not available (**Base2**). With Policy 220, however, the performance of the team of guards was slightly worse when the communication actions were enabled. Unlike the other policies, Policy 220 results in the guards spreading themselves in-front of the fort and shooting continuously, regardless of whether an attacker is in range or not. Recall that the ad hoc agent communicates the information about hidden attackers in the forest to the nearest guard and only acts as backup when its behavior model predicts that the communicated guard will *not* take the shot. With Policy 220, since the guards shoot continuously, the behavior model often mistook this shooting for a reaction to the communicated information, when in fact the guard was simply following its usual (unrelated) behavior. As a result, the ad hoc agent withheld its own action, causing the attacker to sneak into the fort and therefore the team to lose its opportunity to eliminate the attacker. This result does not contradict our general finding that communication improves performance under partial observability; rather, it shows that communication is only beneficial when the communicated information has a meaningful effect on the receiving agent’s behavior. When a teammate’s policy is not transparent to the ad hoc agent and produces actions unrelated, as with Policy 220, communication can fail to provide any benefit.

Overall, these results indicate that KAT supports adaptation to partial observability with limited communication capabilities.

Half Field Offense

The results of evaluating KAT under partial observability in the Half Field Offense domain are summarized in Table 5.7. The results indicated that KAT’s performance was slightly lower than that of the baseline teams without any ad hoc agents. We believe this difference was mainly due to noise (e.g., in the perceived angle to the goal during certain episodes and action outcomes). The ability to provide performance

<i>Policy</i>	<i>With Comm.</i>	<i>Without Comm. (Base1)</i>
Policy1	73%	58%
Policy2	19%	8%

Table 5.5: Win percentage (%) for guards with hand-crafted policies in Fort Attack domain (Exp1). Communication addresses partial observability.

<i>Policy</i>	<i>With Comm.</i>	<i>Without Comm. (Base2)</i>
Policy 220	79%	85%
Policy 650	42%	41%
Policy 1240	46%	43%
Policy 1600	18%	17%

Table 5.6: Win percentage (%) for guards with built-in policies in Fort Attack domain (Exp2). Communication addresses partial observability.

comparable with teams whose training datasets were orders of magnitude larger shows that KAT supports adaptation to partial observability.

5.4 Chapter Summary

In this chapter, we discussed the third component of KAT: *task anticipation*, which enabled an ad hoc agent to leverage the generic knowledge encoded in an LLM for high-level task anticipation; and perform non-monotonic logical reasoning with domain knowledge and behavior models to plan and execute actions to jointly achieve the current and anticipated tasks. Based on experiments in a realistic, physics-based simulation domain, VirtualHome, we showed that leveraging LLM’s generic knowledge for task anticipation, while incrementally updating the prompts and using external validators, significantly improves team efficiency by enabling the ad hoc agent to better align its actions with its teammates’ long-term goals. Through ablation studies, we showed the significance of each component of KAT and the improved performance compared with various baselines.

Next, we explored KAT’s ability to scale to additional ad hoc agents. Through experiments up to three ad hoc agents and a human agent collaborating in complex household scenarios in VirtualHome, we showed that KAT’s ability to reason with

<i>Version</i>	<i>KAT</i>	<i>Original Team</i>
Limited (2v2)	71%	76%
Full (4v5)	18%	20%

Table 5.7: Goals scored (i.e., games won) by offense team in Half Field Offense domain under partial observability (Exp3, Exp4). KAT’s performance comparable with baseline that had no ad hoc agents in the team but used training datasets that were orders of magnitude larger.

prior knowledge, rapidly identify the need for, learn and revise the behavior prediction models for multiple teammates, and apply refinement mechanisms to facilitate scalable collaboration.

In the final section of this chapter, we explored KAT’s ability to work under restricted observability. We extended the Fort Attack and Half Field Offense domains to support partial observability, and evaluated KAT’s performance under such settings. Our results suggested that reasoning with prior knowledge and predictive models also supports adaptation to partial observability when paired with communication.

Based on the results presented in this section, we make the following key observations:

- **Observation 4:** *KAT’s reasoning with prior knowledge and predictive models improves team performance* - Supported by VirtualHome domain (**Exp1** in Section 5.1.2).
- **Observation 5:** *KAT’s use of LLMs to anticipate tasks and use them as joint goals, improves performance compared to planning for one task at a time* – Supported by VirtualHome domain (**Exp1** in Section 5.1.2).
- **Observation 6:** *KAT’s use of incrementally updated prompts and validators improve task anticipation capability of the LLM and team’s performance* – Supported by VirtualHome domain (**Exp2** and **3** in Section 5.1.2).
- **Observation 7:** *KAT’s use of refinement and rapidly-learned predictive models promotes scalability* - Supported by VirtualHome domain (Section 5.2.2).
- **Observation 8:** *KAT’s reasoning with prior knowledge and predictive*

models support adaptation to partial observability with communication
- Supported by Fort Attack domain (**Exp1,2,3** and **4** in Section 5.3.2).

These observations further support Research Question1 from Section 1.2.

Overall, this chapter indicated the effectiveness of our KAT architecture in handling complex scenarios, where agents must reason about multiple possible goals, scale to many ad hoc agents, and operate under limited observability. In the next chapter, we extend our architecture to enable an ad hoc agent to acquire and integrate previously unknown knowledge.

Chapter 6

Knowledge Acquisition

Previous chapters introduced three key components of KAT architecture: (1) *knowledge representation and reasoning*, (2) *learned agent behavior models*, and (3) *task anticipation*, which enabled an ad hoc agent to compute actions based on non-monotonic logical reasoning with prior domain knowledge and incrementally learned predictive models of other agents’ behavior while effectively leveraging the generic knowledge encoded in an LLM for high-level task anticipation. This chapter explores the fourth component of KAT, *knowledge acquisition*, where we extend our current framework to address the key challenge that each agent’s knowledge of the domain may differ. Specifically, we enable an ad hoc agent to acquire knowledge in the form of objects, actions, and axioms using two strategies: (i) the agent learns from cues provided by a human during task execution (Section 6.2); and (ii) the agent explores actions in different states, identifying and correcting inconsistencies (Section 6.3). We then detail the evaluation on the effectiveness, accuracy, and consistency of KAT’s acquired knowledge, starting with the experimental procedure (Section 6.4), followed by an analysis of its performance (Section 6.5).

6.1 General Overview

In practical application domains, agents often do not have comprehensive knowledge, leading to inconsistencies and inefficient collaboration. For example, extend the human-AI agent collaboration scenario from Example 6 in the VirtualHome domain to include another ad hoc agent with incomplete knowledge (e.g., missing objects, actions, and axioms about the domain) compared to the other agents. To overcome the challenges of AHT (i.e., collaborating with other agents “on the fly” without prior coordination), this

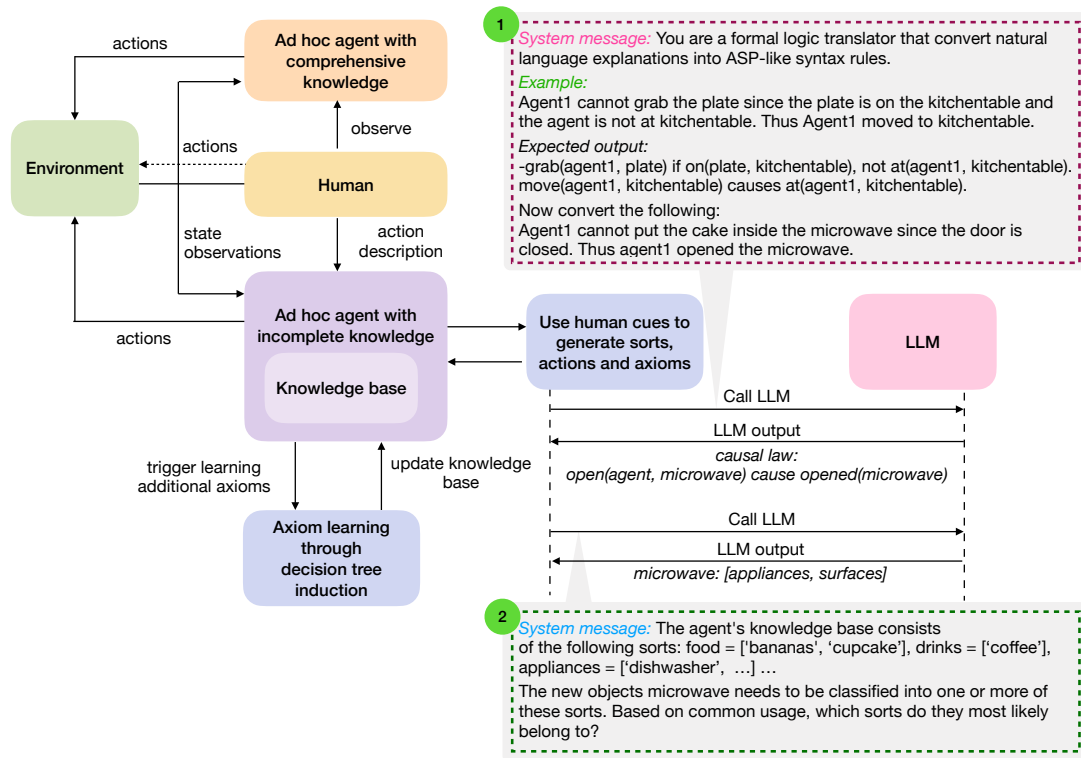


Figure 6.1: KAT leverages LLM, decision tree induction and prior knowledge to map natural language descriptions of actions and unexpected outcomes into new objects, actions and axioms in non-monotonic logical reasoning.

agent need the ability to recognize and integrate new knowledge it may come across while interacting with the domain and other agents. Without mechanisms for acquiring and incorporating such knowledge on the fly, the agents may be limited in their ability to generalize, adapt, and coordinate effectively.

Our architecture enables such an ad hoc agent to incrementally acquire missing domain knowledge in terms of unfamiliar objects and their categories, actions, and axioms that govern domain dynamics from limited examples, reducing ambiguity, and leading to reliable decision-making over time. Specifically, as shown in Figure 6.1, the ad hoc agent acquires knowledge using two strategies. First, the agent learns from cues provided by the human during task execution (Section 6.2); second, it refines this knowledge by exploring the newly learned actions in different initial states, identifying inconsistencies and addressing them (Section 6.3). In the following sections, we explore these methods in more detail.

6.2 Learning From Natural Language Descriptions

We begin by describing the first method, in which the ad hoc agent learns new domain knowledge from cues provided by a human during task execution. Consider a home environment consisting of three agents as shown in Figure 6.1: a human, an ad hoc agent with relatively complete knowledge (i.e., has access to a more comprehensive and accurate model of the domain), and an ad hoc agent with incomplete knowledge. At each step, all three agents receive state observations from the domain, which they independently use to select and execute their own actions. In addition to executing its own actions, the human occasionally describes an agent’s actions (e.g., “Agent 1 cannot put the cake inside the microwave since the microwave’s door is closed”). We assume that the human agent provides truthful and accurate information, following the standard assumption in ad hoc teamwork that agents are cooperative and act to support team objectives rather than intentionally providing misleading information. The ad hoc agent with incomplete knowledge uses these descriptions to learn previously unknown domain knowledge in the form of objects, actions, and axioms (Algorithm 6). We describe the algorithm in detail below.

When the ad hoc agent receives a cue from the human, it automatically generates and sends a prompt to the LLM by combining a predefined system message (message 1 in Figure 6.1) with two examples of input and expected output for few-shot prompting, the structure of the two types of axioms (e.g., “ a causes f if p ”), and the query based on the cue (Lines 1-2, Algorithm 6). The LLM then acts as a logic translator and maps these cues into structured logic rules (candidate axioms) using their known structure. The LLM’s output is then parsed using regular expressions to discard any output that does not match the expected format of axioms and extract the axioms from output that passes this check (Line 3). From the extracted axiom, the agent extracts actions, objects, and action preconditions and effects by mapping them to causal laws or executability conditions using known axiom templates (Lines 3-6). The agent then examines its current Answer Set Prolog program, $(\Pi(\mathcal{D}, \mathcal{H}))$, to verify whether the objects from the cue are already defined (Line 7). Any new objects are returned to the LLM with a second prompt (message 2 in Figure 6.1) based on a predefined message template, existing sorts in $\Pi(\mathcal{D}, \mathcal{H})$, and the new objects, asking the LLM to assign a sort label to the new objects (Lines 8-11). The LLM categorizes the new objects into one or more of the known basic sorts. If multiple sort labels are returned for an object, the agent resolves this by selecting the lowest category in the sort hierarchy to be used

(Line 13) and the new object is added to the Answer Set Prolog program under that sort category (Line 14).

Next, the agent examines its current knowledge base ($\Pi(\mathcal{D}, \mathcal{H})$) to identify whether the action from the human cue is already defined, and if exists then retrieve the existing action (Line 17). Since the same action can be defined in multiple ways, the agent first extracts the action verb (e.g., *grab*) and checks whether it exists in the knowledge base. If an action that semantically matches this action verb exists in $\Pi(\mathcal{D}, \mathcal{H})$, the agent retrieves it; this action may exhibit the same behavior as the one in the cue but have a different name, e.g., *pick*(A, O) instead of *grab*(A, O). The agent considers two action verbs to semantically match if one is a synonym of the other. Specifically, the agent retrieves synonyms for the action verb extracted from the cue using WordNet (Miller, 1995), and checks whether any of the synonyms match an existing action verb in Σ of $\Pi(\mathcal{D}, \mathcal{H})$. If a match is found, the action verb from the LLM output is replaced by the existing action in Σ . Also, the sorts of the arguments of the action extracted from the human cue may differ from those of the action in $\Pi(\mathcal{D}, \mathcal{H})$, e.g., they may be subsorts or parent sorts. If the sorts of arguments of the action in $\Pi(\mathcal{D}, \mathcal{H})$ are parent sorts of those in the cue, the agent uses the existing action as is; if not, the agent lifts the sort of the existing action's arguments to the new sorts and updates $\Pi(\mathcal{D}, \mathcal{H})$ (Lines 20-26). It is also possible that the number of the arguments or/and the ordering of the arguments of the action extracted from the LLM output may differ from those of the corresponding action in $\Pi(\mathcal{D}, \mathcal{H})$. The agent resolves this by retaining the existing action definition from the knowledge base. If the extracted action (and any equivalent) does not exist in $\Pi(\mathcal{D}, \mathcal{H})$, the agent adds it with the lowest (i.e., most specific) sort labels for arguments (Lines 28-34). This process ensures that we do not introduce the same action multiple times with different sorts as its arguments.

This procedure is also repeated for literals (Lines 35-54), and the actions and literals are used to convert the extracted axioms to the appropriate format, e.g., '*a causes l*' converted to '*holds(f, I+I) :- occurs(a, I)*' (Lines 55-57). The axioms are then added to the $\Pi(\mathcal{D}, \mathcal{H})$ if they do not exist (Lines 58-60). A newly learned axiom may introduce a logical inconsistency with the agent's existing knowledge. Such inconsistencies are detected and corrected through the decision tree induction process described in Section 6.3. If the inconsistency prevents the Answer Set Prolog solver from returning an answer set at plan time, the agent falls back on the recovery mechanism described in Section 4.6 (i.e., execute the next action from its previously computed plan, select a domain-specific safe action, or pass the task to a teammate).

Algorithm 6: Learning From Cues

Input: *cue*: Human cue; $\Pi(\mathcal{D}, \mathcal{H})$: Answer Set Prolog program; *LLM*: Pretrained LLM model described in equation 5.1; *systems_msg1*: Predefined system message describing the task (system msg1 in Figure 6.1); *system_msg2*: Predefined system message describing the task (system msg2 in Figure 6.1); *examples*: two fixed example (cue, axiom) pairs showing how a human cue can be converted to a causal law and an executability condition; $\mathcal{P}_{ax}$: regular expression pattern to identify Answer Set Prolog axioms; $\mathcal{P}_{act}$: regular expression pattern to identify Answer Set Prolog actions; $\mathcal{P}_{lit}$: regular expression pattern to identify Answer Set Prolog literals; $\mathcal{P}_{obj}$: regular expression pattern to identify Answer Set Prolog objects.

Output: $\Pi(\mathcal{D}, \mathcal{H})$: Updated Answer Set Prolog program.

```

1 prompt  $\leftarrow$  system_msg1  $\cup$  examples  $\cup$  cue
2 response  $\leftarrow$  LLM(prompt)
3 axiom  $\leftarrow$   $\mathcal{P}_{ax}$ (response) // NULL if no match
4 act  $\leftarrow$   $\mathcal{P}_{act}$ (axiom) // NULL if no match
5 literals  $\leftarrow$   $\mathcal{P}_{lit}$ (axiom) // NULL if no match
6 objects  $\leftarrow$   $\mathcal{P}_{obj}$ (axiom) // NULL if no match
7 new_objects  $\leftarrow$   $\{o \in \text{objects} \mid o \notin \Sigma \text{ of } \Pi(\mathcal{D}, \mathcal{H})\}$ 
8 if new_objects  $\neq \emptyset$  then
9   Sorts  $\leftarrow$   $\{s \mid s \text{ is a sort declared in } \Pi(\mathcal{D}, \mathcal{H})\}$ 
10  prompt  $\leftarrow$  system_msg2  $\cup$  Sorts  $\cup$  new_objects
11  response  $\leftarrow$  LLM(prompt)
12  for o  $\in$  new_objects do
13     $s^* \leftarrow s \in \text{response}$  such that  $\forall s' \in \text{response}, s$  is a subsort of  $s'$ 
14     $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{(o, s^*)\}$ 
15  end
16 end
17 existing_act  $\leftarrow a, a \in \Pi(\mathcal{D}, \mathcal{H}) : a$  is an action with the same behavior as act
18 Sorts  $\leftarrow$   $\{s \mid s \text{ is a sort declared in } \Pi(\mathcal{D}, \mathcal{H})\}$ 
19 if existing_act with low level sorts then
20   for oi  $\in$  act do
21     si  $\leftarrow$  sort(oi, Sorts)
22     if sorti(existing_act)  $<$  si then
23       sorti(existing_act)  $\leftarrow$  si
24     end
25   end
26    $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{\text{existing\_act}\}$ 
27   act  $\leftarrow$  existing_act

```

```

28 else if  $\neg \text{existing\_act}$  then
29   for  $o_i \in \text{act}$  do
30      $s_i \leftarrow \text{sort}(o_i, \text{Sorts})$ 
31      $\text{sort}_i(\text{act}) \leftarrow s_i$ 
32   end
33    $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{\text{act}\}$ 
34 end
35 for  $\text{literal} \in \text{literals}$  do
36    $\text{existing\_literal} \leftarrow l, l \in \Pi(\mathcal{D}, \mathcal{H}) :$ 
37    $l$  is a literal with the same behaviour as  $\text{literal}$ 
38   if  $\text{existing\_literal}$  with low level sorts then
39     for  $o_i \in \text{literal}$  do
40        $s_i \leftarrow \text{sort}(o_i, \text{Sorts})$ 
41       if  $\text{sort}_i(\text{existing\_literal}) < s_i$  then
42          $\text{sort}_i(\text{existing\_literal}) \leftarrow s_i$ 
43       end
44     end
45      $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{\text{existing\_literal}\}$ 
46      $\text{all\_literals} \leftarrow \text{all\_literals} \cup \{\text{existing\_literal}\}$ 
47   else if  $\neg \text{existing\_literal}$  then
48     for  $o_i \in \text{literal}$  do
49        $s_i \leftarrow \text{sort}(o_i, \text{Sorts})$ 
50        $\text{sort}_i(\text{literal}) \leftarrow s_i$ 
51     end
52      $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{\text{literal}\}$ 
53      $\text{all\_literals} \leftarrow \text{all\_literals} \cup \{\text{literal}\}$ 
54   end
55  $\text{axiom}[p] \leftarrow \text{act}, \forall p \in \text{axiom} : \text{behaviour}(p) = \text{behaviour}(\text{act})$ 
56  $\text{axiom}[p] \leftarrow l, \forall p \in \text{axiom} \setminus \{\text{act}\}, \exists l \in \text{all\_literals} : \text{behaviour}(p) =$ 
57    $\text{behaviour}(l)$ 
58  $\text{axiom} \leftarrow \text{axiom}[\text{sort values} \rightarrow \text{ASP variables}]$ 
59 if  $\text{axiom} \notin \Pi(\mathcal{D}, \mathcal{H})$  then
60    $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{\text{axiom}\}$ 
61 end

```

Example 8 [*Learning from Natural Language Descriptions*]

Consider an ad hoc agent that receives the following cue from the human during task execution:

‘Agent1 cannot put the cake inside the microwave, since the microwave door is closed. Thus, agent1 opened the microwave’

The ad hoc agent generates **Prompt1** (Figure 6.1) using the predefined system message, examples, and the cue as described earlier. This prompt is then sent to the LLM, which generates the following output:

exec cond: $\neg$ put(agent1, cake, microwave) if not opened(microwave)
causal law: open(agent1, microwave) cause opened(microwave)

Next, by applying the known axiom templates to this LLM output, the ad hoc agent identifies and extracts the corresponding actions, literals, and objects from these candidate axioms.

actions: *put(agent1, cake, microwave), open(agent1, microwave)*
literals: *opened(microwave)*
objects: *agent1, cake, microwave*

The agent then checks whether the extracted objects exist in the knowledge base. Since object *microwave* is not, it sends the second prompt (**Prompt2** in Figure 6.1) to the LLM using the system message, existing sorts extracted from the agent’s knowledge base, and the new object *microwave*. In response, the LLM returns the sort categories to which the object *microwave* may belong:

microwave: [appliances, surfaces]

The agent selects the most specific category, *appliances*, and updates its knowledge base to include the new object *microwave* within this category.

Next, the agent replaces the object references in the actions with their corresponding sort categories (e.g., *put(agent1, cake, microwave)* is converted to the generalized form *put(agent, food, appliance)*, and *open(agent1, microwave)* converted to *open(agent, appliance)*). It then extracts the action verbs *put*, *open* and checks whether they exist in the knowledge base. Since the action *put* exists, the agent retrieves the corresponding action *put(agent, fruit, surface)*. The agent compares this retrieved action with the generalized cue action, and as they differ in their sort categories, the agent examines the hierarchical relationship between the sort categories, *fruit* vs *food*, *surface* vs *appliance*. Since *appliance* is a subsort of *surface*, the agent keeps it as it is.

However, as *food* is a parent sort of *fruit* the agent raise the sort level in the action to *put(agent, food, surface)*, and update the knowledge base with the new action. With no matching action found for *open(agent, appliance)* in the knowledge base, the agent adds *open(agent, appliance)* as a new action to the knowledge base.

The agent then replaces the object references in the literals, *opened(appliance)*, and checks whether they already exist in the knowledge base. Since a matching literal is found, the agent reuses it without modifications.

Next, the agent uses the selected actions (*put(agent, food, surface)*, *open(agent, appliance)*) and literals (*opened(appliance)*), together with the axiom templates provided by the LLM to create executability conditions and causal laws. It replaces the sort values in the axioms with their corresponding variables and, after confirming that these axioms are not already in its knowledge base, adds the newly learned axioms to the knowledge base.

executability condition: $\neg \text{occurs}(\text{put}(R, F, A), I) :- \text{not holds}(\text{opened}(A), I)$
causal law: $\text{holds}(\text{opened}(A), I + 1) :- \text{occurs}(\text{open}(R, A), I)$

The learned knowledge is used for subsequent reasoning, with the corresponding observations used to augment and refine the existing knowledge through decision tree induction, as described in the next section.

6.3 Learning From Simulated Execution

The second strategy enables the ad hoc agent to refine its knowledge based on observations obtained by actively using its current knowledge for planning and execution. Our approach adapts and extends prior work (Mota et al., 2021), combining *decision tree induction* with knowledge-based reasoning. We integrate this approach into AHT, enabling our agent to learn and refine its own knowledge in response to inconsistencies it observed during interaction. These inconsistencies are stored as labeled examples, from which decision trees are learned and translated into logical rules (axioms) that represent domain dynamics. The resulting axioms are then validated and incorporated into the ad hoc agent knowledge base, supporting adaptive decision-making in AHT scenarios. Below, we describe this process in more details.

Algorithm 7: Learning From Simulated Execution

Input: $\mathcal{A}$: newly learned actions; $\Pi(\mathcal{D}, \mathcal{H})$: Answer Set Prolog program; $\mathcal{N}$: number of training examples; $\mathcal{K}$: number of iterations; *threshold*: confidence level; θ : environment configuration parameters, e.g., x% time the microwave door is open, y% time the location of the agent is same as the location of the object in consideration.

Output: $\Pi(\mathcal{D}, \mathcal{H})$: Updated Answer Set Prolog program.

```

1  while  $i < 50$  do
2       $a_I \leftarrow a, \quad a \in \mathcal{A}$ 
3       $S_I \leftarrow \mathcal{S}(\theta)$ 
4       $S = T(S_I, a_I)$ 
5      if  $S$  is not consistent with expectations then
6           $trigger\_learning \leftarrow True$ 
7           $a \leftarrow a_I$ 
8          exit
9      end
10 end
11 if  $trigger\_learning$  then
12      $unobserved = \emptyset, unexpected = \emptyset$ 
13      $Sorts \leftarrow \{s \mid s \text{ is a sort declared in } \Pi(\mathcal{D}, \mathcal{H})\}$ 
14      $a \leftarrow$  declaration of  $a$  in  $\Pi(\mathcal{D}, \mathcal{H})$ 
15     while  $unobserved < \mathcal{N}$  or  $unexpected < \mathcal{N}$  do
16          $a_i \leftarrow a[s_i \rightarrow o_i, \forall s_i \in a]$ 
17          $S_i \leftarrow \mathcal{S}(\theta)$ 
18          $S \leftarrow T(S_i, a_i)$ 
19          $unobserved \leftarrow \{l \in S \mid \exists o \in a_i, o \in l, l \text{ is } unobserved\}$ 
20          $unexpected \leftarrow \{l \in S \mid \exists o \in a_i, o \in l, i \text{ is } unexpected\}$ 
21     end
22     if  $unobserved > \mathcal{N}$  then
23          $Axioms \leftarrow \text{DecisionTreeInduction}(unobserved, \mathcal{K}, threshold)$ 
24     end
25     if  $unexpected > \mathcal{N}$  then
26          $Axioms \leftarrow \text{DecisionTreeInduction}(unexpected, \mathcal{K}, threshold)$ 
27     end
28 end
29  $\Pi(\mathcal{D}, \mathcal{H}) \leftarrow \Pi(\mathcal{D}, \mathcal{H}) \cup \{Axioms\}$ 
30 return  $\Pi(\mathcal{D}, \mathcal{H})$ 

```

Algorithm 8: Decision Tree Induction

Input: $\mathcal{F}$: Relevant literals; $\Pi(\mathcal{D}, \mathcal{H})$: Answer Set Prolog program; $\mathcal{K}$:
Number of iterations; *threshold*: Confidence level.

Output: *Axioms*: Candidate axioms.

```

1   $Sorts \leftarrow \{s \mid s \text{ is a sort declared in } \Pi(\mathcal{D}, \mathcal{H})\}$ 
2  for  $f \in \mathcal{F}$  do
3      for  $o_i \in f$  do
4           $s_i \leftarrow sort(o_i, Sorts)$ 
5           $sort_i(f) \leftarrow s_i$ 
6      end
7       $f \leftarrow f[\text{sort values} \rightarrow \text{ASP variables}]$ 
8  end
9   $dataset \leftarrow \{(f, 1) \mid f \in \mathcal{F} : \text{inconsistent}\} \cup \{(f, 0) \mid f \in \mathcal{F} : \text{consistent}\}$ 
10  $Axioms \leftarrow \emptyset$ 
11 for  $k = 1$  to  $\mathcal{K}$  do
12      $cands \leftarrow \emptyset$ 
13      $training, test \leftarrow dataset$ 
14      $tree \leftarrow CART(training)$ 
15      $cand\_paths \leftarrow \{path \in tree \mid \exists x \in test : x \text{ traverses } path\}$ 
16     for  $c \in cand\_paths$  do
17         if  $leaf\_support(c) \geq threshold$  and  $min\_samples(c) \geq 0.02$  and
             $leaf\_dom\_class(c) = inconsistent$  then
18              $C \leftarrow \{N \mid N \subseteq nodes(c), N \neq \emptyset\}$ 
19              $cands \leftarrow \{c \in C \mid support(c) \geq threshold\}$ 
20         end
21     end
22      $Axioms \leftarrow Axioms \cup cands$ 
23 end
24  $Axioms \leftarrow \{r \in Axioms \mid freq(r, Axioms) > n\}$ 
25 return  $Axioms$ 

```

At the end of each iteration, the ad hoc agent selects an action a_I from the newly learned set of actions A , and an initial state of the environment S_I , and simulates the execution of a_I in S_I (Lines 2-4, Algorithm 7). It then collect information about the outcomes (e.g., end state, presence or absence of an inconsistency). In the presence

of an inconsistency: a missing expected outcome indicates the potential absence of an executability condition(s); and observation of additional effects other than what is expected, indicates a missing causal law(s); the agent triggers learning with the selected action (Lines 5-9). If all observations match expectations for different actions and states, the current knowledge is considered to be comprehensive.

The agent responds to an inconsistency by simulating the execution of a_i in different states and storing the observed state transition. The agent constructs different versions of the selected action a_i by using different sorts from the same group (Line 13-16), select an initial state S_i (Line 17) and simulate its effects (Line 18). All fluent literals in the answer set at initial state, with an object constant that is also in a_i are then stored (Line 19-20). The agent repeats this procedure until sufficient data has been gathered.

Separate decision trees are constructed for learning missing causal laws and executability conditions (Lines 22-27). In the collected *training* examples, the ground terms in the literals are replaced by variables (Lines 1-8, Algorithm 8), and the dataset is reformatted with the fluent literals as features and the presence or absence of inconsistency as the class label (Line 9). Each training example then records the occurrence or non-occurrence of a fluent literal, and the presence or absence of an inconsistency, as a binary value. The processed data set is split to training and testing and the training data set is used to learn a decision tree by splitting the nodes using features that have not been used before and are likely to result in the highest reduction in entropy (Lines 13-14).

The learned tree is used to classify the test data set and for each example the sequence of decision tree nodes traversed from the root to the leaf node is recorded (Line 15). For each recorded path, only those whose leaf node's dominant class is inconsistent and agrees with their class label up to a threshold level (90%) and contains at least a minimum percentage (2%) of the dataset are used to generate candidate axioms (Lines 16-17). The candidate axioms are generated by taking all possible combinations of the nodes appearing in each selected path (Line 18). The resulting candidates are filtered, and only those that have sufficient support among the test examples (threshold value 98% in our experiments) are retained (Line 19). These candidates are further filtered to retain the general forms of the axioms. Also, the decision tree induction process is repeated multiple times over the training data to explore different subsets of data (Line 24). Only axioms that are retained over multiple such repetitions are returned and added to $\Pi(\mathcal{D}, \mathcal{H})$ (Line 29).

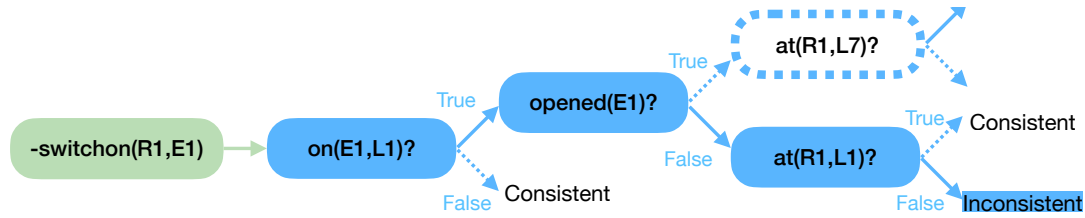


Figure 6.2: Part of the decision tree created to learn missing executability conditions.

To better illustrate how the agent learns new axioms through observations, we present the following example from the VirtualHome domain.

Example 9 [*Axiom Induction from Observations*]

Consider an ad hoc agent that has recently learned the action *switch_on(agent, appliance)*, which represents turning on an appliance. The agent is still unaware of all the preconditions required for the action to be executable (i.e., executability conditions). Thus, the agent simulates the action in different initial states (Lines 1-4). When the agent attempts to perform this action on a microwave (i.e., *switch on* the microwave) while its door is open, the action unexpectedly fails. This is due to the safety mechanism in the microwave that prevents it from being turned on when the door is open. As the agent is unaware of this precondition, it expected the state of the microwave to change from switched off to switched on (as a successful outcome). As the expected outcome was not observed, the failure suggests the existence of an unknown precondition (Lines 5-9). The agent identified this as an inconsistency in its knowledge base and start simulating the execution of action *switch_on* in different initial states and store the observations. All the fluents literals in the answer set which has the object constants, *agent = ad hoc agent* and *appliance = microwave* are saved (Lines 12-20). Once enough data has been gathered, the agent reformats the data set as described before, and start building the decision tree (Lines 22-27). Part of the decision tree generated by this process is shown in Figure 6.2. Statement 6.1 shows two axioms that the agent has learned by traversing the tree. Statement 6.1(a) implies that the agent cannot switch on an appliance, if the door of the appliance is open; Statement 6.1(b) implies that the agent cannot switch on an appliance if the agent is not in the same location as the appliance.

$$-occurs(\text{switchon}(R, E), I) \leftarrow holds(\text{opened}(E), I) \quad (6.1a)$$

$$-occurs(\text{switchon}(R, E), I) \leftarrow holds(\text{on}(E, L), I), not\ holds(\text{at}(R, L), I) \quad (6.1b)$$

In the next section, we describe the experimental setup used to evaluate the knowledge acquisition capabilities of KAT architecture.

6.4 Experimental Procedure

We experimentally evaluated KAT’s ability to acquire new domain knowledge in three domains, VirtualHome, Fort Attack, and Half Field Offense (Section 4.7.1). Recall that we introduced two strategies for knowledge acquisition in this chapter; (1) *learning from natural language descriptions* (Section 6.2); and (2) *learning from simulated execution* (Section 6.3). VirtualHome domain, being the most complex of the three domains, with a significantly larger action space, and a set of axioms, and the presence of a human-like agent, allowed us to evaluate both methods for knowledge acquisition comprehensively. While, Fort Attack and Half Field Offense domains were used to evaluate the second method (learning from simulated execution) due to their limited action spaces. Removing action knowledge from these domains risked making the agent non-functional and incapable of completing the basic tasks.

VirtualHome

Similar to our previous experiments in the VirtualHome domain (described in Section 5.1.1), each episode consisted of a human agent and an ad hoc agent, and an external task generator provided the sequence of tasks to be completed during each episode. To evaluate the knowledge acquisition capabilities, we introduced a second ad hoc agent with an incomplete knowledge base to the two agent team (ad hoc agent and human). Specifically, the complete domain knowledge of an ad hoc agent consisted of 31 objects, 7 actions, 9 causal laws, and 26 executability conditions. The new ad hoc agent’s Answer Set Prolog program included only a subset of this knowledge, containing 17 objects, 4 actions, 6 causal laws, and 16 executability conditions. This corresponded to the absence of around 40 – 45% knowledge. A complete specification of the domain knowledge, also showing the subset of the knowledge provided to the incomplete agent, is included in Appendix A.4. We provided the agent enough initial

knowledge to perform some basic activities, while also withholding key knowledge to create gaps that limited the agent’s ability to complete tasks. All three agents received the same domain observations, which they used to plan their individual actions. The two ad hoc agents used the learned behavior prediction models of its teammates to anticipate their likely actions (Section 4.4), while the human agent’s actions were solely based on its own knowledge and observations, without using any models to predict the actions of its teammates.

During task execution, the human agent periodically describes actions of the knowledgeable ad hoc agent to the ad hoc agent with missing knowledge. This agent then used the procedure described in Section 6.2 to process these descriptions into Answer Set Prolog sorts, actions, and axioms with the help of an LLM and added the validated information to its knowledge base. At the end of each episode, it also used decision tree induction to learn new axioms (Section 6.3).

We then evaluated the ad hoc agent’s ability to learn missing knowledge across 10 episodes, with each episode randomly selected from five different task sequences and each sequence consisting of four tasks (**Exp1**). In each episode, the agent began with the same initial knowledge (i.e., fixed subset of sorts, actions, axioms). We then recorded the number of objects, actions, and axioms learned in each trial, and measured the precision and recall of learning these axioms compared with a complete Answer Set Prolog program (ground truth).

Next, in **Exp2**, we extended each episode from **Exp1** to include three consecutive runs. The first run in an episode had the same initial knowledge as in **Exp1**, but the subsequent two runs built on the knowledge acquired in the earlier runs and a potentially different sequence of tasks. This process was repeated for the 10 episodes, allowing the agent to refine its learned knowledge between repetitions. We recorded the objects, actions, and axioms learned after each run and episode, and computed the precision and recall as the performance measures.

Finally, to evaluate the impact the learned knowledge has on the performance of the ad hoc agent, in **Exp3**, we ran 20 trials with and without the learned axioms (**Base1**). As the performance measure, we recorded the number of steps (plan length) and the total time taken to complete the task(s).

Fort Attack

Each episode of our experiments in the Fort Attack domain consisted of two teams: guard team (three guards), attacker team (three attackers). Our ad hoc agent repre-

sented one of the guard team members. Each other agent in their respective teams (guard/attacker) were assigned a policy from the built-in policies described in Section 4.7.2. Since Fort Attack domain consists of a relatively small action and axiom space, removing action knowledge risked disabling the agent’s basic functionality. Thus, in this domain we restricted our evaluation to the second method, *learning from simulated execution* (Section 6.3).

First, to evaluate the ad hoc agent’s ability to accurately learn previously unknown axioms, in **Exp4**, we removed three axioms (two causal laws and one executability condition) from the agent’s knowledge base.

$$\text{holds}(\text{face}(R,D),I+1) \leftarrow \text{occurs}(\text{rotate}(R,D),I). \quad (6.2a)$$

$$\text{holds}(\text{in}(R,X,Y),I+1) \leftarrow \text{occurs}(\text{move}(R,X,Y),I). \quad (6.2b)$$

$$\neg \text{occurs}(\text{shoot}(R,A),I) \leftarrow \neg \text{holds}(\text{in_range}(R,A),I). \quad (6.2c)$$

We then executed the algorithm from Section 6.3 twenty times in the Fort Attack domain. Each time, the ad hoc agent selected a different action and simulated its effects to incrementally learn all the missing axioms. As the **performance measures**, we recorded the precision and recall of the learned axioms, based on how accurately the agent identified the literals within the missing axioms.

Next, in **Exp5**, we allowed the ad hoc agent to use the axioms it learned in **Exp4** for reasoning. We conducted 100 episodes with each built-in policy in the Fort Attack domain. As the baseline (**Base2**) we used an ad hoc agent that does not use the learned axioms in its planning. As the performance measure, we used the guard team’s win percentage as the ad hoc agent was part of that team.

Half Field Offense

Each game episode in this domain consisted of two teams: offense team and the defense team. The ad hoc agent represented one of the offense team members and the teammates were selected from the six external agent teams from 2013 RoboCup simulation competition, *Helios*, *Gliders*, *Cyrus*, *Axiom* and *Aut*, and the defense agents were based on *Agent2d* team (Section 4.7.2).

Similar to Fort Attack domain, there is a tight coupling between available actions and task solvability in the Half Field Offense domain. Thus, we used this domain to evaluate the second method, *learning from simulated execution*. We removed four axioms (one causal law and three executability condition) from the agent’s knowledge

base in **Exp6**.

$$\text{holds}(\text{in}(R, X, Y), I + 1) \leftarrow \text{occurs}(\text{dribble}(R, X, Y), I). \quad (6.3a)$$

$$\neg \text{occurs}(\text{dribble}(R, X, Y), I) \leftarrow \neg \text{holds}(\text{has_ball}(R), I). \quad (6.3b)$$

$$\neg \text{occurs}(\text{shoot}(R), I) \leftarrow \neg \text{holds}(\text{has_ball}(R), I). \quad (6.3c)$$

$$\neg \text{occurs}(\text{pass}(R, T1), I) \leftarrow \neg \text{holds}(\text{has_ball}(R), I). \quad (6.3d)$$

We ran the axiom learning algorithm (Section 6.3) for twenty times and during each run, the ad hoc agent selected a different action and simulated its effects across the domain. To evaluate the accuracy of the learned axioms, we measured the precision and recall, based on how accurately the agent identified the relevant literals within each axiom.

To evaluate the impact of the newly acquired knowledge on the ad hoc agent's performance, in **Exp7**, we allowed the agent to use the axioms it learned in **Exp6** in the Half Field Offense domain. We ran 100 game episodes for each built-in policy and compared the results against a baseline (**Base3**), in which the ad hoc agent does not use the learned axioms. The performance measure was the fraction of the goals scored by offense team in the Half Field Offense domain.

6.5 Quantitative Results

This section presents the results of the experiments described above, organized into three domains: VirtualHome, Fort Attack, and Half Field Offense.

VirtualHome

The results of **Exp1** are summarized in Table 6.1. We observed high precision and recall for learning the missing axioms. Next, Figure 6.3 summarized the results of **Exp2** with three consecutive runs in each of 10 episodes. By the end of the first run, the ad hoc agent successfully learned 4-5 of 14 missing objects, all three missing actions, 1-2 of three missing causal laws, and 6-7 of 10 missing executability conditions. After three consecutive runs, the number of discovered objects increased to 8-9 out of 14, 2-3 out of three causal laws, and 9-10 out of 10 executability conditions. The few remaining objects that were not learned are primarily due to their limited relevance to the tasks or because they were handled directly by the human, and therefore not described to the ad hoc agent. The steady increase in number of objects, actions, and

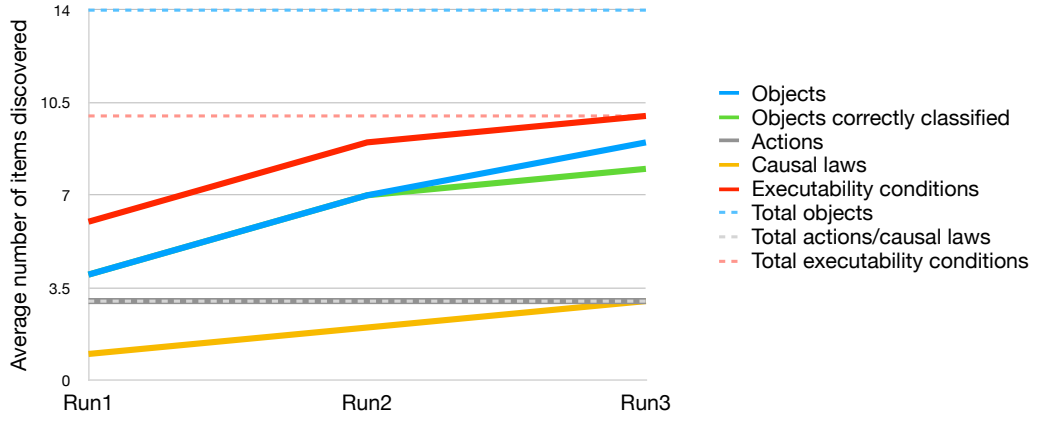


Figure 6.3: Average number of objects, actions, and axioms learned by the ad hoc agent in three consecutive runs. The results are averaged over 10 trials, with each trial consisting of three runs (Exp2).

<i>Axiom type</i>	<i>Precision</i>	<i>Recall</i>
Causal laws	0.96	1.0
Executability conditions	1.0	1.0

Table 6.1: Average precision and recall of learned axioms in 10 episodes; the agent accurately learns previously unknown axioms (Exp1).

axioms learned, along with the high precision and recall, indicated the agent’s ability to reliably learn new knowledge. Although the LLM occasionally provided incorrect Answer Set Prolog formats, these errors occurred rarely; most of the time the LLM output was correct as its use was limited to straightforward tasks and the mechanisms described in Section 6.2 were sufficient to account for these errors.

Next, Table 6.2 summarizes the results of **Exp3**, which evaluated the effectiveness of the learned axioms in computing plans for completing tasks as a team. Since the actual time taken and the number of action steps required to complete the tasks can vary substantially based on the tasks under consideration, the average of these values over the individual trials may not be meaningful. Thus, we ran paired trials with and without the learned axioms and computed the value of these measures for the former (i.e., when the learned axioms were included) as a fraction of the values of these measures for the latter (i.e., without the learned axioms). Table 6.2 shows that when the ad hoc agent reasoned with the learned objects, actions, and axioms, it substantially im-

<i>Architecture</i>	<i>Steps</i>	<i>Time</i>
With learned axioms	0.76	0.84
Without learned axioms (Base1)	1.00	1.00

Table 6.2: Average number of action steps and time taken by the team to complete task sequences; represented as a fraction of the values of these measures for the baseline; reasoning with learned knowledge substantially improves performance. (Exp3)

proved performance compared to not using the acquired knowledge. In the absence of the learned knowledge, at least one ad hoc agent often could not compute valid plans to complete the assigned tasks, and could not contribute much to the team's performance. The team was essentially operating with one less member in such cases, and the other two members had to execute additional actions to complete the tasks. These results emphasize the importance of learning previously unknown knowledge during execution.

Fort Attack

As stated earlier, in **Exp4**, three axioms (two causal laws and one executability condition) were removed from the ad hoc agent's knowledge, and the axiom learning algorithm was executed 20 times. We then measured the precision and recall of the agent learning these missing axioms in each run and summarized the results in Table 6.3. We observed high precision and average recall values. The lower recall can be explained by the fact that only axioms with a high degree of support from the observed transitions were added to the ad hoc agent's knowledge, *i.e.*, a correct candidate axiom could be filtered and not added to the ad hoc agent's knowledge if its addition is not justified by sufficient experiences of the ad hoc agent. Also, we observed that the recall values improved with additional experiences. These results indicate that the agent can accurately learn previously unknown causal laws and executability conditions when using KAT architecture.

Next, Table 6.5 summarize the results of **Exp5**. There was a significant improvement in the number of game episodes in which the guards team won in the Fort Attack domain, particularly with policies 220, 650 and 1240, compared with **Base2**, which did not use the acquired knowledge from **Exp4**. With the built-in policy 1600 the results were comparable with and without the learned axioms. Note that the learned

<i>Domain</i>	<i>Precision</i>	<i>Recall</i>
Fort Attack	1.00	0.43

Table 6.3: Precision and recall of the learned axioms in Fort Attack domain (Exp4).

<i>Domain</i>	<i>Precision</i>	<i>Recall</i>
Half Field Offense	0.89	0.47

Table 6.4: Precision and recall of the learned axioms in Half Field Offense domain (Exp6).

<i>Policy</i>	<i>Without learned axioms (Base2)</i>	<i>With learned axioms</i>
Policy 220	81%	86%
Policy 650	38%	46%
Policy 1240	47%	56%
Policy 1600	21%	21%

Table 6.5: Win percentage of the team of guards in the Fort Attack domain with and without the learned axioms (Exp5).

axioms are not necessarily used in each episode. These results show that reasoning with incrementally learned axioms improves the performance of the ad hoc agent and the team.

Half Field Offense

In **Exp6**, four axioms (one causal law and three executability conditions) were removed from the ad hoc agent’s knowledge, and the axiom learning algorithm was executed 20 times. Table 6.4 summarize the precision and recall values of the learned axioms in each run. In line with the previous results from **Exp4** in the Fort Attack domain, we observed high precision and average recall values in this domain. These results further indicates that the agent can accurately learn previously unknown causal laws and executability conditions using KAT.

As summarized in Table 6.6 similar performance was observed in **Exp7**, with a substantial improvement in the fraction of goals scored by the offense team when the ad hoc agent included the learned axioms during reasoning and the other agents used the policies *Helios*, *Gliders*, *Cyrus* and *Axiom*. With *Aut* policy, the results were comparable with and without the learned axioms. These results further supports that reasoning with incrementally learned axioms improves the performance of the ad hoc agent and

<i>Policy</i>	<i>Without learned axioms (Base3)</i>	<i>With learned axioms</i>
Helios	30%	35%
Gliders	25%	29%
Cyrus	32%	38%
Axiom	14%	24%
Aut	19%	19%

Table 6.6: Win percentage of the team of guards in the Half Field Offense domain with and without the learned axioms (Exp7).

the team.

We mainly observed two types of errors during our evaluation. First when using the method *Learning from Natural Language Descriptions*, the LLM provided incorrect knowledge in two forms. First, the LLM may assign an object to an incorrect sort category. However, we noticed that these errors were mainly occurred due to our design of the sort hierarchy rather than the LLM itself. For example, we observed the LLM categorized the object *computer* under the sort category *others* instead of *electricals* as we expected. In our sort hierarchy, *electricals* was defined as a subcategory of *appliances*, distinguishing objects that can be switched on/off but do not have an openable door (e.g., *computer*, *coffee machine*) from objects that can be both switched on/off and opened/closed (e.g., *microwave*, *fridge*). Since *cellphone* does not support a switch on action in the VirtualHome domain, we placed it under category *others* instead of *electricals*. When *cellphone* was present in the *others* category, the LLM categorized *computer* under *others* as well. When the agent learned the object *computer* before *cellphone*, it correctly placed the *computer* under *electricals*. This, we consider this to be an error in the design of our sort hierarchy rather than an error made by the LLM.

Second, the LLM may provide already known actions or literals with different names/forms since it does not have access to the agent’s current knowledge base. We already anticipated this kind of errors and addressed them through the mechanisms described in Section 6.2. Namely, when the LLM used a different name for an existing action, e.g., *pick(agent, object)* instead of *grab(agent, object)*, this was resolved using WordNet synonym matching; when the LLM provided an action with the wrong number of arguments, e.g., *grab(object)* instead of *grab(agent, object)*; or with arguments in the wrong order, e.g., *grab(object, agent)* instead of *grab(agent, object)*,

we used the existing action to resolve this since a matching action already exists in the knowledge base.

Next, when using the second method *Learning from Simulated Execution* we observed two types of errors. First, when the size of the training data was small, the agent sometimes learned causal laws with additional preconditions. For example, the agent learned axioms like:

$$\text{holds}(\text{opened}(E), I+1) \leftarrow \text{occurs}(\text{open}(E), I), \text{holds}(\text{at}(R, E), I). \quad (6.4)$$

and,

$$\text{holds}(\text{opened}(E), I+1) \leftarrow \text{occurs}(\text{open}(E), I), \text{holds}(\text{on}(E, L), I). \quad (6.5)$$

where the correct causal law was:

$$\text{holds}(\text{opened}(E), I+1) \leftarrow \text{occurs}(\text{open}(E), I). \quad (6.6)$$

This error was reduced as more training data became available. Second, in domains such as Fort Attack and Half Field Offense, where locations were represented using X and Y grid coordinates that were drawn from overlapping ranges of values, the agent sometimes learned axioms that did not distinguish between the two coordinates, e.g., $\text{dribble}(R, X1, X2)$ instead of $\text{dribble}(R, X, Y)$. This was due to each numeric value can be either an X or a Y coordinate in the training data. We rely on the validation mechanisms described in above sections to discard such erroneous candidates. Our design choices prioritize precision over learning incorrect axioms as that is considerably more costly to the agent’s subsequent reasoning than a missing axiom. This is consistent with the lower recall observed in the Fort Attack and Half Field Offense domains (Tables 6.3 and 6.4).

6.6 Qualitative Results

In this section, we provide some execution traces from the VirtualHome domain as a qualitative evaluation of the knowledge acquisition capabilities of KAT.

Consider the situation in which the ad hoc agent was unaware of the action $\text{grab}(\text{agent}, \text{object})$ that refers to an object being picked up. When the human provided the cue, “Agent2 grabbed the apple”, the ad hoc agent used the process described in Section 3.3 to obtain the following output from the LLM:

$$\text{grab}(\text{agent2}, \text{apple}) \text{ causes } \text{in_hand}(\text{agent2}, \text{apple}). \quad (6.7)$$

From this LLM output, the agent first extracted $grab(agent2, apple)$. Since $apple$ is not already defined in its current knowledge base (Answer Set Prolog program), the agent sent this to the LLM (message 2 in Figure 2) to receive the output $apple: food$, which states that $apple$ belongs to the sort $food$. The ad hoc agent then generalized this action to $grab(agent, food)$ based on the LLM output, and added the action to its knowledge. Further, the agent also learned the causal law:

$$holds(in_hand(R, F), I + 1) \leftarrow occurs(grab(R, F), I). \quad (6.8)$$

In subsequent runs, the ad hoc agent applied the grab action to food items, but remained unaware that this action was applicable to other objects such as a book or a cellphone. As a result, it did not contribute to tasks such as preparing the workstation. However, since the agents executed tasks as a team, the human description of the execution of $grab(agent, book)$ by another agent enabled the ad hoc agent to generalize the action to $grab(agent, graspable)$ and update the causal law:

$$holds(in_hand(R, G), I + 1) \leftarrow occurs(grab(R, G), I). \quad (6.9)$$

Once the agent learned the above causal laws, it was able to use them to compute plans and complete tasks even though it was unaware of all the executability conditions for this action. Later, the human provided the cues “Agent2 cannot grab the plate since the plate is on the kitchentable and Agent2 is not at kitchentable” and “Agent2 moved to kitchentable”. The agent used the LLM to translate these cues to the following executability condition and causal law:

$$\neg grab(agent2, plate) \text{ if } on(plate, kitchentable), \text{ not } at(agent2, kitchentable). \quad (6.10a)$$

$$move(agent2, kitchentable) \text{ causes } at(agent2, kitchentable). \quad (6.10b)$$

The ad hoc agent used this output to learn the following axioms:

$$\neg occurs(grab(R, G), I) \leftarrow holds(on(G, L), I), \text{ not } holds(at(R, L), I). \quad (6.11a)$$

$$holds(at(R, L), I + 1) \leftarrow occurs(move(R, L), I). \quad (6.11b)$$

Since Statement 6.11(b) was already in its knowledge base, the ad hoc agent only added Statement 6.11(a) to its knowledge base. These results indicate that the agent is able to incrementally learn and generalize the learned knowledge to improve task completion performance.

6.7 Chapter Summary

This chapter explored the fourth component of KAT, *knowledge acquisition*, a strategy for the ad hoc agent to acquire unknown domain knowledge in terms of objects, actions and axioms (causal laws and executability conditions) while collaborating with other agents in complex domains. Specifically, the architecture enabled the ad hoc agent to: (1) leverage a LLM to translate natural language descriptions of action outcomes into formal Answer Set Prolog representations of previously unknown objects, actions, and axioms; (2) use decision tree induction to incrementally learn previously unknown axioms based on observations.

Through experiments conducted in the three simulation environments Fort Attack, Half Field Offense, and VirtualHome, we showed that KAT enabled an ad hoc agent to acquire missing knowledge reliably and efficiently, resulting in minimal plans and improved performance. Based on the results presented in this section, we make the following key observations:

- **Observation 9:** *KAT enables an ad hoc agent to accurately learn previously unknown objects, actions, and axioms* - Supported by all domains (Exp1, 2, 4 and 6).
- **Observation 10:** *Reasoning with incrementally learned knowledge improves the performance of the ad hoc agent and the team* - Supported by all domains (Exp3, 5 and 7).

These observations further support the Research Question1 from Section 1.2. In the next section, we explore KAT's ability to explain the reasoning behind its decisions and belief to enhance transparency.

Chapter 7

Explanation Generation

Previous chapters explored four key components of KAT architecture; *knowledge representation and reasoning*, *learned agent behavior models*, *task anticipation*, and *knowledge acquisition*. This chapter introduces the fifth and final component, *explanation generation*, which enables an ad hoc agent to explain the reasoning behind its decisions and beliefs, promoting transparency. We begin by describing KAT’s approach for facilitating transparency by providing explanations of its behavior as answers to different types of questions (Section 7.2). We then detail the evaluation process used to assess the effectiveness of the explanations generated (Section 7.3). Finally, we discuss and analyze the results obtained during the evaluation (Section 7.4).

7.1 General Overview

In AHT, agents must collaborate on-the-fly with unfamiliar teammates, without prior coordination or shared protocols. While recent methods for AHT have leveraged powerful learning based methods to model and adapt to teammate behavior, these approaches are usually opaque, offering limited insight into how or why an agent made a decision. This lack of transparency poses challenges for interpretability, trust, and debugging. In contrast, KAT supports transparency in decision-making by enabling an ad hoc agent to provide on-demand *explanations*, as responses to questions on its own decisions, and those of other agents. As discussed in Section 3.2, ability of an automated decision-making system to reliably answer questions about its decisions and beliefs promotes acceptability, and has been indicated to play an important role in human reasoning and learning as well.

Unlike methods in the existing literature that seek to make an entire learned model

interpretable, or to explain (or justify) all the choices made by a reasoning system, in our architecture we focus on quickly identifying the relevant information and use them to construct relational descriptions in response to different types of questions on demand. KAT's use of knowledge-based reasoning and simple predictive models provide the foundation to support this desired transparency in the ad hoc agent's decisions and beliefs, and those of other agents in the domain. Also, our approach for generating the desired descriptions 'on-demand' promotes computational efficiency.

The underlying principle is that of *explainable agency*, which corresponds to enabling specific functional abilities Langley (2019); Langley et al. (2017). Specifically, the agent must be able to: (i) provide on-demand justification of decisions made during (or after) plan generation and execution by considering alternative choices; (ii) report actions executed and present information at a suitable level of abstraction; (iii) describe how actual events deviated from a computed plan and how it adapted to these deviations; and (iv) communicate information about decisions and justification such that it makes contact with human concepts such as beliefs and goals.

KAT is designed to represent and reason with knowledge at different abstractions. We build on this representation and the associated update processes to embed the principle of explainable agency, i.e., to implement the abilities mentioned above, enabling the ad hoc agent to generate relational descriptions in response to four types of questions identified as being important in work on explainable planning Fox et al. (2017), that is *causal*, *contrastive*, *belief justification* and *counterfactual* questions.

In the following section, we describe in detail the procedure the ad hoc agent follows to generate responses for each of these types of questions.

7.2 Generating Relational Descriptions

The *explanation generator* component of KAT (introduced in Chapter 4, Figure 4.1), is used for constructing the explanations. This component consists of two subcomponents, the *text converter* and the *analyzer*. The *text converter* processes the questions from human or other agents in the domain and extract information such as question type and any relevant literals present in the question. If Q represents the space of questions, τ represents the question type ($\tau \in \text{causal, contrastive, belief, counterfactual}$), and Θ represents the set of relevant terms extracted from the question, the text

converter's parsing function is:

$$parse : Q \rightarrow (\tau, \Theta) \quad (7.1)$$

For example, for a causal question “Why did you execute action a_I at step I ?”, $\Theta = \{a_I, I\}$. This information is then passed to the *analyzer* which identify and extracts the relevant axioms, and literals from the Answer Set Prolog program and the answer set, that needs to construct the explanation.

$$analyze : (\tau, \Theta, \Pi(\mathcal{D}, \mathcal{H}), \mathcal{AS}) \rightarrow E \quad (7.2)$$

Depending on the value of τ , the analyzer will invoke relevant algorithm to compute E as:

$$E = \begin{cases} \text{Algorithm 9}(\Theta, \Pi(\mathcal{D}, \mathcal{H}), \mathcal{AS}) & \text{if } \tau = \text{causal} \\ \text{Algorithm 10}(\Theta, \Pi(\mathcal{D}, \mathcal{H}), \mathcal{AS}) & \text{if } \tau = \text{contrastive} \\ \text{Algorithm 11}(\Theta, \Pi(\mathcal{D}, \mathcal{H}), \mathcal{AS}) & \text{if } \tau = \text{belief} \\ \text{Algorithm 12}(\Theta, \Pi(\mathcal{D}, \mathcal{H}), \mathcal{AS}) & \text{if } \tau = \text{counterfactual} \end{cases} \quad (7.3)$$

E is then passed back to the *text converter* to be inserted in to a generic response template and create the human readable explanation as answer to that question. It maps E to natural language response using fixed template T_τ associated with question type τ :

$$generate : (\tau, E, T_\tau) \rightarrow s \quad (7.4)$$

where s is the final explanation, formed by inserting the literals of E (and, where applicable Θ) into T_τ .

Below, we begin by describing the procedure our ad hoc agent follows to generate explanations for **causal** questions.

7.2.1 Causal Questions

Causal questions aim to explain the reasoning behind an agent's action choices, and take the form “Why did you execute action a_I at step I ?”. Here I is a step in the agent's plan, and a_I is the action selected by the agent to execute at step I . Upon receiving such a query, the ad hoc agent first checks whether action a_I is the last action of plan $\mathcal{P}$ executed by the agent (Line 1, Algorithm 9).

If not, the agent extracts the actions $\{a_{af}\} \in \mathcal{P}$ that occurred immediately after $a_I \in \mathcal{P}$ (Line 2). Next, the agent examines the program $\Pi(\mathcal{D}, \mathcal{H})$ to identify any axioms

with the negation of $a_i \in \{\mathcal{A}_{af}\}$ in *head*, i.e., axioms encoding conditions that prevent a_i from occurring (Line 3). The agent then checks if each such selected axiom's *body* is satisfied by the *answer set* at time step I (Line 5). If yes, it indicates that action a_i could not be executed unless the fluent literals f in the *body* of the axiom change value. The agent stores the fluent literals f in *body* whose value changed from I to $I + 1$ (Line 6). The value of all such literals $\{\mathcal{F}\}$ over all identified axioms has been changed by the execution of a_I , the action under consideration. The agent uses these literals to answer the question about a_I (Line 7).

Algorithm 9: Generate Causal Explanations

Input: $\mathcal{P}$: Plan; a_I : action; $\mathcal{G}$: goal; $\Pi(\mathcal{D}, \mathcal{H})$: program; $\mathcal{AS}$: answer set

Output: $\mathcal{E}$: Explanation literals

```

1  if  $a_I$  is not the last action in  $\mathcal{P}$  then
2       $\mathcal{A}_{af} \leftarrow \{a_i \in \mathcal{P} \mid i > I\}$ 
3       $\mathcal{R} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid \text{head}(r) = \neg a_i, a_i \in \mathcal{A}_{af}\}$ 
4      for  $r \in \mathcal{R}$  do
5          if  $\text{body}(r)$  satisfied in  $\mathcal{AS}$  at  $I$  then
6               $\mathcal{F} \leftarrow \{f \in \text{body}(r) \mid \text{val}(f, I) \neq \text{val}(f, I + 1)\}$ 
7               $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{F}$ 
8          end
9      end
10 else
11      $\mathcal{R} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid a_I, f' \in \text{body}(r)\}$ 
12      $\mathcal{F}' \leftarrow \{f' \mid f' \in \text{body}(r), r \in \mathcal{R}\}$ 
13     for  $l \in \mathcal{F}'$  do
14          $\mathcal{C} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid \text{head}(r) = l\}$ 
15         for  $r \in \mathcal{C}$  do
16             if  $\text{body}(r)$  satisfied in  $\mathcal{AS}$  at  $I$  then
17                  $\mathcal{E} \leftarrow \mathcal{E} \cup \{l \cup \text{body}(r)\}$ 
18             end
19         end
20     end
21      $\mathcal{E} \leftarrow \mathcal{E} \cup \mathcal{G}$ 
22 end
23 return  $\mathcal{E}$ 

```

If a_I is the last action in $\mathcal{P}$, the agent examines $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with a_I in their *body* alongside other additional literals $\{\mathcal{F}'\}$, i.e., axioms encoding effects of a_I if additional conditions are true (Line 11). The agent then extracts these literals $\{\mathcal{F}'\}$ from the identified axioms and examines $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with literal $l \in \{\mathcal{F}'\}$ in their *head*, i.e., state constraints (Lines 12 - 14). The agent then checks whether each of the selected axiom's *body* is satisfied by the *answer set* at time step I (Line 16). If yes, the agent use the literal $l \in \{\mathcal{F}'\}$ and the *body* literals of the axioms to construct the answer as they have prompted the agent to execute a_I . Furthermore, as the last action in $\mathcal{P}$, a_I directly contributed toward achieving the goal, and the agent also uses the goal G terms to construct the answer.

Next, we provide a brief argument for the soundness of Algorithm 9.

The loops in Lines 4 and 13 are guaranteed to terminate after a finite number of iterations as the set of actions in $\mathcal{A}_{af}$, the number of axioms in $\Pi(\mathcal{D}, \mathcal{H})$ and the number of literals in the body of any selected axiom are finite. By verifying that the body of each selected axiom is satisfied by the answer set ($\mathcal{AS}$) at given step, the algorithm ensures that only relevant axioms are used to generate the explanations. If no valid explanations were found the algorithm will return null.

To illustrate the explanation generation procedure for causal questions, we present two scenarios from VirtualHome (when the queried action is not the final action and followed by other action steps) and Half Field Offense (when the queried action is the final action of the agent's plan) domains.

Example 10 [*Non-terminal Action — Agent Assisting in Preparing Breakfast*]

Consider the following scenario from the VirtualHome domain, in which the ad hoc agent is assisting the human in preparing breakfast: the cake is inside the microwave, the microwave is closed and switched off, and the ad hoc agent is inside the kitchen. To help the human prepare breakfast, the ad hoc agent generates the following plan:

```
occurs(move(ahagent, kitchen_smalltable),5)
occurs(switchoff(ahagent, microwave),6)
occurs(open(ahagent, microwave),7)
occurs(grab(ahagent, cake),8)
occurs(move(ahagent, kitchentable),9)
occurs(put(ahagent, cake, kitchentable),10)
```

We present here only the part of the plan that is relevant to this example; the remaining steps will be discussed later in this section in the context of a different example. When

queried “*Why did you open the microwave in step 7?*” the agent first extracts all the actions that occurred after queried action, $occurs(open(ahagent, microwave), 7)$. Next, it retrieves the axioms that have the negation of those actions as their head. An example of such executability conditions is:

$$\neg occurs(grab(R, G), I) \leftarrow holds(on(G, E), I), \neg holds(opened(E), I), \quad (7.5a) \\ \#agent(R), \#graspable(G), \#electrical(E).$$

which implies that the agent cannot retrieve a graspable object from inside an electrical appliance if the appliance’s door is closed.

The agent then suitably grounds the variables in the body of the axiom (R with $ahagent$, G with $cake$, E with $microwave$, and I with 7) and checks whether the grounded body of the axiom is satisfied by the answer set at the queried step 7. Since $holds(on(cake, microwave), 7)$ is true and $holds(opened(microwave), 7)$ is false, satisfying the axiom, the agent checks whether the values of those literals have been changed in the subsequent step (step 8). At step 8, the value of $holds(on(cake, microwave), 8)$ remains the same (true); however, the value of $holds(opened(microwave), 8)$ has changed (true). This indicates that the queried action, $occurs(open(ahagent, microwave), 7)$, changed the value of the literal. The agent uses these body literals to construct the explanation “*I opened the microwave in step 7 since the cake was inside and the door was closed*”.

Example 11 [*Terminal Action — Agent Playing Soccer as a Offense Team Member*]

Consider a scenario from the Half Field Offense domain, where the ad hoc agent is playing as an offense team member. The agent has possession of the ball, and its goal is to score. To achieve this, the agent generates the following plan:

$$occurs(pass(ahagent, offense2), 0)$$

When queried about the last action of the agent’s plan, “*Why did you pass the ball to offense2 in step 0?*”, the agent first extracts all axioms that have the queried action in their body along with any additional literals (preconditions). An example of such a causal law is:

$$holds(achieve_goal, I + 1) \leftarrow occurs(pass(R, O), I), \#agent(R), \quad (7.6a) \\ \#teammate(O), holds(pass_conditions_satisfied(O), I).$$

This implies that the agent may consider passing the ball to its teammate if the passing conditions are met. The agent then extracts the additional literals from the body of the causal law and searches for axioms that have these literals in their head. An example of such a state constraint is:

$$\begin{aligned} \text{holds}(\text{pass_conditions_satisfied}(O), I) \leftarrow & \text{holds}(\text{def_too_close}(\text{ahagent}), I), \quad (7.7a) \\ & \text{holds}(\text{shooting_angle}(V), I), V < 19, \\ & \#teammate(O). \end{aligned}$$

This indicates that when an offense team member is in close proximity to the ad hoc agent, and the agent's shooting angle falls below a predefined value, it should consider passing the ball to its teammate. Next, the agent checks whether the body of the above state constraint is satisfied by the answer set at the queried step (step 0). Since it is satisfied, the agent uses these literals to construct the answer “*I passed the ball to offense2 since the defense agents were too close and I did not had a good shooting angle*”.

Next, we describe the procedure the ad hoc agent follows to generate explanations for **contrastive** questions.

7.2.2 Contrastive Questions

Contrastive questions take the form of “*Why did you not execute action a at step I ?*” and seek to reveal the reasoning behind an agent's decision not to select a particular action. To answer such a query, the ad hoc agent first examines the program $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with the negation of a_I in its *head*, i.e., executability conditions (Line 1, Algorithm 10). For each selected axiom, the agent checks whether the axiom's *body* is satisfied by *answer set* at I (Lines 2-3). If yes, the agent collects the fluent literals $\{f\}$ in the *body*, as they prevented consideration of a_I , and uses them to construct the explanation (Line 4).

Algorithm 10: Generate Contrastive Explanations**Input:** $\mathcal{P}$: Plan; a_I : action; $\mathcal{G}$: goal; $\Pi(\mathcal{D}, \mathcal{H})$: program; $\mathcal{AS}$: answer set**Output:** $\mathcal{E}$: Explanation literals

```

1  $\mathcal{R} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid head(r) = \neg a_I\}$ 
2 for  $r \in \mathcal{R}$  do
3   if  $body(r)$  satisfied in  $\mathcal{AS}$  at  $I$  then
4      $\mathcal{E} \leftarrow \mathcal{E} \cup \{f \mid f \in body(r)\}$ 
5   end
6 end
7 if  $\mathcal{E} = \emptyset$  then
8    $\mathcal{R}' \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid a_I, f' \in body(r)\}$ 
9    $\mathcal{F}' \leftarrow \{f' \mid f' \in body(r), r \in \mathcal{R}'\}$ 
10  for  $l \in \mathcal{F}'$  do
11     $\mathcal{C} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid head(r) = l\}$ 
12    for  $r \in \mathcal{C}$  do
13      if  $body(r)$  not satisfied in  $\mathcal{AS}$  at  $I$  then
14         $\mathcal{E} \leftarrow \mathcal{E} \cup \{l \cup body(r)\}$ 
15      end
16    end
17  end
18 end
19 if  $\mathcal{E} = \emptyset$  then
20    $\mathcal{C} \leftarrow cost(\mathcal{P})$ 
21    $\mathcal{C}' \leftarrow cost(\mathcal{P}' = \mathcal{P} \cup a_I)$ 
22   if  $\mathcal{C}' > \mathcal{C}$  then
23      $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{C}, \mathcal{C}'\}$ 
24   end
25 end
26 return  $\mathcal{E}$ 

```

If no fluent literals have been identified, the agent examines $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with a_I in its *body* alongside other literals $\{f'\}$, i.e., causal laws (Line 8). The agent extracts these additional precondition literals $\{\mathcal{F}'\}$ and examines $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with $l \in \{\mathcal{F}'\}$ in its *head*, i.e., state constraints (Lines 9-11). For each selected axiom, the agent then checks if the axiom's *body* is satisfied by *answer set* at

step I (Lines 12-13). If not, it uses the literals in the *body* of the axiom, and the literal l , to construct the answer as they prevented the agent from executing action a_I (Line 14).

If the literals to construct an answer have not been identified so far, the agent computes the cost C of $\mathcal{P}$ to achieve the goal $\mathcal{G}$, and the cost C' of the $\mathcal{P}'$ that also include a_I (Lines 20-21). If the cost of $\mathcal{P}'$ is higher than that of $\mathcal{P}$ (i.e., $C' > C$), the agent concludes that this prevented it from including a_I in its plan (Line 22). The agent uses the cost terms to construct the answer (Line 23).

The loops in Lines 2, 10 and 12 range over the finite set of axioms in $\Pi(\mathcal{D}, \mathcal{H})$, and are thus guaranteed to terminate. A literal is considered a valid explanation only if the body of its corresponding axiom is satisfied by the answer set ($\mathcal{AS}$) at the relevant step. The algorithm evaluates multiple strategies to identify a justification for why the queried action was not selected. If no valid explanation is found, the algorithm returns null rather than generating fabricated explanations.

Below, we illustrate the procedure for generating explanations for contrastive questions by building on the previous examples from the VirtualHome and Half Field Offense domains.

Example 12 [*Agent Assisting in Preparing Breakfast*]

In our earlier example scenario from VirtualHome, the agent chose to open the microwave in step 7. This decision was justified by the agent, who explained that it believed the cake was inside the microwave and that the door was closed, making it necessary to open the door to proceed with the rest of the task. Given this explanation, a follow-up question would be: if opening the microwave was required, “*Why did you not open the microwave in step 6?*”.

To answer this question, the agent first extracts the axioms that have the negation of the action $occurs(open(ahagent, microwave), I)$ in their head. One such executability condition is:

$$\neg occurs(open(R, E), I) \leftarrow holds(switchedon(E), I), \quad (7.8a) \\ \#agent(R), \#electricals(E).$$

which implies that the agent cannot open the door of an electrical appliance if it is switched on. The agent then checks whether the axiom’s body literals, $holds(switchedon(microwave), 6)$, are satisfied by the answer set at step 6. Since it is, the agent uses these literals to construct the explanation “*I did not open the microwave at step 1 since the microwave was switched on*”.

Example 13 [*Agent Playing Soccer as an Offense Team Member*]

In the previous scenario from the Half Field Offense domain, the ad hoc agent chose to pass the ball to its teammate instead of shooting to score. Thus, consider the follow-up contrastive question to the agent, “*Why did you not shoot to score in step 1?*”.

To answer this question, the agent first extracts the axioms that have the negation of the action $occurs(shoot(R), I)$ in their head. There are two such executability conditions:

$$\neg occurs(shoot(R), I) \leftarrow \neg holds(has_ball(R), I), \#agent(R). \quad (7.9a)$$

$$\neg occurs(shoot(R), I) \leftarrow holds(far_from_goal, I), \#agent(R). \quad (7.9b)$$

Statement 7.9(a) implies that the agent cannot score if it does not possess the ball, and statement 7.9(b) indicates that the agent is unable to score if it is positioned far away from the goal. However, in this scenario the agent does possess the ball and it is positioned near the goal, thus neither statement is satisfied by the answer set at step 1. Thus, the agent searches for axioms that have the queried action in their body along with additional literals. One such causal law is:

$$holds(score_goal, I + 1) \leftarrow occurs(shoot(R), I), \quad (7.10a)$$

$$holds(agent_angle(V), I), V \geq 19.$$

The agent then checks the answer set to determine whether the body of the selected causal law $holds(agent_angle(V), I), V \geq 19$ is satisfied. Since the agent’s shooting angle is below the threshold value, the body of the axiom is not satisfied. Thus, the agent uses this information to generate the explanation “*I did not shoot in step 1 since I did not have a good shooting angle*”.

In the next section, we describe the procedure the ad hoc agent follows to generate explanations to **justify** its **beliefs**.

7.2.3 Justify Beliefs

Algorithm 11: Generate Belief Justifications

Input: l_I : belief; $\Pi(\mathcal{D}, \mathcal{H})$: program; $\mathcal{AS}$: answer set

Output: $\mathcal{E}$: Explanation literals

```

1  $\mathcal{R} \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid head(r) = l_I\}$ 
2 for  $r \in \mathcal{R}$  do
3   if  $body(r)$  satisfied in  $\mathcal{AS}$  at  $I$  then
4      $\mathcal{E} \leftarrow \mathcal{E} \cup \{f \mid f \in body(r)\}$ 
5      $\mathcal{F} \leftarrow \{f \mid f \in body(r)\}$ 
6   end
7 end
8 if  $belief\_tracing = true$  then
9    $Tree \leftarrow$  create root node  $l_I$ 
10   $Tree \leftarrow$  branches  $(\mathcal{R}, \mathcal{F})$ 
11  for  $f \in \mathcal{F}$  do
12     $\mathcal{R}' \leftarrow \{r \in \Pi(\mathcal{D}, \mathcal{H}) \mid head(r) = f\}$ 
13    for  $r \in \mathcal{R}'$  do
14      if  $body(r)$  satisfied in  $\mathcal{AS}$  then
15         $\mathcal{F} \leftarrow \mathcal{F} \cup \{f \mid f \in body(r)\}$ 
16         $Tree \leftarrow$  branches  $(r, \mathcal{F})$ 
17      end
18    end
19  end
20 end
21 return  $\mathcal{E}, Tree$ 

```

Belief justification questions aim to understand the reasoning behind an agent's beliefs. They take the form “*Why did you believe l at step I ?*”, where l is a literal which represent a specific belief held by the agent at step I . In response to such a query, the ad hoc agent first replaces the grounded terms of the belief l_I with their corresponding variables and examines $\Pi(\mathcal{D}, \mathcal{H})$ to identify axioms with l_I in *head*, i.e., state constraints (Line 1, Algorithm 11). There can be multiple such state constraints with literal l_I as their *head*. The agent identify and retrieves all such constraints. The agent then checks whether each selected axiom's *body* is satisfied by the answer set at step I (Lines 2-3). If yes, it collects the fluent literals $\{f\}$ in the *body* as they support belief l_I and

uses them to construct the answer (Line 4). If multiple axioms are identified, the agent selects one of them to provide the explanation.

If belief tracing is enabled, the agent creates a tree with l_I as its root node and each axiom (identified in Line 1) as a branch (Lines 8-10). With the axiom, the agent also stores the supporting fluents ($\mathcal{F}$). It then repeats this procedure for each fluent literal in F as target belief until no more axioms are identified (Lines 11-19). The paths of the belief tree are then used to construct the explanation.

Next, we provide a brief argument for the soundness of Algorithm 11. As with the previous algorithms, the axioms are drawn from $\Pi(\mathcal{D}, \mathcal{H})$, and thus the loops in Lines 2 and 13 are guaranteed to terminate. Belief tracing is performed only when explicitly requested (i.e., when the tracing flag is set to `true`), allowing the user to choose between receiving only the most relevant explanation or an explanation tree that recursively traces the beliefs supporting the queried belief. As the step index cannot go below step 0, the recursion in Line 11 is guaranteed to terminate, eventually reaching literals with no further supporting axioms. Only literals whose corresponding axiom bodies are satisfied by the answer set ($\mathcal{AS}$) are added to the explanation tree, ensuring every path in the tree holds in $\mathcal{AS}$ and is a valid justification for the target belief l_I .

We illustrate the ad hoc agent's explanation generation procedure for justifying beliefs using the following example. To maintain consistency and clarity, we continue using the same plan from the previous example in the VirtualHome domain. However, unlike the earlier cases of causal and contrastive questions, where the agent generated explanations after executing the plan, in the following example for belief justification, questions are posed before the plan is executed.

Example 14 [*Agent Assisting in Preparing Breakfast*]

Continuing with the same scenario from the VirtualHome domain, below is a complete version of the plan generated by the agent to help the human prepare breakfast.

```
occurs(move(ahagent, kitchen_smalltable),0)
occurs(switchon(ahagent, microwave),1)
occurs(grab(ahagent, cereal),2)
occurs(move(ahagent, kitchentable),3)
occurs(put(ahagent, cereal, kitchentable),4)
occurs(move(ahagent, kitchen_smalltable),5)
occurs(switchoff(ahagent, microwave),6)
occurs(open(ahagent, microwave),7)
```

$occurs(grab(ahagent, cake), 8)$
 $occurs(move(ahagent, kitchentable), 9)$
 $occurs(put(ahagent, cake, kitchentable), 10)$

According to the generated plan the ad hoc agent intends to switch off the microwave in step 6, based on its belief that the cake was sufficiently heated (ready to eat). This leads to the following belief justification question, “*Why did you believe the cake was heated in step 6?*”. To answer this question, the agent first extracts the axioms which has $heated(G), I$ as their head. One such state constraint is:

$$holds(heated(F), I) \leftarrow holds(on(F, S), I), holds(switched_on(S), I), \quad (7.11a)$$

$$\#cooking_electricals(S), \#food(F).$$

which implies that if a food item is inside a cooking appliance and the appliance is turned on, then the food becomes heated. The ad hoc agent then examines whether the body of the selected state constraint is satisfied by the answer set. Since the cake was placed inside the microwave in step 6, and the microwave was turned on, both literals $holds(on(cake, microwave), 6), holds(switched_on(microwave), 6)$ in the body of the axiom are satisfied. This allows the agent to use these literals to construct an explanation as an answer to the query.

However, since belief tracing is enabled in this scenario, instead of providing the explanation here, the agent constructs a belief tree with $holds(heated(cake), 6)$ as its root node. The second level of the tree consists of the axioms that have the queried literal $holds(heated(cake), 6)$ as their head. Since there is only one such axiom, this level of the tree contains a single node representing the axiom 7.11, as shown in Figure 7.1. As mentioned previously, state constraint 7.11 is satisfied by the answer set. Thus, the agent uses the literals $holds(on(cake, microwave), 6), holds(switched_on(microwave), 6)$ as extended beliefs to initiate the next level of the belief tree. Each of these literals acts as a node in the belief tree. Thus, the third level of the tree consists of the axioms that has these extended beliefs in their head, such as:

$$holds(switched_on(A), I + 1) \leftarrow occurs(switch_on(R, A), I). \quad (7.12a)$$

$$holds(on(O, S), I + 1) \leftarrow occurs(put(R, O, S), I), O! = S. \quad (7.12b)$$

where Statement 7.12a implies that the agent switching on an appliance causes the appliance to be switched on; and Statement 7.12b indicates that placing an object on a surface causes the object to be on that surface. The agent then examines whether these axioms are satisfied by the answer set.

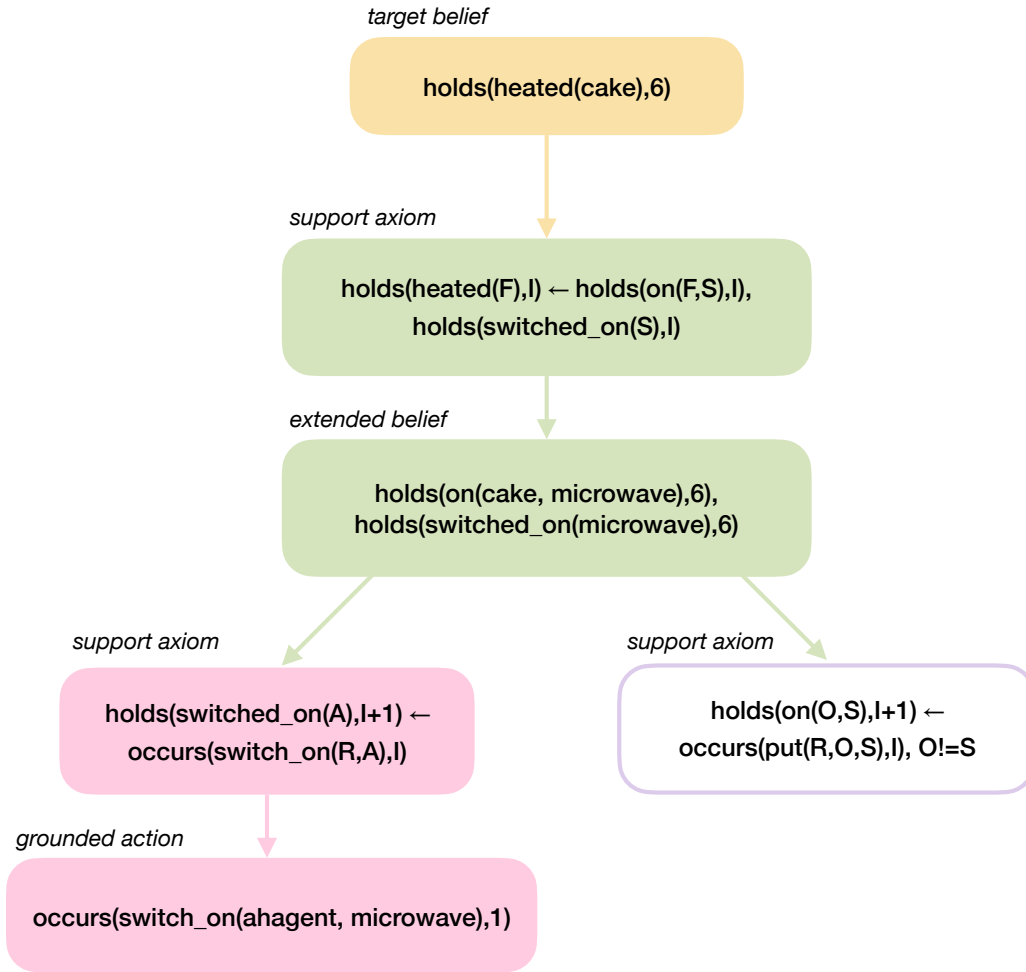


Figure 7.1: Belief tracing to explore why the ad hoc agent believed the cake was heated in step 6.

According to the generated plan, the agent switch on the microwave is step 1, resulting in statement 7.12a being satisfied by the answer set. Thus, the agent adds the corresponding grounded action to the tree. However, the agent lacks information about when the cake was placed inside the microwave, making the current answer set unable to satisfy statement 7.12b. Thus, statement 7.12b is not explored further. Since the remaining branch has also reached a grounded action and cannot be expanded further, the agent uses the literals along the paths of the tree to construct the explanation: “*I believed the cake was heated since I observed that the cake was on the microwave and the microwave was switched on in step 1*”.

7.2.4 Counterfactual Questions

Counterfactual questions explore *what if* scenarios by querying how outcomes would change if the action choices of the agents were different. They investigate alternative possibilities and take the form of “*What will be the outcome if agent R (or you) execute a'_I in (future) step I ?*” and “*What will be the resultant world state at (future) step I if the agent R executes actions predicted by learned models?*”. To answer such queries, the ad hoc agent first retrieves the most recent state observation of environment S_{I-n} in relation to step I , i.e., start with the current best estimate of world state (Line 1, Algorithm 12). It then uses the predictive behavior models of other agents to retrieve their action $\mathcal{A}_{I-n}^{ot}$ for step $I-n$; it also uses KAT to compute the action a_{I-n} that it should execute (Lines 2-4). The agent then performs a mental simulation of the immediate future step $I-n+1$ from current state of the world S_{I-n} using existing knowledge and action choices identified $(\mathcal{A}_{I-n}^{ot}, a_{I-n})$ (Line 5). The agent repeats above n times, i.e., roll out the future and explore effects of the action of ad hoc agent and human until environment reaches the queried step I . It then collects the state information S_I at I (Line 7). The agent then feeds the state information S_I to the predictive behavior model of the agent R to retrieve the action a_I^o , and use KAT to retrieve the action for the ad hoc agent a_I in state S_I (Line 7-9). Specifically, the agent traverses the FF tree model of the human (FF_h) to identify branches that were active in selecting a_I^o ; these branches and a_I^o are used to construct the answer (Line 10).

If the counterfactual question is about the outcome of a specific action a' executed by the other agent R at time step I (i.e., *What will be the outcome if agent R executes a' in future step I ?*), this action is used directly instead of the predicted action Line 8 of Algorithm 12. In a similar manner, if the question is about the outcome of a hypothetical action by the ad hoc agent at a specific future time step (i.e., *What will be the outcome if you executes a' in future step I ?*), this action is considered in Line 9. In addition, actions at a specific time step in the past can also be considered by examining the recorded history and then performing a mental simulation.

When simulating each step using the transition function T , the algorithm applies the same behavior models and the same action-selection procedure (KAT) used by the ad hoc agent during actual execution. This ensures the simulated trajectory closely matches actual execution, reproducing what KAT and other agents would execute if the plan were rolled forward. Thus, the predicted action and the active branches identified are consistent with the agent’s actual decision-making procedure.

Algorithm 12: Generate Counterfactual Explanations

Input: I : future step; S_{I-n} : recent observation; FF : FF tree behavior models of other agents; $\mathcal{A}$: other agents.

Output: $\mathcal{E}$: Explanation literals

```

1  $S \leftarrow S_{I-n}$ 
2 for  $t = I - n$  to  $I - 1$  do
3    $\mathcal{A}_t^{ot} \leftarrow \{FF_j(S) \mid j \in \mathcal{A}\}$ 
4    $a_t \leftarrow \text{KAT}(S)$ 
5    $S \leftarrow T(S, \mathcal{A}_t^{ot}, a_t)$ 
6 end
7  $S_I \leftarrow S$ 
8  $\mathcal{A}_I^o \leftarrow \{FF_j(S_I) \mid j \in \mathcal{A}\}$ 
9  $a_I \leftarrow \text{KAT}(S_I)$ 
10  $\mathcal{E} \leftarrow$  active branches in  $FF_j$  for  $a_I^j \in \mathcal{A}_I^o$ 
11 return  $a_I^j, \mathcal{E}$ 

```

We illustrate the above procedure the ad hoc agent uses to generate explanations for counterfactual questions using the following example. Note that similar to the previous example for belief justification questions, the counterfactual questions are also posed before the plan was executed.

Example 15 [*Agent Assisting in Preparing Breakfast*]

In the earlier example, we observed the plan generated by the ad hoc agent. However, the specific actions that the human agent will execute to achieve the shared goal are unknown to the ad hoc agent. To collaborate effectively, the ad hoc agent must anticipate and incorporate the human's actions in its planning. This motivates the following question to the agent, “*What do you think the human will do in step 2?*”.

To answer this question, the agent first retrieves its current best estimate of the state of the world, which is the state at step 0. Then it uses the predictive behavior model of the human agent to estimate the human's action at step 0. The agent also uses KAT architecture to determine its own action for step 0. The agent then performs a mental simulation to obtain the world state at step 1. At step 1, the agent repeats this process: it predicts the human's action using the behavior model, retrieves its own action from KAT, and simulates the resulting state at step 2. And finally, at step 2, the agent uses the human behavior model one more time to estimates the human's action. In our example, the predicted sequence of human actions were as follows:

- Step 0: move to the kitchen counter containing the stove,
- Step 1: place the cutlets on the frying pan on the stove, and
- Step 2: switch on the stove.

The agent uses the predicted actions for step 2 to generate the explanation “*The human will decide to switch on the stove in step 2*”.

This leads to a follow-up question: “*Why do you think the human will switch on the stove in step 2?*”. To answer this, the agent uses the predictive behavior model decision tree to identify active branches corresponding to the predicted action. Using this information, the agent constructs the explanation: “*I believe the human will switch on the stove in step 2 since in the previous step the human placed the cutlets on the frying pan and the human’s pose along the z-axis was less than 2.36*”.

In the next section, we describe the experimental setup used for evaluating the agent’s ability to generate explanations in Fort Attack, Half Field Offense and VirtualHome domains and measure the accuracy and reliability of generating those explanations.

7.3 Experimental Procedure

We evaluated the explanation generation capabilities of our ad hoc agent in the three simulated benchmark domains Fort Attack, Half Field Offense and VirtualHome (Section 4.7.1). Below, we outline the procedure used to indicate how the interplay between representation, reasoning, and learning enables reliable and efficient explanation generation.

Fort Attack

Similar to the previous experiments in the Fort Attack domain, each episode consisted of three guards and three attackers, and the ad hoc agent represented one of the guard team members. Each other agent in the team (guard/attacker) was assigned a policy from the built-in policies described in Section 4.7.2. We ran 20 game episodes and randomly sampled 10 instances of state observation and agent available domain knowledge from different game episodes in the Fort Attack domain.

In **Exp1**, we posed 25 different questions (belonging to the four types of questions described before) to the agent from each sampled plan and recorded the precision and recall of retrieving relevant literals for constructing answers to these questions as

explanations. A relevant literal is any literal that would be expected to appear in an explanation, for that explanation to be considered correct and complete, given the domain knowledge available to the agent. We determined the ground truth set of relevant literals for each question by considering the agent’s available knowledge. For calculating precision and recall we used the following formulae:

$$Precision = \frac{TP}{TP + FP} \quad (7.13)$$

$$Recall = \frac{TP}{TP + FN} \quad (7.14)$$

where TP (true positives) are correct, relevant literals returned by the agent, FP (false positives) are literals returned by the agent that are not relevant (i.e., not in the ground truth explanation), and FN (false negatives) are relevant literals that the agent failed to return. We evaluated and scored each generated explanation manually against the corresponding ground truth literals.

We note that evaluating other qualities of an explanation, such as simplicity or usefulness, typically require human-participant studies, however, these were considered outside the scope of this work given the time and controlled-setup constraints involved. We instead focused on measuring the faithfulness of an explanation to the agent’s own knowledge and the domain dynamics, which precision and recall are well suited to capture.

Next, to evaluate the effect of learned knowledge on explanation quality, in **Exp2** we repeated the same experiment after removing the three axioms (Statements 6.2) that the agent learned in the previous Section 6.4 from the knowledge base. Thus, in this setting, the agent operated with less knowledge than in **Exp1**.

Half Field Offense

For the experiments in the Half Field Offense domain we used the external agent teams from 2013 RoboCup simulation competitions, *Helios*, *Gliders*, *Cyrus*, *Axiom* and *Aut* as offense team members, and *Agent2d* as the defense team (Section 4.7.2). The ad hoc agent represented one of the offense team members.

We followed the same experimental setup as in the Fort Attack domain and posed 16 different questions to the agent, based on 10 instances of state observations and agent’s domain knowledge, randomly selected from 20 game episodes (**Exp3**). We then evaluated the agent’s ability to retrieve relevant literals for generating the explanations as answers to these questions.

As an extension to **Exp3**, in **Exp4**, we removed the axioms (Statements 6.3) the agent learned in Section 6.4 from its knowledge base and repeated the **Exp3**.

Virtual Home

In the VirtualHome domain, each episode consisted of a human agent that chooses its actions based on an Answer Set Prolog program, and our ad hoc agent following KAT. To evaluate the ad hoc agent's ability to generate explanations of its decisions and beliefs and those of the human agent, we designed **Exp5**, where we randomly selected 10 configurations (from 20 episodes) and posed 32 different questions (divided between the four types of questions) about chosen steps in each trial. The agent generated answers to those questions following the procedure described in Section 7.2. We then recorded the precision and recall of retrieving literals to answer these questions.

Additionally, we provide execution traces as qualitative examples of the architecture's performance in the Fort Attack, Half Field Offense and VirtualHome domains in Section 7.4.2.

In each domain, we manually constructed each question to be posed to the agent after inspecting the corresponding plan. To maintain consistency, we used a fixed set of questions for each domain and slightly revised each question based on the plan. We tried to ask the same set of questions from every plan as much as possible; the question type remained the same, but we changed the variables according to the relevant plan. For example, if the first question posed for the first sampled plan was a causal question about a specific action, the same question type was posed for the corresponding action in every other sampled plan in that domain, unless that action was invalid in that sampled plan. For example, if the first question of plan A was "Why did you grab the water glass in step 2?", the first question of plan B could instead be "Why did you grab the bread slice in step 3?". We only altered this question sequence when the relevant action or belief was not present in a particular agent's plan; for example, the above question could not be posed to an agent whose plan did not involve a grab action, and we could not ask the agent why it believed the cutlets were cooked if the agent's plan did not involve the cutlets. In such cases, we instead asked a question involving a similar, valid action or belief present in that plan — such as asking why the agent believed the cake was heated. We made such changes rarely, only when the action or belief involved in a question was not relevant to the agent's plan.

7.4 Results

This section presents the results of the experiments described above. We first include the quantitative results (Section 7.4.2) in the three domains Fort Attack, Half Field Offense and VirtualHome, followed by qualitative results (Section 7.4.2).

7.4.1 Quantitative Results

Fort Attack

Table 7.1 summarizes the precision and recall values corresponding to **Exp1**, which evaluated the ad hoc agent's ability to generate relational descriptions as explanations of its own decisions and beliefs, as well as those of other agents. In this experiment, the knowledge base used by the agent to reason included the axioms it had learned during the *knowledge acquisition* experiments described in Chapter 6. We observed high precision and recall values, demonstrating the agent's capability to automatically extract the correct literals when providing relational descriptions as explanations in response to different types of questions.

Next, Table 7.2 presents the results of **Exp2**, where the ad hoc agent generated explanations for the same set of queries but without access to the axioms learned in the earlier *knowledge acquisition* experiments (Chapter 6). We observed lower precision and recall values, specially with a significant reduction in the recall values when compared to **Exp1**. This indicates that, under incomplete knowledge, the ad hoc agent often failed to retrieve many of the literals required to form a complete explanation. It is important to note that precision and recall were calculated relative to the ground truth. Thus, explanations that were internally consistent with the agent's own knowledge were still penalized when they diverged from the ground truth explanation. Also, the queries that were no longer valid under the agent's current plan were treated as failures.

These finding indicates that the acquired knowledge not only improves team performance, as discussed in Section 6.5, but also significantly improves the agent's ability to generate accurate and relevant explanations.

<i>Question type</i>	<i>TP</i>	<i>FN</i>	<i>FP</i>	<i>Precision</i>	<i>Recall</i>
Action justification	72	0	0	1.00	1.00
Contrastive	90	2	2	0.98	0.98
Belief justification	123	21	6	0.95	0.85
Counterfactual	110	0	0	1.00	1.00

Table 7.1: Precision and recall of retrieving relevant literals for constructing explanations in the *Fort Attack* domain with the learned axioms (Exp1), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.

<i>Question type</i>	<i>TP</i>	<i>FN</i>	<i>FP</i>	<i>Precision</i>	<i>Recall</i>
Action justification	24	49	0	1.00	0.32
Contrastive	23	69	8	0.74	0.25
Belief justification	69	75	26	0.73	0.48
Counterfactual	70	40	40	0.64	0.64

Table 7.2: Precision and recall of retrieving relevant literals for constructing explanations in the *Fort Attack* domain without the learned axioms (Exp2), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.

Half Field Offense

Table 7.3 summarizes the precision and recall values for **Exp3**, where the ad hoc agent generated explanations using a knowledge base that included the axioms learned during the *knowledge acquisition* phase (Chapter 6). Similar to Fort Attack domain, the agent indicated high precision and recall values, indicating that it was able to accurately identify the relevant literals to explain its own decisions and those of others.

Table 7.4 presents the results from **Exp4**, where the ad hoc agent generated the explanations for different queries while the learned axioms were removed from the knowledge base. Once again, we observed that the precision and recall values decreased in the absence of the acquired knowledge.

This further suggests that incorporating learned axioms enhances the ad hoc agent’s ability to generate accurate and complete explanations, regardless of the domain in which it operates.

<i>Question type</i>	<i>TP</i>	<i>FN</i>	<i>FP</i>	<i>Precision</i>	<i>Recall</i>
Action justification	44	0	0	1.00	1.00
Contrastive	93	2	3	0.96	0.98
Belief justification	54	9	5	0.92	0.86
Counterfactual	60	0	0	1.00	1.00

Table 7.3: Precision and recall of retrieving relevant literals for constructing explanations in the Half Field Offense domain with the learned axioms (Exp3), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.

<i>Question type</i>	<i>TP</i>	<i>FN</i>	<i>FP</i>	<i>Precision</i>	<i>Recall</i>
Action justification	30	14	14	0.68	0.68
Contrastive	67	28	10	0.87	0.71
Belief justification	40	23	22	0.65	0.63
Counterfactual	46	14	14	0.77	0.77

Table 7.4: Precision and recall of retrieving relevant literals for constructing explanations in the Half Field Offense domain without the learned axioms (Exp4), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.

Virtual Home

Lastly, Table 7.5 summarizes the results of **Exp5**. We observed high values of precision and recall, indicating the ability of the agent to automatically extract the correct literals to provide relational descriptions as explanations in response to different types of questions. This further indicates the generalizability of the agent’s explanation generation capability across domains.

7.4.2 Qualitative Results

This section provides some execution traces illustrating the capabilities of our KAT architecture in Fort Attack and VirtualHome domains.

<i>Question type</i>	<i>TP</i>	<i>FN</i>	<i>FP</i>	<i>Precision</i>	<i>Recall</i>
Action justification	190	0	0	1.00	1.00
Contrastive	89	1	11	0.89	0.99
Belief justification	214	14	28	0.88	0.94
Counterfactual	310	90	0	1.00	0.78

Table 7.5: Precision and recall of retrieving relevant literals for providing explanations in the VirtualHome domain (Exp5), including the number of true positives (TP), false positives (FP), and false negatives (FN) per question type.

Plan 13: Plan generated by the ad hoc agent to achieve its goal: *shoot at-attacker1*

- 1 move(ad_hoc_agent, 4, 13)
 - 2 move(ad_hoc_agent, 5, 13)
 - 3 move(ad_hoc_agent, 6, 13)
 - 4 move(ad_hoc_agent, 7, 13)
 - 5 shoot(ad_hoc_agent, attacker1)
-

Fort Attack

Consider the scenario in Figure 7.2, in which the ad hoc agent in grid (3,13) at time step 0, has the goal of shooting an attacker in grid (9,8). The agent is aware of its location and orientation in the map, as well as the pose information of the other agent in the domain and their status (active or not). Additionally, the agent also has information on how locations and directions are arranged around its current pose. Since the attacker agent is not in range for the ad hoc agent to shoot at the current step, Plan 13 was generated by the agent to achieve its goal.

Execution Example 1. [Action Justification and Contrastive]

As an example of providing relational descriptions of decisions and beliefs, consider the following exchange with the ad hoc agent after it executed this plan successfully.

- **Questioner:** “Why did you move to (7,13) in step 4?”
- **Ad hoc Agent:** “Because attacker1 was not in range for shooting and my goal was to shoot the attacker1 in step 5”.

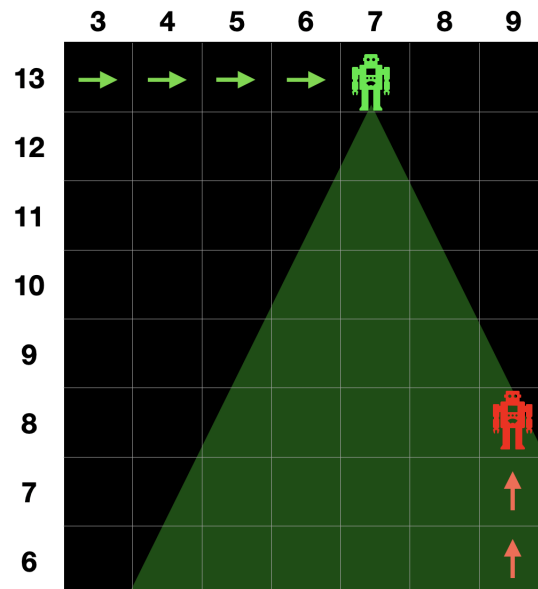


Figure 7.2: Part of the Fort Attack domain showing the ad hoc guard agent (green) moving to track and shoot an attacker (red).

The agent's response highlight both its motivation for choosing the action as well as the final objective it is trying to achieve. The agent can also be asked why it did not consider shooting the attacker (final objective) in a prior step.

- **Questioner:** "Why did you not shoot attacker1 in step 2?"
- **Ad hoc Agent:** "Because attacker1 was not in the shooting range at step 2 since it had moved towards fort in step 0 and step 1."

The agent's response explains why the suggested action was not feasible at the previous step. The agent can be further inquired on why it did not consider selecting an action that have no constraints preventing it.

- **Questioner:** "Why did you not move to (3,13) in step 2?"
- **Ad hoc Agent:** "Because that action increases the cost of the plan; new plan cost = 8, old plan cost = 7."

The response from the agent explains why the suggested action was not included in the plan, given it would increase the cost of the overall plan unnecessarily.

Execution Example 2. [Action and Belief Justification]

The agent could be further inquired on its choice of actions and beliefs as follows:

- **Questioner:** “Why did you shoot in step 5?”
- **Ad hoc Agent:** “Because my goal involved shooting attacker1 who was within shooting range at step 5.”

The agent’s response highlight its final objective of shooting the attacker1, and its belief at time step 5 that the attacker1 is within its shooting range. The agent could be further inquired on its beliefs.

- **Questioner:** “Why did you believe attacker1 was in shooting range at 5?”
- **Ad hoc Agent:** “Because I observed that the attacker1 was located in 9,9 in step 5 and I was in 7,13 facing south. This satisfied the conditions $Y2 - Y1 \leq 5, Y2 - Y1 \geq 4, X2 - 2 \leq X1, X1 \leq X2 + 2$.”

Based on the above answer, the agent could be further inquired on its beliefs about the attacker agent’s location.

- **Questioner:** “Why did you believe the attacker1 was not in 9,8 at 1?”
- **Ad hoc Agent:** “Because I observed that the attacker1 was located in 9,8 in step 0 and it moved towards fort in step 0.”

Execution Example 3. [*Counterfactuals on Hypothetical Actions*]

Given the current plan, it is useful to examine what the agent will do in future steps, or how the world may change as a result of the agent’s actions. Moreover, it is important to understand how the outcome might differ if the agent were to choose actions other than those predicted.

- **Questioner:** “What will the world look like in time step 2?”
- **Ad hoc Agent:** “The world in time step 2 will consist of 3 alive guards and 3 alive attackers. The nearest attacker to the fort will be attacker1 in location 9,10.”

The agent’s response highlights the key information it expects to observe in the inquired future step, such as the number of alive guards and attackers and their proximity to the fort. The agent could also be inquired about its predictions regarding the actions of other agents in this state.

- **Questioner:** “What do you think the guard2 will do in step 3?”

Plan 14: Part of the plan generated by the ad hoc agent to achieve its goal:

prepare breakfast

- 1 find(ad_hoc_agent, bread slice)
 - 2 grab(ad_hoc_agent, bread slice)
 - 3 find(ad_hoc_agent, kitchen table)
 - 4 put(ad_hoc_agent, bread slice, kitchen table)
 - 5 find(ad_hoc_agent, stove)
 - 6 switchon(ad_hoc_agent, stove)
-

- **Ad hoc Agent:** “The guard2 will decide to move left in step 3.”
- **Questioner:** “Why do you think the guard2 will move left in step 3?”
- **Ad hoc Agent:** “Guard2 will decide to move left since it notices attackers who are too close to the fort in left side, and the other guards are on the right side”
The agent’s response includes the possible actions it predicts other agents would take and its reasoning about why the other agent choose those actions. The agent could also be asked what will happen if the other agent chooses a different action than what is predicted.
- **Questioner:** “What will happen if guard1 shoots in step 2?”
- **Ad hoc Agent:** “If guard1 shoots in step 2, the number of alive attackers will be reduced to 1 in step 3. There will be 3 guards alive, and the nearest attacker to the fort will be attacker3.”

Virtual Home

Next, we provide execution traces illustrating the explanation generation capabilities of the ad hoc agent in the VirtualHome domain. Consider the scenario in which the: bread slice is inside toaster; cutlets are on kitchen counter; poundcake is on kitchen table; water glass is in bedroom; microwave is closed and switched off; frying pan is on stove that is switched off; and the human and ad hoc agent are in the kitchen.

To help the human prepare breakfast, the ad hoc agent generated a plan with 23 steps, some of which are shown in Plan 14; the agent expects the human to complete some intermediate steps.

Execution Example 1. [*Action Justification, Contrastive, Counterfactual*]

Consider the following exchange with the ad hoc agent after executing the first plan step.

- **Questioner:** “Why did you find the bread slice in step 0?”
- **Ad hoc Agent:** “Because I had not found the bread slice yet and I wanted to grab it in step 1”.

The agent’s response draws attention to the target action’s outcome being a requirement for executing a subsequent action. The agent can also be asked why it did not consider picking up a different object.

- **Questioner:** “Why did you not find the water glass in step 0?”
- **Ad hoc Agent:** “Because I predicted that the human will find the water glass in step 0.”

The response from the agent includes the possible future actions it expects the other agent (human) to perform. The agent may further be inquired about the human’s action choices.

- **Questioner:** “Why do you think human will grab water glass in step 1?”
- **Ad hoc Agent:** “Because my prediction is that the human wants to bring the glass to the table.”
- **Questioner:** “What if the human decided to grab the cutlets in step 1?”
- **Ad hoc Agent:** “If the human grabs the cutlets in step 1, they will be in the human’s hands in step 2.”

To answer questions about hypothetical situations, the ad hoc agent has to simulate the evolution of state and the execution of actions by the human and the agent for a few steps in order to identify and report the motivation for specific action choices.

Execution Example 2. [*Action Justification, Contrastive, Belief Tracing*]

The accuracy of the model predicting human action choices (85% in the VirtualHome domain) indicates that the predictions can be incorrect, particularly in situations in which the agent’s understanding of the human is limited to a few observations. Thus, in the first step of this scenario, the human decided to find and grab the bread slice as the first action. Since the ad hoc agent decided to do the same action, it created a

Plan 15: Part of the plan generated by the ad hoc agent to avoid conflict with human.

- 1 find(ad_hoc_agent, cutlets)
 - 2 grab(ad_hoc_agent, cutlets)
 - 3 find(ad_hoc_agent, frying pan)
 - 4 putin(ad_hoc_agent, cutlets, frying pan)
 - 5 find(ad_hoc_agent, poundcake)
 - 6 grab(ad_hoc_agent, poundcake)
-

conflict. Plan 15 shows key steps of the plan computed by the ad hoc agent to overcome this conflict. The subsequent conversation between the questioner and the ad hoc agent is as follows:

- **Questioner:** “Why did you find cutlets in step 1?”
- **Ad hoc Agent:** “Because I have not found the cutlets and I wanted to grab them in time step 2”.
- **Questioner:** “Why did you not grab bread slice in step1?”
- **Ad hoc Agent:** “Because the human was holding the bread slice.”

This exchange indicates the ability of the ad hoc agent to change its plan in order to prevent a conflict with the human and to justify this decision. Figure 7.3 shows the belief tree created for tracing the ad hoc agent’s beliefs and justifying why it did not grab the bread slice in step 1. The yellow box represent the target belief in question. The green box represent the satisfied axioms, while the pink box represent their grounded literals. The hollow boxes represent the axioms that have the target belief as its head but were not satisfied in this example.

These results further suggests that KAT enables an ad hoc agent to generate relational descriptions as explanations of its decisions and beliefs and those of the other agents.

7.5 Chapter Summary

Transparent decision-making enables agents and human collaborators to understand the reasoning behind actions, anticipate behavior, and coordinate more effectively.

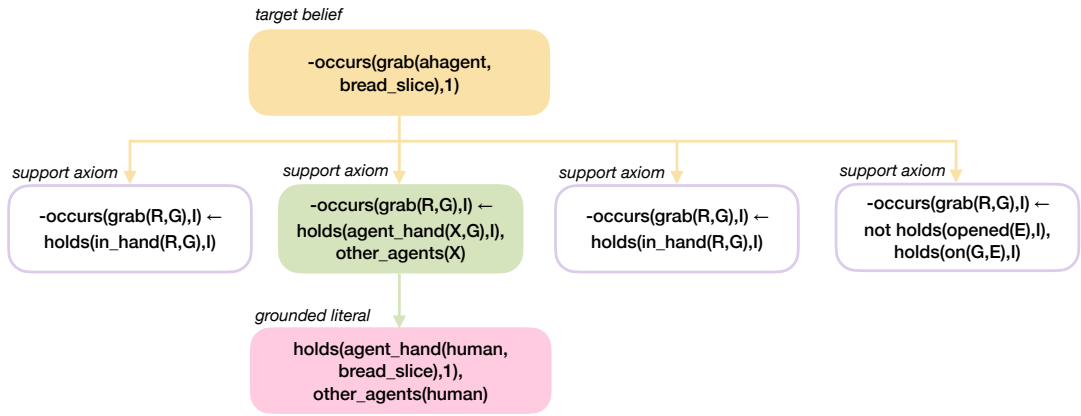


Figure 7.3: Belief tracing to explore why the ad hoc agent did not grab the bread slice in step 1.

This chapter explored the fifth and final component of KAT, *explanation generation*, a methodology for the ad hoc agent to automatically construct on demand relational descriptions of its own decisions and beliefs and those of other agents as explanations in response to different types of questions. The agent does so by iteratively and selectively identifying the axioms and literals that influence a particular action or belief, and using those whose antecedents satisfied in the relevant answer sets to construct the explanations.

Based on the results of the three domains Fort Attack, Half Field Offense and VirtualHome, we make the following key observations:

- **Observation 11:** *KAT supports the generation of relational descriptions as explanations of the ad hoc agent's decisions and beliefs and those of the other agents* - Supported by all domains (**Exp1, 3 and 5**).
- **Observation 12:** *Reasoning with incrementally learned axioms improves the ad hoc agent's ability to generate accurate and relevant explanations* - Supported by Fort Attack and Half Field Offense domains (**Exp2 and 4**).

These observations support the Research Question2 from Section 1.2.

Chapter 8

Conclusion

This thesis addresses the challenge of AHT, where an ad hoc agent collaborates with other agents without prior coordination. Methods considered state of the art for AHT formulate it primarily as a learning problem, using a large labeled dataset of different situations to model the action choices of other agents (or agent *types*) and determine the actions of the ad hoc agent. Such datasets are not readily available in practical domains, and these methods are often opaque making it difficult to understand the reasoning behind the ad hoc agent’s decisions.

To address these limitations, we propose KAT, an architecture that integrates the complementary strengths of knowledge-based and data-driven methods for reasoning and learning. Chapter 4 introduces the core architecture, where for any given goal, we enable an ad hoc agent to determine its actions through non-monotonic logical reasoning with: (a) prior domain-specific commonsense knowledge; and (b) models learned and revised rapidly to predict the behavior of other agents. Chapter 5 presents the third component: (c) task anticipation, where the agent anticipates abstract future goals based on generic knowledge of similar situations in a pretrained LLM. Chapter 6 focuses on the fourth component of KAT: (d) knowledge acquisition, where the ad hoc agent processes natural language descriptions and observations of other agents’ behavior, using a combination of a pretrained LLM and decision-tree induction to incrementally acquire and revise knowledge in the form of objects, actions, and axioms that govern domain dynamics. Finally, Chapter 7 presents the fifth component: (e) explanation generation, where the agent generates relational descriptions as on-demand explanations of its decisions and beliefs and those of other agents, in response to various types of questions. In the remainder of this chapter, we bring the thesis to a close by highlighting the key results, their significance, and the limitations of the proposed

architecture.

Section 8.1 summarizes the key findings from Chapters 4, 5, 6, and 7. Section 8.2 discusses the limitations of our architecture. Finally, Section 8.3 outlines promising directions for further work that may address the issues identified and extend the scope of our methods.

8.1 Key Results

The results we obtained by evaluating KAT across the three domains Fort Attack, Half Field Offense and VirtualHome highlight several notable outcomes. While each chapter concludes with a summary of its specific contributions, in this section we present the key outcomes that stand out across the thesis as a whole. These results indicate both the methodological contribution of this thesis in designing adaptive, transparent and resource efficient architectures and the effectiveness of KAT in addressing the challenges of AHT.

- (Thesis-wide) A key contribution of this thesis is the introduction and validation of a methodology for designing an AHT architecture that embeds the fundamental principles of refinement, ecological rationality, interactive learning and explainable agency, which are largely underexplored in modern AI literature. Our architecture, KAT, instantiates this methodology to enable an ad hoc agent to: automatically identify and reason with relevant information; leverage the generic knowledge encoded in a pretrained LLM for high-level task anticipation; rapidly learn models predicting the action choices of teammates; perform non-monotonic logical reasoning with prior knowledge and behavior models to plan and execute actions to jointly achieve the current and anticipated tasks; leverage a LLM to translate natural language descriptions of action outcomes into formal Answer Set Prolog representations of previously unknown objects, actions, and axioms; use decision tree induction to incrementally learn axioms based on observations; and generate on-demand explanations of the agent’s decisions and beliefs and those of others as answers to different queries. All empirical contributions presented in this thesis are enabled by these methodological choices, which provide a foundation for reliable, efficient, transparent, and scalable performance in AHT settings.
- (Section 4.8 from Chapter 4) One of the most notable outcomes of our work is

that KAT enables ad hoc agents to adapt to different teammates while performing comparably or better than the state of the art baselines in different scenarios and requiring orders of magnitude fewer resources (e.g., training examples and computation). KAT was able to do so by incrementally and rapidly learning models of other agents' behavior from limited training examples. These results challenge the common assumption that effective adaptation requires focusing on computationally intensive optimization over large datasets and many potential teammate models. Instead, KAT indicates that incremental learning from limited examples, combined with commonsense domain knowledge, can enable agents to adapt effectively to different teammates while using substantially less data and computation.

- (Section 5.2.2 from Chapter 5) A common criticism of reasoning-based methods is that they often do not scale well to larger or more complex domains. KAT indicates that by combining refinement mechanisms and rapidly-learned predictive models, KAT supports scalability. The refinement process enables the agent to focus reasoning on the relevant parts of the domain as needed, rather than reasoning over the entire domain description at once. This design choice enables the architecture to scale effectively across domains of varying complexity, including scenarios with many agents and interactions.
- (Section 5.1.2 and 5.1.3 from Chapter 5) KAT indicates that the use of generic knowledge from a pretrained Large Language Model to anticipate high-level future tasks and use them as joint goals improves team performance. This is due to KAT's incrementally updated prompts and the use of simple validators, which enhance the LLM's performance by incorporating contextual knowledge from domain experience. As a result, the agent can coordinate effectively, supporting better team performance.
- (Section 6.5 and 6.6 from Chapter 6) Reasoning methods are often criticized for their dependence on comprehensive domain knowledge. An important question, therefore, is how an agent can reason effectively with the knowledge that is available, while incrementally learning and revising additional knowledge over time. KAT indicates that it enables ad hoc agents to successfully learn previously unknown domain knowledge in terms of objects, actions, and axioms, reliably and efficiently. Reasoning with this incrementally learned knowledge results in minimal plans and improved performance for both the agent and the team.

- (Section 7.4 from Chapter 7) A potential concern with state of the art AHT methods is that they often lack transparency, making it difficult to understand the reasoning behind an agent's decisions and beliefs. KAT's use of knowledge-based reasoning and simple predictive models enables ad hoc agents to provide relational descriptions as on-demand explanations of their decisions and beliefs and those of the other agents, in response to different types of questions. Furthermore, reasoning with incrementally learned knowledge improves the accuracy and relevance of these explanations. This is important as it improves trust and interpretability, and allows agents to understand and anticipate the behavior of each other.

Implementation details of KAT, are provided in Appendix A.1.

8.2 Discussion

As discussed in Section 8.1, incorporating knowledge-based reasoning and heuristic-based learning enabled KAT to adapt to and collaborate efficiently with other agents. Furthermore, KAT has indicated several advantages over existing methods, indicating its suitability for addressing the AHT problem. The scope of this study, however, is shaped by the assumptions outlined in Sections 1.2, and certain limitations arising from these assumptions remain and should be considered.

First, the evaluation of KAT in this study has been limited to simulated domains. Although we conducted extensive experiments in three benchmark domains of varying complexities (Fort Attack, Half Field Offense and VirtualHome), these results alone do not guarantee reliable performance in real-world settings. The adaptation and resource-efficiency results reported in Sections 4.8, 5.1.3, and Section 5.2.2 depend on how closely these simulated domains reflect real-world settings. As simulated agents, sensors and environment dynamics are simpler than the real world, and does not guarantee that the same efficiency gains would carry over to real-world deployment. However, components of the architecture, such as non-monotonic logical reasoning and decision heuristics, have been employed successfully in robotics and other open-world domains. But additional studies are required to determine how KAT performs under real-world conditions such as perceptual noise, human variability, and environment dynamics that cannot be fully captured in simulation.

Secondly, the evaluation of KAT under partial observability is limited and requires further investigation. While we investigated KAT's performance under partial sensing,

where the agent can observe only part of the environment, this analysis remains limited and requires further study. The adaptation results reported in Section 5.3.2 rely on the agent's ability to build an accurate model of other agents' behavior in order to adapt effectively. As this adaptation was primarily demonstrated under full or mildly restricted observability, the same result does not indicate that KAT would adapt just as well when the agent's view of the environment is more limited or when operating under noisy observations and delayed communication. Furthermore, while the domains used in this thesis include some stochasticity in agent actions and uncertain action outcomes, a more detailed investigation is required to understand how KAT behaves when actions are highly non-deterministic and the environment introduces substantial noise.

Thirdly, this study assumes that all teammates behave cooperatively. While some experiments involved interactions with adversarial agents in the Fort Attack and Half Field Offense domains, these agents were clearly identified as adversarial from the beginning, as they belonged to opposing teams. This assumption qualifies both the adaptation results in Sections 4.8, 5.1.3, 5.2.2, and the knowledge acquisition results in Section 7.4, where we do not consider agents whose status as a teammate or opponent may not be known a priori, or scenarios in which teammates provide misleading information, act against the team's objectives, or behave adversarially within the team. In real-world applications, AHT may involve agents who are uncooperative or trying to deceive the ad hoc agent. Such conditions can affect an ad hoc agent's ability to model teammate behavior, coordinate effectively, and maintain robust performance and need further investigation.

Lastly, we assume that the actions of other agents, or the relevant states, can be observed directly in order to update and learn the behavior prediction models. This assumption may not hold in real-world domains, where such information is often not directly observable and must instead be inferred from noisy, incomplete, or unreliable sensor data. Such errors can be integrated to the learned models and negatively affect the agent's ability to predict teammate behavior accurately. Thus, the reported accuracy of these models and adaptation results in Sections 4.8, 5.1.3, 5.2.2, and the explanation generation results in Section 6.6, may not generalize to settings where the agent must rely on uncertain observations. Evaluating KAT under conditions in which other agents' actions must be inferred from noisy and incomplete observations remains as future work.

8.3 Directions for Future Work

This section outlines several promising directions for further work that may help address the issues identified in Section 8.2 and further improve our KAT architecture.

Adapting to Adversarial Agents: In real-world AHT domains, an agent may encounter teammates that behave adversarially within the team. While KAT's heuristic-based predictive behavior models can capture changes in teammate behavior from a limited number of examples, in this study we have not explored their ability to recognize or respond to adversarial or strategically decisive agents. Thus, an important direction for future work is to investigate how an ad hoc agent can identify and adapt to such behaviors. This could involve developing mechanisms to model and track unexpected or adversarial behavior patterns, and dynamically adjusting the agent's decision-making process in response.

Domain Knowledge Requirement: KAT requires some minimum amount of initial knowledge to be provided. The absence of sufficient prior knowledge can pose challenges and make the agent unusable. The use of human-provided descriptions to acquire domain knowledge autonomously, as described in Chapter 6, offers a promising approach to overcome this limitation. However, this raises important research questions: how can the agent detect and handle incorrect or misleading information provided by other agents, and how can knowledge—once learned or provided—be effectively transferred across domains? Investigating these questions presents a promising direction for future work, as it could provide valuable insights into designing more flexible and adaptive ad hoc agents.

Evaluating KAT in Physical Robots: Evaluating KAT on physical robots presents an important direction for future research. Deploying KAT on a physical robot and evaluating with humans would allow us to identify limitations that do not arise in simulation domains, such as perceptual noise, actuation delays, and the human interpretation of explanations generated by the agent.

Tailoring Explanations to Users: KAT currently supports providing explanations of the agent's decisions and beliefs through relational descriptions. However, the effectiveness of these explanations may vary depending on the user's expertise and context. For example, a user attempting to debug the system may require explanations that highlight system errors or inconsistencies, whereas an end-user should receive explanations that convey relevant reasoning without exposing technical details or bugs. Investigating how different types of explanations can be constructed, and identifying

the characteristics that make them useful to different audiences, represents an important research direction. Future work could validate explanations through human studies or other established evaluation methods.

Incorporating Negotiation and Communication Protocols: Another interesting future research direction is how to incorporate negotiation and communication protocols into KAT (where available), particularly when coordinating with larger, heterogeneous teams. Such mechanisms could enable ad hoc agents to collaboratively resolve conflicts, share intent, and distribute tasks more efficiently in settings where agents differ in capabilities, knowledge, or objectives. Thus, exploring how negotiation strategies and communication protocols can be integrated into KAT, while preserving its robustness and scalability, represents a promising research direction.

Appendix A

Supporting Materials

This appendix presents supporting material for the KAT architecture proposed in this thesis.

A.1 Implementation Details

- Framework: implemented in Python.
- Knowledge representation: SPARC (Balai et al., 2013).
- ASP Reasoning engine: Clingo (Gebser et al., 2017).
- LLM: GPT-4o mini (OpenAI and team, 2024), accessed via OpenAI API; default parameters (temperature = 1.0); no domain-specific fine-tuning.
- Hardware: Standard desktop machine with AMD Ryzen 9 3900XT (12-core, 24 threads), 31.2GB RAM, 2.5TB disk, Ubuntu 20.04.6 LTS (64-bit).
- Code repository: (Dodampegama and Sridharan, 2026) (for VirtualHome) and (Dodampegama and Sridharan, 2024) (for Fort Attack and Half Field Offense).

The full implementation of KAT, together with supporting materials for each domain, has been made publicly available. For the VirtualHome domain, the repository (Dodampegama and Sridharan, 2026) includes: the Answer Set Prolog knowledge base for the ad hoc agent and human; learned behavior models used to predict other agents' actions; implementation and example outputs of the explanation generation component; implementation of the knowledge acquisition component; any modified simulation files integrating KAT with the VirtualHome simulator; the Python code

connecting these components; and execution traces demonstrating the architecture’s behavior. It also includes execution videos of human and ad hoc agents collaborating on the domain. Corresponding repositories are also available for Fort Attack and Half Field Offense domains, each including the domain-specific Answer Set Prolog programs, learned models, integration python code, and execution videos (Dodampegama and Sridharan, 2024).

A.2 Answer Set Prolog Knowledge Base

Here we provide an example of the initial knowledge base (Answer Set Prolog program, Section 4.2) of the ad hoc agent in the VirtualHome environment. The knowledge bases for other domains, Fort Attack and Half Field Offense follow a similar structure and are available in our GitHub repository (Dodampegama and Sridharan, 2024).

```

1 #const n = 10.
2
3 sorts
4 #food = {cereal, breadslice, bananas, apple, plum, chips, cutlets,
5         cupcake, candybar}.
6 #drinks = {coffee, juice, milk, wine}.
7 #electricals = {dishwasher, microwave}.
8 #appliances = {computer, coffeemaker} + #electricals.
9 #plates = {plate}.
10 #glasses = {waterglass, mug}.
11 #containers = {coffeepot} + #plates + #glasses.
12 #others = {cellphone, book, boardgame}.
13 #locations = {kitchentable, kitchen_smalltable, counter_one,
14             counter_three, bedroom_coffeetable, bedroom_desk,
15             livingroom_coffeetable, livingroom_desk}.
16 #surfaces = #locations + #appliances.
17 #graspable = #food + #drinks + #containers + #others.
18 #objects = #appliances + #graspable.
19 #agent = {ahagent1}.
20 #other_agents = {human, ahagent2}.
21 #all_agents = #agent + #other_agents.
22 #step = 0..n.
23 % actions
24 #agent_actions = move(#agent, #locations) +
25                 grab(#agent, #graspable) +

```

```

26         put(#agent,#graspable,#surfaces) +
27         open(#agent,#electricals) +
28         close(#agent,#electricals) +
29         switchon(#agent,#appliances) +
30         switchoff(#agent,#appliances).
31 #exogenous_actions = exo_move(#other_agents,#locations) +
32         exo_grab(#other_agents,#graspable) +
33         exo_put(#other_agents,#graspable,#surfaces) +
34         exo_open(#other_agents,#electricals) +
35         exo_close(#other_agents,#electricals) +
36         exo_switchon(#other_agents,#appliances) +
37         exo_switchoff(#other_agents,#appliances).
38 #actions = #agent_actions + #exogenous_actions.
39 % inertial fluents
40 #inertial_f = at(#agent,#locations) +
41         in_hand(#agent,#graspable) +
42         on(#objects,#surfaces) +
43         opened(#electricals) +
44         switchedon(#appliances).
45 % defined fluents
46 #defined_f = agent_at(#other_agents,#locations) +
47         agent_hand(#other_agents,#graspable).
48 #fluents = #inertial_f + #defined_f.
49
50 predicates
51 holds(#fluents, #step).
52 occurs(#actions, #step).
53 next_to(#locations, #locations).
54 success().
55 something_happened(#step).
56 goal(#step).
57 planning(#step).
58 current_step(#step).
59
60 rules
61 % ----- casual laws ----- %
62
63 % move causes the agent to be at that location.
64 holds(at(R,L),I+1) :- occurs(move(R,L),I), #agent(R), #locations(L).
65
66 % grab causes an object to be in the hand of the agent.

```

```

67 holds(in_hand(R,G),I+1) :- occurs(grab(R,G),I), #agent(R), #
    graspable(G).
68
69 % grab causes an object to be removed form the current surface.
70 -holds(on(G,S),I+1) :- occurs(grab(R,G),I), holds(on(G,S),I), #agent
    (R), #graspable(G), #surfaces(S).
71
72 % put causes an object to be placed in the relevant surface.
73 holds(on(G,S),I+1) :- occurs(put(R,G,S),I), holds(in_hand(R,G),I), G
    != S, #agent(R), #graspable(G), #surfaces(S).
74
75 % put causes an object to be removed from the agent hand.
76 -holds(in_hand(R,G),I+1) :- occurs(put(R,G,S),I), holds(in_hand(R,G)
    ,I), G != S, #agent(R), #graspable(G), #surfaces(S).
77
78 % open causes the agent to open a door of a electrical.
79 holds(opened(E),I+1) :- occurs(open(R,E),I), not holds(opened(E),I),
    #agent(R), #electricals(E).
80
81 % close causes the agent to close a door of a appliance.
82 -holds(opened(E),I+1) :- occurs(close(R,E),I), holds(opened(E),I), #
    agent(R), #electricals(E).
83
84 % switch on causes the agent to switch on an appliance.
85 holds(switchedon(A),I+1) :- occurs(switchon(R,A),I), not holds(
    switchedon(A),I), #agent(R), #appliances(A).
86
87 % switch off causes the agent to switch off an appliance.
88 -holds(switchedon(A),I+1) :- occurs(switchoff(R,A),I), holds(
    switchedon(A),I), #agent(R), #appliances(A).
89
90 % exogeneous actions effect
91 holds(agent_at(O,L),I+1) :- occurs(exo_move(O,L),I), #other_agents(O
    ), #locations(L).
92 holds(agent_hand(O,G),I+1) :- occurs(exo_grab(O,G),I), #other_agents
    (O), #graspable(G).
93 -holds(on(G,S),I+1) :- occurs(exo_grab(O,G),I), holds(on(G,S),I), #
    other_agents(O), #graspable(G), #surfaces(S).
94 holds(on(G,S),I+1) :- occurs(exo_put(O,G,S),I), #other_agents(O), #
    graspable(G), #surfaces(S).
95 -holds(agent_hand(O,G),I+1) :- occurs(exo_put(O,G,S),I), #
    other_agents(O), #graspable(G), #surfaces(S).

```

```

96 holds(opened(E),I+1) :- occurs(exo_open(O,E),I), #other_agents(O), #
    electricals(E).
97 -holds(opened(E),I+1) :- occurs(exo_close(O,E),I), #other_agents(O),
    #electricals(E).
98 holds(switchedon(A),I+1) :- occurs(exo_switchon(O,A),I), #
    other_agents(O), #appliances(A).
99 -holds(switchedon(A),I+1) :- occurs(exo_switchoff(O,A),I), #
    other_agents(O), #appliances(A).
100
101 % ----- state constraints ----- %
102
103 % agent cannot be in two places at the same time.
104 -holds(at(R,L1),I) :- holds(at(R,L2),I), L1 != L2, #agent(R), #
    locations(L1), #locations(L2).
105
106 % next_to works both ways.
107 next_to(L1,L2) :- next_to(L2,L1).
108
109 % if not specified then not next_to each other.
110 -next_to(L1,L2) :- not next_to(L1,L2).
111
112 % ----- executability conditions ----- %
113
114 %% move
115 % impossible to move to a location if the agent is already in that
    location.
116 -occurs(move(R,L),I) :- holds(at(R,L),I), #agent(R), #locations(L).
117
118 % impossible to move between locations that are not next to each
    other.
119 -occurs(move(R,L1),I) :- holds(at(R,L2),I), -next_to(L1,L2).
120
121 %% grab
122 % impossible to grab an object if the agent is not at the same
    location as the object.
123 -occurs(grab(R,G),I) :- not holds(at(R,L),I), holds(on(G,L),I), #
    agent(R), #graspable(G), #locations(L).
124
125 % impossible to grab something from an appliance if the agent is not
    in the same location as the electrical.
126 -occurs(grab(R,G),I) :- not holds(at(R,L),I), holds(on(A,L),I), holds
    (on(G,A),I), #agent(R), #graspable(G), #appliances(A), #locations

```

```

(L) .
127
128 % impossible to grab something from inside an electrical if the door
    is closed.
129 -occurs(grab(R,G),I):- not holds(opened(E),I), holds(on(G,E),I), #
    agent(R), #graspable(G), #electricals(E).
130
131 % impossible to grab something if that object is already in the hand
    of the agent.
132 -occurs(grab(R,G),I):- holds(in_hand(R,G),I), #agent(R), #graspable(
    G).
133
134 % impossible to grab something if that object is in the hand of the
    other agent.
135 -occurs(grab(R,G),I):- holds(agent_hand(T,G),I), #other_agents(T), #
    agent(R), #graspable(G).
136
137 % impossible to grab a third object if two objects are already in
    the hand of the agent
138 -occurs(grab(R,O3),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),
    I), O1 != O2, O2 != O3, O1 != O3, #agent(R), #graspable(O1), #
    graspable(O2), #graspable(O3).
139
140 %% put
141 % impossible to put an object down if the objects is not in the hand
    of the agent.
142 -occurs(put(R,G,S),I) :- not holds(in_hand(R,G),I), #agent(R), #
    graspable(G), #surfaces(S).
143
144 % impossible to put an object down if the agent is not at that
    location.
145 -occurs(put(R,G,L),I) :- not holds(at(R,L),I), #agent(R), #graspable
    (G), #locations(L).
146
147 % impossible to put an object inside an appliance if the agent is
    not at the location of the appliance.
148 -occurs(put(R,G,A),I) :- not holds(at(R,L),I), holds(on(A,L),I), #
    agent(R), #graspable(G), #locations(L), #appliances(A).
149
150 % impossible to put something inside an electrical if the door is
    closed.

```

```

151 -occurs(put(R,G,E),I) :- not holds(opened(E),I), #agent(R), #
    graspable(G), #electricals(E).
152
153 %% switchon
154 % impossible to switchon if two objects are already in the hand of
    the agent
155 -occurs(switchon(R,A),I):- holds(in_hand(R,O1),I), holds(in_hand(R,
    O2),I), O1 != O2, #agent(R), #graspable(O1), #graspable(O2), #
    appliances(A).
156
157 % impossible to switch on an appliance if the agent is not in the
    same location as the appliance.
158 -occurs(switchon(R,A),I):- not holds(at(R,L),I), holds(on(A,L),I), #
    agent(R), #appliances(A), #locations(L).
159
160 % impossible to switch on an electrical unless the door is closed.
161 -occurs(switchon(R,E),I) :- holds(opened(E),I), #agent(R), #
    electricals(E).
162
163 % impossible to switch on an appliance if it is already switched on.
164 -occurs(switchon(R,A),I):- holds(switchedon(A),I), #agent(R), #
    appliances(A).
165
166 %% switchoff
167 % impossible to switchoff if two objects are already in the hand of
    the agent
168 -occurs(switchoff(R,A),I):- holds(in_hand(R,O1),I), holds(in_hand(R,
    O2),I), O1 != O2, #agent(R), #graspable(O1), #graspable(O2), #
    appliances(A).
169
170 % impossible to switch off an appliance if the agent is not in the
    same location as the appliance.
171 -occurs(switchoff(R,A),I):- not holds(at(R,L),I), holds(on(A,L),I),
    #agent(R), #appliances(A), #locations(L).
172
173 % impossible to switch off an appliance if it is already switched
    off.
174 -occurs(switchoff(R,A),I):- not holds(switchedon(A),I), #agent(R), #
    appliances(A).
175
176 %% open

```

```

177 % impossible to open if two objects are already in the hand of the
    agent.
178 -occurs(open(R,E),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),I
    ), O1 != O2, #agent(R), #graspable(O1), #graspable(O2), #
    electricals(E).
179
180 % impossible to open the door of an electrical if the agent is not
    in the same location as the electrical.
181 -occurs(open(R,E),I):- not holds(at(R,L),I), holds(on(E,L),I), #
    agent(R), #electricals(E), #locations(L).
182
183 % impossible to open a door if it is already opened.
184 -occurs(open(R,E),I):- holds(opened(E),I), #agent(R), #electricals(E
    ).
185
186 % impossible to open the door of an electrical if it is not
    switched_off.
187 -occurs(open(R,E),I) :- holds(switchedon(E),I), #agent(R), #
    electricals(E).
188
189 %% close
190 % impossible to close if two objects are already in the hand of the
    agent
191 -occurs(close(R,E),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),
    I), O1 != O2, #agent(R), #graspable(O1), #graspable(O2), #
    electricals(E).
192
193 % impossible to close the door of an electrical if the agent is not
    in the same location as the electrical.
194 -occurs(close(R,E),I):- not holds(at(R,L),I), holds(on(E,L),I), #
    agent(R), #electricals(E), #locations(L).
195
196 % impossible to close a door if it is already closed.
197 -occurs(close(R,E),I):- not holds(opened(E),I), #agent(R), #
    electricals(E).
198
199 % ----- inertial axioms ----- %
200
201 holds(F,I+1) :- #inertial_f(F), holds(F,I), not -holds(F,I+1).
202 -holds(F,I+1) :- #inertial_f(F), -holds(F,I), not holds(F,I+1).
203
204 % ----- CWA ----- %

```

```

205
206 -occurs(A,I) :- not occurs(A,I).
207 -holds(F,0) :- #inertial_f(F), not holds(F,0).
208 -holds(F,I) :- #defined_f(F), not holds(F,I).
209
210 % ----- planning ----- %
211
212 % to achieve success the system should satisfy the goal. Failure is
    not acceptable.
213 success :- goal(I), I <= n.
214 :- not success, current_step(I0), planning(I0).
215
216 % consider the occurrence of exogenous actions when they are
    absolutely necessary for resolving a conflict.
217 occurs(A,I) :+ #agent_actions(A), #step(I), current_step(I0),
    planning(I0), I0 <= I.
218
219 % agent can not execute parallel actions.
220 -occurs(A1,I) :- occurs(A2,I), A1 != A2, #agent_actions(A1), #
    agent_actions(A2).
221
222 % an action should occur at each time step until the goal is
    achieved.
223 something_happened(I1) :- current_step(I0), planning(I0), I0 <= I1,
    occurs(A,I1), #agent_actions(A).
224 :- not something_happened(I), something_happened(I+1), I0 <= I,
    current_step(I0), planning(I0).
225
226 %%%-----%%
227
228 planning(I) :- current_step(I).
229
230 planning(0).
231 current_step(0).
232
233 %%%-----%%
234
235 % define the relative arrangement of places.
236 next_to(counter_one, kitchentable).
237 next_to(counter_one, counter_three).
238 next_to(counter_three, kitchentable).
239 next_to(kitchentable, kitchen_smalltable).

```

```

240 next_to(kitchen_smalltable, bedroom_desk).
241 next_to(bedroom_desk, bedroom_coffeetable).
242 next_to(kitchen_smalltable, livingroom_desk).
243 next_to(livingroom_desk, livingroom_coffeetable).
244
245 % ----- %
246
247 display
248 occurs.

```

Listing A.1: ASP knowledge base of the ad hoc agent following KAT architecture in the VirtualHome domain.

A.3 Task Anticipation Prompts and Outputs

Here we provide the complete prompts given to the LLM and the corresponding outputs for the qualitative examples discussed in Chapter 5, Section 5.1.3.

Example 10. *[Task Anticipation with Two Agents; Human and KAT-based Ad Hoc Agent]*

Consider the scenario in which the ad hoc agent following our KAT architecture is collaborating with a human in the *VirtualHome* environment. The ground-truth task routine for the given day was: *Prepare Breakfast for a Weekday, Prepare Workstation, Make Coffee, Serve Snacks*.

The prompt to the LLM was automatically constructed by the ad hoc agent using a predefined system message, two examples randomly chosen from the agent’s prior experience for few-shot prompting and described using Chain-of-Thought (CoT) method, and information about current and completed tasks (Section 5.1). The complete prompt constructed in this example by the ad hoc agent is shown below:

You are an intelligent household assistant. Anticipate future tasks based on previous days' data. You will get a list of tasks that can be executed in the household environment, day of the week, an example of a partially completed routine and the expected output. Complete the rest of the tasks for the partially completed routine only using the tasks from the task list, following the examples given.

Task List: [Prepare breakfast weekday, Prepare coffee, Prepare home workstation, Prepare lunch, Pack bag, Clean kitchen, Prepare breakfast weekend, Prepare table for guests, Serve snacks, Prepare activities, Clean dishes]

Example1: Complete the following routine on a weekend the human is having guests for lunch : []

[Prepare breakfast weekend, Prepare table for guests, Prepare lunch, Clean dishes] As everyday, on weekends it is important to prepare breakfast first. Since the guests are arriving the agent must next make table to serve lunch to the guests. It is important to start on lunch as soon as the table is ready. Finally the agent may not have time to clean the whole kitchen while having guests but it can clean atleast the dirty dishes from lunch to maintain the hygiene.

Example2: Complete the following routine on a weekday the human is leaving for office: []

[Prepare breakfast weekday, Pack bag, Prepare lunch, Clean kitchen] Since the human is leaving for the office, it's a busy morning focused on essentials. Breakfast is prepared first to start the day with energy. Next, packing the bag ensures they have everything needed for work. Lunch is then prepared in case they need it later. Finally, after they've left, the kitchen is cleaned to maintain a tidy space.

Complete the following routine on a weekday the human is working from home and need a snack: []

The output generated by the LLM for above query consisted of a list of anticipated future tasks as follows:

Prepare breakfast weekday, Prepare home work-station, Serve snacks, Clean kitchen

Since the human is working from home, the first task is to prepare breakfast to fuel their day. Next, setting up the home work-station is essential for a productive work environment. As they need a snack, serving snacks follows to keep them energized. Finally, cleaning the kitchen ensures the space remains tidy and conducive to work.

This output is then passed to the validator where it was compared against the agent's prior experience for verification. The refined task list obtained after validation is:

[Prepare breakfast weekday, Prepare home work-station, Serve snacks]

While the first three tasks were often performed together, based on prior experience the final anticipated task — Clean kitchen, was rarely selected. Thus, the validator removed it from the list.

The first task from the resulting list was then used by the ad hoc agent to plan together with any currently assigned tasks. If the first task of the list is same as the current task, the agent selected the next task from the list. This allowed the agent to jointly plan for multiple tasks and execute actions efficiently.

Example 11. *[Task Anticipation with Three Agents; Human and Two KAT-based Ad Hoc Agents]*

In this scenario the domain consisted of three agents, one human and two ad hoc agents, both following the KAT architecture. The ground-truth task routine for this setting was: *Prepare Breakfast for a Weekend, Prepare Table for Guests, Prepare Lunch, Clean Dishes.*

Each ad hoc agent independently constructed a prompt and queried the LLM to anticipate future tasks. The prompt used by one of the agents is shown below (the prompt used by the other agent was similar except for the examples included):

You are an intelligent household assistant. Anticipate future tasks based on previous days' data. You will get a list of tasks that can be executed in the household environment, day of the week, an example of a partially completed routine and the expected output. Complete the rest of the tasks for the partially completed routine only using the tasks from the task list, following the examples given.

Task List: [Prepare breakfast weekday, Prepare coffee, Prepare home workstation, Prepare lunch, Pack bag, Clean kitchen, Prepare breakfast weekend, Prepare table for guests, Serve snacks, Prepare activities, Clean dishes]

Example1: Complete the following routine on a weekend the human is not having guests: []

[Prepare breakfast weekend, Clean kitchen, Prepare activities, Prepare lunch]

Since it is a weekend, the routine starts with preparing breakfast. After breakfast, the kitchen needs to be cleaned to remove any mess from the meal. The next task involves preparing activities to enjoy the day. Finally, lunch should be prepared to ensure the human is nourished throughout the day.

Example2: Complete the following routine on a weekday the human is leaving for office: []

[Prepare breakfast weekday, Pack bag, Prepare lunch, Clean kitchen] Since the human is leaving for the office, it's a busy morning focused on essentials. Breakfast is prepared first to start the day with energy. Next, packing the bag ensures they have everything needed for work. Lunch is then prepared in case they need it later. Finally, after they've left, the kitchen is cleaned to maintain a tidy space.

Complete the following routine on a weekend the human is having guests for lunch: []

However, since each agent prompted the LLM separately, they received two different outputs from the LLM. The output received by the first ad hoc agent was:

Prepare breakfast weekend, Prepare table for guests, Prepare lunch, Serve snacks

This output after refined by the validator resulted in:

[Prepare breakfast weekend, Prepare table for guests, Prepare lunch]

Since the human usually did not require snacks after lunch, the validator consulted prior experience and removed the ‘Serve snacks’ task from the generated list.

The output received by the second ad hoc agent was:

Prepare breakfast weekend, Prepare table for guests, Prepare lunch, Serve snacks, Clean dishes

Again, the validator refined the LLM’s output by removing the ‘Serve snacks’ task. Additionally, the LLM correctly anticipated the ‘Clean dishes’ task for the second agent, as it is consistent with prior observations in similar contexts. Thus it was retained. The resulting task list for the second agent was:

[Prepare breakfast weekend, Prepare table for guests, Prepare lunch, Clean dishes]

These refined task lists were then used by the two ad hoc agents independently in their planning to select actions and jointly achieve goals.

Example 12. *[Task Anticipation with Two Agents; Human and LLM Based Ad Hoc Agent]*

In this scenario the domain consisted of two agents, one human and an ad hoc agent that used the LLM to directly compute sequences of actions to be executed in the *VirtualHome* domain to achieve specific tasks (Base 4 – Section 5.1.1). The ground-truth task routine was *Prepare Breakfast for a Weekday, Prepare Workstation, Make Coffee, Prepare Lunch*.

The prompt to the LLM by the ad hoc agent consisted of a system message, details of actions available in *VirtualHome* and their intended purpose, some *Action Feasibility Rules*, information about adjacent places in the domain, detailed example of selecting an action, current world state (location of agents, objects, and appliances, each appliance’s state, objects held by the agents), description on task to be performed, immediate previous actions of the human and the ad hoc agent, any specific information to be considered on any given day (human working from home, weekday or the weekend, human is expecting guests or not), and a query asking the LLM to generate the next action in the sequence for achieving the assigned goal (see Section 5.1.1 for more details). Below is the prompt created by the agent in this case:

You are an intelligent household assistant collaborating with a human to perform daily tasks. Your task is to achieve the goal by selecting the next feasible action in a logical sequence. Briefly explain your reasoning and predict only one action at a time, returning the action as a Python list, e.g., [move(agent,location)]. If the goal is already achieved return 'goal_achieved'. You may choose not to take any action if necessary.

Possible actions:

move(agent,location): Move the agent to an adjacent location.

grab(agent,object): Pick up an object.

put(agent,object,surface): Place an object on a surface.

open(agent,appliance): Open an appliance.

close(agent,appliance): Close an appliance.

switchon(agent,appliance): Switch on an appliance.

switchoff(agent,appliance): Switch off an appliance.

Action Feasibility Rules:

Movement Limitation(Critical): You must only move to adjacent locations. Adjacent locations are explicitly defined in the next_to relationships below. If the destination is not adjacent, the move action is prohibited. Always check adjacency before predicting a move.

Object Location: You must be in the same location as an object to perform any action on it (e.g., grab or put).

Carrying Limit: You cannot hold more than two objects at a time. When holding two objects, actions like open, close, switchon, or switchoff require putting down at least one object first.

Appliance Safety: For safety, you should not open appliance doors when they are switched on.

If the human is holding an object, assume they will handle the tasks involving that object. You must not attempt to grab, or interact with the object in the humans possession. Instead, focus on completing other parts of the goal unrelated to that object.

You can only move between the following adjacent places:

```

next_to(counter_one, kitchentable)

next_to(counter_one, counter_three)

next_to(counter_three, kitchentable)

next_to(kitchentable, kitchen_smalltable)

next_to(kitchen_smalltable, bedroom_desk)

next_to(bedroom_desk, bedroom_coffeetable)

next_to(kitchen_smalltable, livingroom_desk)

next_to(livingroom_desk, livingroom_coffeetable)

```

Example Sequence for Rational Action:

If a task requires moving an item from one room to another, you should:

1. Move to the item's location.
2. Grab the item.
3. Move to the destination.
4. Place the item.

Current world state:

human location: kitchentable, agent location: kitchentable, milk on kitchentable, apple on livingroom_coffeetable, plum on livingroom_coffeetable, mug on livingroom_coffeetable, cupcake on bedroom_coffeetable, candy-bar on bedroom_coffeetable, wine on bedroom_coffeetable, boardgame on bedroom_coffeetable, waterglass on livingroom_desk, computer on livingroom_desk, juice on bedroom_desk, plate on counter_one, cutlets on counter_one, coffeemaker on counter_three, coffeepot placed inside coffeemaker, book on kitchen_smalltable, chips on kitchen_smalltable, cereal on livingroom_desk, bananas on livingroom_coffeetable, breadslice on counter_three, dishwasher under counter_three, cellphone on livingroom_coffeetable, dishwasher door closed, dishwasher not switchedon, computer not switchedon, coffeemaker not switchedon, coffee not prepared.

The teams current goal is to Prepare breakfast on a weekday:

goal = on (cereal, kitchentable), on (breadslice, kitchentable), on (bananas, kitchentable), switchedon (computer), on (book, livingroom_desk), on (cell-phone, livingroom_desk)

Immediate previous actions of the human were: move(human,kitchentable)

Your successful previous actions were: move(agent,kitchentable)

Anticipate an action for you to perform given above state and constraints on a weekday the human is working from home. Ensure the Action Feasibility Rules: Movement Limitation(Critical), Object Location, Carrying Limit and Appliance Safety are strictly respected and predict the next logical action.

The output received by the ad hoc agent from the LLM for above query was:

The current goal is to prepare breakfast by placing cereal, breadslice, and bananas on the kitchentable.

Currently, the human is at the kitchentable, and the cereal is located on the livingroom_desk, while the breadslice is on counter_three, and the bananas are on the livingroom_coffeetable. The human can handle the items they are near, while the agent can assist by moving to grab items that the human cannot access directly.

To start the process, the agent should move to grab one of the objects. The closest item to the agent at kitchentable is the breadslice on counter_three.

Hence, the next logical action would be to:

1. Move to the counter_three to grab the breadslice.

So, I predict the next action as follows: ['move(agent,counter_three)']

Since the action selected by the LLM is feasible in this scenario, the ad hoc agent and human team performed the following actions:

Human ⟨move⟩ coffeetable | Agent ⟨move⟩ counter_three

In the next three consecutive steps the LLM received similar prompts and selected to perform the following actions: grab(agent, breadslice), move(agent, kitchentable),

`put(agent, breadslice, kitchentable).`

The actions performance by the human and the ad hoc agent in these steps were:

Human $\langle \text{grab} \rangle$ bananas | Agent $\langle \text{grab} \rangle$ breadslice

Human $\langle \text{move} \rangle$ livingroom_desk | Agent $\langle \text{move} \rangle$ kitchentable

Human $\langle \text{grab} \rangle$ cereal | Agent $\langle \text{putback} \rangle$ breadslice kitchentable

However the next action anticipated by the LLM for the ad hoc agent was:

In the current state, the human is holding both cereal and bananas, which means they cannot handle any additional objects. Since the goal involves switching on the computer and placing the book and cellphone on the livingroom desk, I need to focus on actions that can help achieve this goal. The computer is located at the livingroom desk, and I cannot carry any items for the human while the human is holding items.

Since there's no action feasible for me to grab or put anything while the human is busy, one action I can perform immediately is to move to the livingroom desk to prepare for placing the items once the human is free.

I will predict the next action based on this reasoning.

Next action: [`'move(agent, livingroom_desk)'`]

This action `move(agent, livingroom_desk)` violated the 'Movement Limitation (Critical)' rule in **Action Feasibility Rules**, which required the agent to verify the adjacency of locations before attempting to move.

Since the action selected by the LLM was not feasible, i.e., the agent cannot move between places that are not adjacent, the agent followed the recovery mechanism and sent feedback to the LLM (see Section 5.1.1). We allowed up to three such corrections per trial. Following was the feedback that was added to the previous prompt and resent to the LLM for action selection.

Your previous action `move(agent, livingroom_desk)` was not feasible since you can only move to adjacent locations. `livingroom_desk` and `kitchentable` are not next to each other. Please select a feasible action.

Ensure the Action Feasibility Rules: Movement Limitation(Critical) is strictly respected.

This lead the LLM to provide the following output:

Given the current state, the agent is at the kitchentable, and there are several items (including the book and cellphone) on the livingroom_coffeetable that need to be placed on the livingroom_desk. However, the agent cannot interact with the items in the human's possession (cereal) and cannot grab any items since they are not within the agent's immediate location.

The next logical action is for the agent to move to a location that allows for future actions, such as grabbing the book, which is on the kitchen_smalltable. The kitchen_smalltable is adjacent to the kitchentable. Therefore, the agent should move to the kitchen_smalltable.

The action I will predict is: ['move(agent,kitchen_smalltable)']

Since this actions was feasible the human and the ad hoc agent continued their execution as follows:

Human <move> kitchen_smalltable | Agent <move> kitchen_smalltable

A.4 Domain Knowledge of the Incomplete Ad Hoc Agent

Here we present the domain knowledge of the incomplete ad hoc agent used in the experiments described in Chapter 6, Section 6.4. Since the overall **structure of the ASP program** of the incomplete ad hoc agent is identical to that of a knowledgeable ad hoc agent in the *VirtualHome* domain, presented in Appendix A.2, here we include only the domain knowledge (objects, actions, causal laws, and executability conditions) that differs between the two agents. That is, the components that remain unchanged (e.g., predicates, state constraints, inertia axioms, the closed-world assumption (CWA), and planning axioms) are not repeated here. Knowledge retained from the complete domain specification is shown in blue, while knowledge removed from the incomplete agent is shown in red. The complete ASP programs used in the experiments are available in our open source code repository (Dodampegama and Sridharan, 2026).

```

1
2 % ----- objects ----- %
3
4 #objects = {cereal, breadslice, bananas, apple, plum, chips, cutlets,
             cupcake, candybar, coffee, juice, milk, wine, dishwasher, computer,
```

```

    coffeemaker, plate, waterglass, mug, coffeepot, cellphone, book,
    boardgame}.

5
6 % ----- actions ----- %
7
8 #agent_actions = move(#agent,#locations) +
9                 grab(#agent,#graspable) +
10                put(#agent,#graspable,#surfaces) +
11                open(#agent,#electricals) +
12                close(#agent,#electricals) +
13                switchon(#agent,#appliances) +
14                switchoff(#agent,#appliances) .
15
16 % ----- casual laws ----- %
17
18 % move causes the agent to be at that location.
19 holds(at(R,L),I+1) :- occurs(move(R,L),I), #agent(R), #locations(L).
20
21 % grab causes an object to be in the hand of the agent.
22 holds(in_hand(R,G),I+1) :- occurs(grab(R,G),I), #agent(R), #graspable(G).
23
24 % grab causes an object to be removed form the current surface.
25 -holds(on(G,S),I+1) :- occurs(grab(R,G),I), holds(on(G,S),I), #agent(R),
    #graspable(G), #surfaces(S).
26
27 % put causes an object to be placed in the relevant surface.
28 holds(on(G,S),I+1) :- occurs(put(R,G,S),I), holds(in_hand(R,G),I), G != S,
    #agent(R), #graspable(G), #surfaces(S).
29
30 % put causes an object to be removed from the agent hand.
31 -holds(in_hand(R,G),I+1) :- occurs(put(R,G,S),I), holds(in_hand(R,G),I), G != S,
    #agent(R), #graspable(G), #surfaces(S).
32
33 % open causes the agent to open a door of a electrical.
34 holds(opened(E),I+1) :- occurs(open(R,E),I), not holds(opened(E),I), #agent(R),
    #electricals(E).
35
36 % close causes the agent to close a door of a appliance.
37 -holds(opened(E),I+1) :- occurs(close(R,E),I), holds(opened(E),I), #agent(R),
    #electricals(E).
38
39 % switch on causes the agent to switch on an appliance.

```

```

40 holds(switchedon(A),I+1) :- occurs(switchon(R,A),I), not holds(switchedon(A),I),
    #agent(R), #appliances(A).
41
42 % switch off causes the agent to switch off an appliance.
43 -holds(switchedon(A),I+1) :- occurs(switchoff(R,A),I), holds(switchedon(A),I),
    #agent(R), #appliances(A).
44
45 % ----- executability conditions ----- %
46
47 % impossible to move to a location if the agent is already in that
    location.
48 -occurs(move(R,L),I) :- holds(at(R,L),I), #agent(R), #locations(L).
49
50 % impossible to move beteen locations that are not next to each
    other.
51 -occurs(move(R,L1),I) :- holds(at(R,L2),I), -next_to(L1,L2).
52
53 % impossible to grab an object if the agent is not at the same
    location as the object.
54 -occurs(grab(R,G),I) :- not holds(at(R,L),I), holds(on(G,L),I), #agent(R),
    #graspable(G), #locations(L).
55
56 % impossible to grab something from an appliance if the agent is not
    in the same location as the electrical.
57 -occurs(grab(R,G),I) :- not holds(at(R,L),I), holds(on(A,L),I), holds(on(G,A),I),
    #agent(R), #graspable(G), #appliances(A), #locations(L).
58
59 % impossible to grab something from inside an electrical if the door
    is closed.
60 -occurs(grab(R,G),I) :- not holds(opened(E),I), holds(on(G,E),I), #agent(R),
    #graspable(G), #electricals(E).
61
62 % impossible to grab something if that object is already in the hand
    of the agent.
63 -occurs(grab(R,G),I) :- holds(in_hand(R,G),I), #agent(R), #graspable(G).
64
65 % impossible to grab something if that object is in the hand of the
    other agent.
66 -occurs(grab(R,G),I) :- holds(agent_hand(T,G),I), #other_agents(T), #agent(R),
    #graspable(G).
67

```

```

68 % impossible to grab a third object if two objects are already in
    the hand of the agent
69 -occurs(grab(R,O3),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),I), O1 != O2,
    O2 != O3, O1 != O3, #agent(R), #graspable(O1), #graspable(O2),
    #graspable(O3).

70
71 % impossible to put an object down if the objects is not in the hand
    of the agent.
72 -occurs(put(R,G,S),I) :- not holds(in_hand(R,G),I), #agent(R), #graspable(G),
    #surfaces(S).

73
74 % impossible to put an object down if the agent is not at that
    location.
75 -occurs(put(R,G,L),I) :- not holds(at(R,L),I), #agent(R), #graspable(G),
    #locations(L).

76
77 % impossible to put an object inside an appliance if the agent is
    not at the location of the appliance.
78 -occurs(put(R,G,A),I) :- not holds(at(R,L),I), holds(on(A,L),I), #agent(R),
    #graspable(G), #locations(L), #appliances(A).

79
80 % impossible to put something inside an electrical if the door is
    closed.
81 -occurs(put(R,G,E),I) :- not holds(opened(E),I), #agent(R), #graspable(G),
    #electricals(E).

82
83 % impossible to switchon if two objects are already in the hand of
    the agent
84 -occurs(switchon(R,A),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),I), O1 !=
    O2, #agent(R), #graspable(O1), #graspable(O2), #appliances(A).

85
86 % impossible to switch on an appliance if the agent is not in the
    same location as the appliance.
87 -occurs(switchon(R,A),I):- not holds(at(R,L),I), holds(on(A,L),I), #agent(R),
    #appliances(A), #locations(L).

88
89 % impossible to switch on an electrical unless the door is closed.
90 -occurs(switchon(R,E),I) :- holds(opened(E),I), #agent(R), #electricals(E).

91
92 % impossible to switch on an appliance if it is already switched on.
93 -occurs(switchon(R,A),I):- holds(switchedon(A),I), #agent(R), #appliances(A).

94

```

```

95 % impossible to switchoff if two objects are already in the hand of
    the agent
96 -occurs(switchoff(R,A),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),I), O1 !=
    O2, #agent(R), #graspable(O1), #graspable(O2), #appliances(A).
97
98 % impossible to switch off an appliance if the agent is not in the
    same location as the appliance.
99 -occurs(switchoff(R,A),I):- not holds(at(R,L),I), holds(on(A,L),I), #agent(R),
    #appliances(A), #locations(L).
100
101 % impossible to switch off an appliance if it is already switched
    off.
102 -occurs(switchoff(R,A),I):- not holds(switchedon(A),I), #agent(R),
    #appliances(A).
103
104 % impossible to open the door of an electrical if the agent is not
    in the same location as the electrical.
105 -occurs(open(R,E),I):- not holds(at(R,L),I), holds(on(E,L),I), #agent(R),
    #electricals(E), #locations(L).
106
107 % impossible to open a door if it is already opened.
108 -occurs(open(R,E),I):- holds(opened(E),I), #agent(R), #electricals(E).
109
110 % impossible to open the door of an electrical if it is not
    switched_off.
111 -occurs(open(R,E),I):- holds(switchedon(E),I), #agent(R), #electricals(E).
112
113 % impossible to close if two objects are already in the hand of the
    agent
114 -occurs(close(R,E),I):- holds(in_hand(R,O1),I), holds(in_hand(R,O2),I), O1 != O2,
    #agent(R), #graspable(O1), #graspable(O2), #electricals(E).
115
116 % impossible to close the door of an electrical if the agent is not
    in the same location as the electrical.
117 -occurs(close(R,E),I):- not holds(at(R,L),I), holds(on(E,L),I), #agent(R),
    #electricals(E), #locations(L).
118
119 % impossible to close a door if it is already closed.

```

```
120 -occurs(close(R,E),I):- not holds(opened(E),I), #agent(R), #electricals(E).
```

Listing A.2: ASP knowledge base of the incomplete ad hoc agent against the knowledge base of a knowledgeable ad hoc agent following KAT architecture in the VirtualHome domain.

Bibliography

- Adadi, A. and Berrada, M. (2018). Peeking inside the black-box: A survey on explainable artificial intelligence (xai). *IEEE Access*, 6:52138–52160.
- Albrecht, S. V., Crandall, J. W., and Ramamoorthy, S. (2015). E-HBA: Using Action Policies for Expert Advice and Agent Typification. In *Workshop on Multiagent Interaction without Prior Coordination at AAAI*.
- Albrecht, S. V. and Ramamoorthy, S. (2013). A game-theoretic model and best-response learning method for ad hoc coordination in multiagent systems. In *International Conference on Autonomous Agents and Multiagent Systems*, Saint Paul, Minnesota, USA.
- Albrecht, S. V. and Stone, P. (2017). Reasoning about Hypothetical Agent Behaviors and their Parameters. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.
- Albrecht, S. V. and Stone, P. (2018). Autonomous agents modelling other agents: A comprehensive survey and open problems. *Artificial Intelligence*, 258:66–95.
- Alviano, M., Trieu, L. L., Cao Son, T., and Balduccini, M. (2023). Explanations for answer set programming. *Electronic Proceedings in Theoretical Computer Science*, 385:27–40.
- Anjomshoe, S., Najjar, A., Calvaresi, D., and Framling, K. (2019). Explainable agents and robots: Results from a systematic literature review. In *International Conference on Autonomous Agents and Multiagent Systems*, Montreal, Canada.
- Balai, E., Gelfond, M., and Zhang, Y. (2013). Towards Answer Set Programming with Sorts. In *Conference on Logic Programming and Nonmonotonic Reasoning*.

- Balduccini, M. and Gelfond, M. (2003). Logic Programs with Consistency-Restoring Rules. In *AAAI Spring Symposium on Logical Formalization of Commonsense Reasoning*.
- Balduccini, M., Regli, W. C., and Nguyen, D. N. (2014). An asp-based architecture for autonomous uavs in dynamic environments: Progress report.
- Baltag, A. and Moss, L. S. (2005). *Logics for Epistemic Programs*, pages 1–60. Springer Netherlands, Dordrecht.
- Baral, C., Gelfond, G., Pontelli, E., and Son, T. C. (2022). An action language for multi-agent domains. *Artificial Intelligence*, 302:103601.
- Baral, C., Gelfond, G., Son, T. C., and Pontelli, E. (2010a). Using answer set programming to model multi-agent scenarios involving agents’ knowledge about other’s knowledge. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, volume 1, page 259–266.
- Baral, C. and Gelfond, M. (1994). Logic programming and knowledge representation. *The Journal of Logic Programming*, 19-20:73–148. Special Issue: Ten Years of Logic Programming.
- Baral, C., Son, T. C., and Pontelli, E. (2010b). Reasoning about multi-agent domains using action language *C*: A preliminary study. In Dix, J., Fisher, M., and Novák, P., editors, *Computational Logic in Multi-Agent Systems*, pages 46–63. Springer Berlin Heidelberg.
- Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., and Herrera, F. (2020). Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58:82–115.
- Barrett, S., Agmon, N., Hazon, N., Kraus, S., and Stone, P. (2014). Communicating with unknown teammates. In *European Conference on Artificial Intelligence*.
- Barrett, S., Rosenfeld, A., Kraus, S., and Stone, P. (2017). Making friends on the fly: Cooperating with new teammates. *Artificial Intelligence*, 242:132–171.
- Barrett, S. and Stone, P. (2015). Cooperating with unknown teammates in complex domains: A robot soccer case study of ad hoc teamwork. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1).

- Barrett, S., Stone, P., and Kraus, S. (2011). Empirical evaluation of ad hoc teamwork in the pursuit domain. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 567–574.
- Barrett, S., Stone, P., Kraus, S., and Rosenfeld, A. (2013). Teamwork with limited knowledge of teammates. In *AAAI Conference on Artificial Intelligence*, volume 27.
- Belle, V., Bolander, T., Herzig, A., and Nebel, B. (2023). Epistemic planning: Perspectives on the special issue. *Artificial Intelligence*, 316:103842.
- Bender, E. M., Gebru, T., McMillan-Major, A., and Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA. Association for Computing Machinery.
- Blount, J., Gelfond, M., and Balduccini, M. (2014). Towards a theory of intentional agents. *AAAI Spring Symposium - Technical Report*, pages 10–17.
- Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281–300.
- Bolander, T. and Andersen, M. B. (2011). Epistemic planning for single- and multi-agent systems. *Journal of Applied Non-Classical Logics*, 21(1):9–34.
- Bonet, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS'00, page 52–61. AAAI Press.
- Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33.
- Borgo, R., Cashmore, M., and Magazzeni, D. (2018). Towards providing explanations for AI planner decisions.
- Bowling, M. and McCracken, P. (2005). Coordination and adaptation in impromptu teams. In *National Conference on Artificial Intelligence*, page 53–58.
- Brachman, R. J. and Levesque, H. J. (2004). Knowledge representation and reasoning.
- Brenner, M. and Nebel, B. (2009). Continual planning and acting in dynamic multi-agent environments. *Autonomous Agents and Multi-Agent Systems*, 19(3):297–331.

- Brooks, D. R., Erdem, E., Erdogan, S. T., Minett, J. W., and Ringe, D. (2007). Inferring phylogenetic trees using answer set programming. *J. Autom. Reason.*, 39(4):471–511.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.
- Bylander, T. (1994). The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204.
- Cabalar, P., Fandinno, J., and Fink, M. (2014). Causal graph justifications of logic programs.
- Cao Tran, S., Pontelli, E., Balduccini, M., and Schaub, T. (2023). Answer set planning: A survey. *Theory and Practice of Logic Programming*, 23(1):226–298.
- Carnap, R. (1947). *Meaning and Necessity: A Study in Semantics and Modal Logic*. University of Chicago Press, Chicago, IL, USA.
- Chakraborti, T., Sreedharan, S., and Kambhampati, S. (2021). The emerging landscape of explainable automated planning & decision making. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI’20*.
- Chakraborti, T., Sreedharan, S., Zhang, Y., and Kambhampati, S. (2017). Plan explanations as model reconciliation: moving beyond explanation as soliloquy. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI’17*, page 156–163. AAAI Press.
- Charniak, E. and Goldman, R. P. (1993). A bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79.
- Chellas, B. F. (1980). *Introduction*, page 3–24. Cambridge University Press.

- Chen, S., Andrejczuk, E., Cao, Z., and Zhang, J. (2020). AATEAM: Achieving the ad hoc teamwork by employing the attention mechanism. In *AAAI*.
- Clancey, W. J. (1983). The epistemology of a rule-based expert system —a framework for explanation. *Artificial Intelligence*, 20(3):215–251.
- Clark, K. L. (1987). *Negation as failure*, pages 311 – 325. Springer, Boston, MA.
- Colmerauer, A. and Roussel, P. (1993). The birth of prolog. In *HOPL-II*.
- Deka, A. and Sycara, K. (2021). Natural emergence of heterogeneous strategies in artificially intelligent competitive teams. In Tan, Y. and Shi, Y., editors, *Advances in Swarm Intelligence*, pages 13–25, Cham. Springer International Publishing.
- Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.
- Dimopoulos, Y., Nebel, B., and Koehler, J. (19). Encoding planning problems in non-monotonic logic programs. In Steel, S. and Alami, R., editors, *Recent Advances in AI Planning*, pages 169–181, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Dodampegama, H. and Sridharan, M. (2024). <https://github.com/hharithaki/KAT>.
- Dodampegama, H. and Sridharan, M. (2026). <https://github.com/hharithaki/KAT-VH>.
- Ehsan, U., Passi, S., Liao, Q. V., Chan, L., Lee, I.-H., Muller, M., and Riedl, M. O. (2024). The who in xai: How ai background shapes perceptions of ai explanations. In *CHI Conference on Human Factors in Computing Systems*.
- Erdem, E., Aker, E., and Patoglu, V. (2012). Answer set programming for collaborative housekeeping robotics: representation, reasoning, and execution. *Intell. Serv. Robot.*, 5(4):275–291.
- Erdem, E. and Patoglu, V. (2018). Applications of ASP in Robotics. *Kunstliche Intelligenz*, 32(2-3):143–149.
- Fagin, R., Halpern, J. Y., Moses, Y., and Vardi, M. (1995). *Reasoning About Knowledge*. MIT Press, Cambridge.

- Fandinno, J. and Schulz, C. (2018). Answering the “why” in answer set programming – a survey of explanation approaches. *Theory and Practice of Logic Programming*, 19:114 – 203.
- Fang, Q., Zeng, J., Xu, H., Hu, Y., and Yin, Q. (2024). Learning ad hoc cooperation policies from limited priors via meta-reinforcement learning. *Applied Sciences*, 14(8).
- Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208.
- Fox, M., Long, D., and Magazzeni, D. (2017). Explainable Planning. In *IJCAI Workshop on Explainable AI*.
- Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2017). Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811.
- Geib, C. W. and Goldman, R. P. (2009). A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132.
- Gelfond, M. and Incelezan, D. (2013a). Some properties of system descriptions of al. *Journal of Applied Non-Classical Logics*, 23(1-2):105–120.
- Gelfond, M. and Incelezan, D. (2013b). Some Properties of System Descriptions of AL_d . *Applied Non-Classical Logics, Special Issue on Equilibrium Logic and ASP*, 23(1-2).
- Gelfond, M. and Kahl, Y. (2014). *Knowledge Representation, Reasoning and the Design of Intelligent Agents*. Cambridge University Press.
- Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R., Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press.
- Gelfond, M. and Lifschitz, V. (1993). Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2):301–321. Special Issue: Non-Monotonic Reasoning and Logic Programming.
- Gelfond, M. and Lifschitz, V. (1999). Action languages. *ETAI*, 3.

- Genesereth, M. R. and Nilsson, N. J. (1990). Logical foundations of artificial intelligence. *Journal of Symbolic Logic*, 55(3):1304–1307.
- Ghallab, M., Nau, D., and Traverso, P. (2004). The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, Burlington.
- Gigerenzer, G. (2016). *Towards a Rational Theory of Heuristics*, pages 34–59. Palgrave Macmillan UK, London.
- Gigerenzer, G. (2020). What is Bounded Rationality? In *Routledge Handbook of Bounded Rationality*. Routledge.
- Gigerenzer, G. and Gaissmaier, W. (2011). Heuristic Decision Making. *Annual Review of Psychology*, 62.
- Gigerenzer, G., Todd, P. M., and Group, A. B. C. R. (1999). *Simple Heuristics That Make Us Smart*. New York, NY, USA.
- Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *International Conference on Machine Learning*.
- Gomez, R., Sridharan, M., and Riley, H. (2020). Towards a theory of intentions for human-robot collaboration. In Bassiliades, N., Chalkiadakis, G., and de Jonge, D., editors, *Multi-Agent Systems and Agreement Technologies*, pages 3–19, Cham. Springer International Publishing.
- Goodman, B. and Flaxman, S. (2017). European union regulations on algorithmic decision making and a “right to explanation”. *AI Magazine*, 38(3):50–57.
- Gopalan, N., Rosen, E., Konidaris, G., and Tellex, S. (2020). Simultaneously learning transferable symbols and language groundings from perceptual data for instruction following. In Toussaint, M., Bicchi, A., and Hermans, T., editors, *Robotics: Science and Systems XVI, Virtual Event / Corvallis, Oregon, USA, July 12-16, 2020*.
- Green, C. (1969). Theorem proving by resolution as a basis for question-answering systems. in: B. Meltzer and D. Michie (eds.), *Machine Intelligence*, pages pp. 183–205.
- Green, L. and Mehr, D. R. (1997). What alters physicians’ decisions to admit to the coronary care unit? *The Journal of family practice*, 45(3):219–26.

- Gu, Y., Dong, L., Wei, F., and Huang, M. (2026). Minillm: On-policy distillation of large language models.
- Guan, L., Valmeekam, K., Sreedharan, S., and Kambhampati, S. (2023). Leveraging pre-trained large language models to construct and utilize world models for model-based task planning. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S., editors, *Advances in Neural Information Processing Systems*, volume 36, pages 79081–79094. Curran Associates, Inc.
- Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., and Pedreschi, D. (2018). A survey of methods for explaining black box models. *ACM Comput. Surv.*, 51(5).
- Halpern, J. Y. and Pearl, J. (2005). Causes and explanations: A structural-model approach. part i: Causes. *The British Journal for the Philosophy of Science*, 56(4):843–887.
- Hanheide, M., Göbelbecker, M., Horn, G. S., Pronobis, A., Sjöö, K., Aydemir, A., Jensfelt, P., Gretton, C., Dearden, R., Janicek, M., Zender, H., Kruijff, G.-J., Hawes, N., and Wyatt, J. L. (2017). Robot task planning and explanation in open and uncertain worlds. *Artificial Intelligence*, 247:119–150. Special Issue on AI and Robotics.
- Hausknecht, M., Mupparaju, P., Subramanian, S., Kalyanakrishnan, S., and Stone, P. (2016). Half field offense: An environment for multiagent learning and ad hoc teamwork. In *AAMAS Adaptive Learning Agents Workshop*.
- Hayes, P. J. (1973). Computation and deduction. In *International Symposium on Mathematical Foundations of Computer Science*.
- Hintikka, K. J. J. (1962). *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, Ithaca, NY, USA.
- Hoffman, R. R., Mueller, S. T., Klein, G., and Litman, J. (2018). Metrics for explainable ai: Challenges and prospects. *ArXiv*, abs/1812.04608.
- Hoffmann, J. and Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.
- Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. (2019). Parameter-efficient transfer learning for

- NLP. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR.
- Hsieh, C.-Y., Li, C.-L., Yeh, C.-k., Nakhost, H., Fujii, Y., Ratner, A., Krishna, R., Lee, C.-Y., and Pfister, T. (2023). Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. In Rogers, A., Boyd-Graber, J., and Okazaki, N., editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 8003–8017, Toronto, Canada. Association for Computational Linguistics.
- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). Lora: Low-rank adaptation of large language models. In *ICLR 2022*.
- Huang, L., Yu, W., Ma, W., Zhong, W., Feng, Z., Wang, H., Chen, Q., Peng, W., Feng, X., Qin, B., and Liu, T. (2025). A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2).
- Incelezan, D. and Gelfond, M. (2016). Modular action language $\neg lm$. *Theory and Practice of Logic Programming*, 16(2):189–235.
- Jennings, N., Faratin, P., Lomuscio, A., Parsons, S., Wooldridge, M., and Sierra, C. (2001). Automated negotiation: Prospects, methods and challenges. *Group Decision and Negotiation*, 10(2):199–215.
- Jenny, M. A., Pachur, T., Williams, S. L., Becker, E., and Margraf, J. (2013). Simple rules for detecting depression. *Journal of Applied Research in Memory and Cognition*, 2(3):149–157.
- Jiang, Y., Zhang, S., Khandelwal, P., and Stone, P. (2019). Task planning in robotics: an empirical comparison of pddl- and asp-based systems. *Frontiers Inf Technol Electronic Eng*, 20:363–373.
- Kambhampati, S., Valmeekam, K., Guan, L., Verma, M., Stechly, K., Bhambri, S., Saldyt, L., and Murthy, A. (2024). Position: LLMs Can’t Plan, But Can Help Planning in LLM-Modulo Frameworks. In *International Conference on Machine Learning*.

- Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models.
- Katsikopoulos, K., Simsek, O., Buckmann, M., and Gigerenzer, G. (2021). *Classification in the Wild: The Science and Art of Transparent Decision Making*. MIT Press.
- Kautz, H. A. and Allen, J. F. (1986). Generalized plan recognition. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, AAAI'86, page 32–37. AAAI Press.
- Kautz, H. A. and Selman, B. (1992). Planning as satisfiability. In *European Conference on Artificial Intelligence*.
- Keller, N. and Katsikopoulos, K. V. (2016). On the role of psychological heuristics in operational research; and a demonstration in military stability operations. *European Journal of Operational Research*, 249(3):1063–1073.
- Keren, S., Gal, A., and Karpas, E. (2020). Goal recognition design - survey. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4847–4853. International Joint Conferences on Artificial Intelligence Organization.
- Kerkez, B. and Cox, M. M. (2002). Case-based plan recognition with incomplete plan libraries.
- Khandelwal, P., Yang, F., Leonetti, M., Lifschitz, V., and Stone, P. (2014). Planning in action language bc while learning action costs for mobile robots. *Proceedings of the International Conference on Automated Planning and Scheduling*, 24(1):472–480.
- Konidaris, G. D., Kaelbling, L. P., and Lozano-Perez, T. (2018). From skills to symbols: Learning symbolic representations for abstract high-level planning. *J. Artif. Intell. Res.*, 61:215–289.
- Kowalski, R. and Kim, J.-S. (1991). A metalogic programming approach to multi-agent knowledge and belief. In Lifschitz, V., editor, *Artificial and Mathematical Theory of Computation*, pages 231–246. Academic Press.

- Kowalski, R. A. (1974). Predicate logic as programming language. In Rosenfeld, J. L., editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 569–574. North-Holland.
- Kraus, S. (2001). *Automated Negotiation and Decision Making in Multiagent Environments*, pages 150–172. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Kripke, S. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94.
- Lalwani, A., Kim, T., Chopra, L., Hahn, C., Jin, Z., and Sachan, M. (2025). Autoformalizing natural language to first-order logic: A case study in logical fallacy detection.
- Langley, P. (2019). Explainable, Normative, and Justified Agency. In *AAAI Conference on Artificial Intelligence*, pages 9775–9779.
- Langley, P., Meadows, B., Sridharan, M., and Choi, D. (2017). Explainable Agency for Intelligent Autonomous Systems. In *Innovative Applications of Artificial Intelligence*, San Francisco, USA.
- Law, M., Russo, A., and Broda, K. (2018). The Complexity and Generality of Learning Answer Set Programs. *Artificial Intelligence*, 259:110–146.
- Law, M., Russo, A., and Broda, K. (2020). The ILASP system for Inductive Learning of Answer Set Programs. *CoRR*.
- Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., and Scarcello, F. (2006). The dl_v system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562.
- Lester, B., Al-Rfou, R., and Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t., editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.
- Levesque, H. J. and Lakemeyer, G. (2008). Cognitive robotics. In van Harmelen, F., Lifschitz, V., and Porter, B. W., editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 869–886. Elsevier.

- Lewis, C. I. and Landford, C. H. (1934). Symbolic logic. *Philosophy of Science*, 1(2):239–246.
- Li, X. L. and Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. In Zong, C., Xia, F., Li, W., and Navigli, R., editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.
- Lifschitz, V. (1999). Answer set planning. In De Schreye, D., editor, *Logic Programming: Proceedings of the 1999 International Conference on Logic Programming*. The MIT Press.
- Lifschitz, V. (2008). What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3, AAAI'08*, page 1594–1597. AAAI Press.
- Liu, X., Li, P., Yang, W., Guo, D., and Liu, H. (2024). Leveraging Large Language Model for Heterogeneous Ad Hoc Teamwork Collaboration. In *Robotics: Science and Systems*, Delft, The Netherlands.
- Lu, S., Bigoulaeva, I., Sachdeva, R., Madabushi, H. T., and Gurevych, I. (2024). Are Emergent Abilities in Large Language Models just In-Context Learning? In *Annual Meeting of the Association for Computational Linguistics*, pages 5098–5139, Bangkok, Thailand.
- Macke, W., Mirsky, R., and Stone, P. (2021). Expected value of communication for planning in ad hoc teamwork. In *AAAI Conference on Artificial Intelligence*.
- Martignon, L., Katsikopoulos, K. V., and Woike, J. K. (2008). Categorization with limited resources: A family of simple heuristics. *Journal of Mathematical Psychology*, 52(6):352–361.
- Martignon, L., Vitouch, O., Takezawa, M., and Forster, M. R. (2003). *Naive and Yet Enlightened: From Natural Frequencies to Fast and Frugal Decision Trees*, chapter 10, pages 189–211. John Wiley & Sons, Ltd.
- McAllester, D. A. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *AAAI Conference on Artificial Intelligence*.

- McCarthy, J. (1959). Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London. Her Majesty’s Stationary Office.
- McCarthy, J. (1980). Circumscription—a form of non-monotonic reasoning. *Artif. Intell.*, 13(1–2):27–39.
- McDermott, D. and Doyle, J. (1980). Non-monotonic logic i. *Artificial Intelligence*, 13(1):41–72. Special Issue on Non-Monotonic Logic.
- McDermott, D., Ghallab, M., Howe, A. E., Knoblock, C. A., Ram, A., Veloso, M. M., Weld, D. S., and Wilkins, D. E. (1998). Pddl-the planning domain definition language.
- Melo, F. S. and Sardinha, A. (2016). Ad hoc teamwork by learning teammates’ task. *Autonomous Agents and Multi-Agent Systems*, 30(2):175–219.
- Meneguzzi, F. and Fraga Pereira, R. (2021). A survey on goal recognition as planning. In Zhou, Z.-H., editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 4524–4532. International Joint Conferences on Artificial Intelligence Organization.
- Miller, G. A. (1995). Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41.
- Miller, T. (2019). Explanations in Artificial Intelligence: Insights from the Social Sciences. *Artificial Intelligence*, 267:1–38.
- Mirsky, R., Carlucho, I., Rahman, A., Fosong, E., Macke, W., Sridharan, M., Stone, P., and Albrecht, S. (2022). A Survey of Ad Hoc Teamwork: Definitions, Methods, and Open Problems. In *European Conference on Multiagent Systems*.
- Mirsky, R., Macke, W., Wang, A., Yedidsion, H., and Stone, P. (2020). A penny for your thoughts: The value of communication in ad hoc teamwork. In *International Joint Conference on Artificial Intelligence*, pages 254–260.
- Mirsky, R., Stern, R., Gal, Y., and Kalech, M. (2017). Plan recognition design. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31.
- Moore, R. C. (1985). Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25(1):75–94.

- Mota, T., Sridharan, M., and Leonardis, A. (2021). Integrated Commonsense Reasoning and Deep Learning for Transparent Decision Making in Robotics. *Springer Nature CS*, 2(242).
- Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., and Yeoh, W. (2017). Generalized target assignment and path finding using answer set programming. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17*, page 1216–1223. AAAI Press.
- Niemelä, I. and Simons, P. (1997). Smodels — an implementation of the stable model and well-founded semantics for normal logic programs. In Dix, J., Furbach, U., and Nerode, A., editors, *Logic Programming And Nonmonotonic Reasoning*, pages 420–429, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., and Barry, M. (2001). An a-prolog decision support system for the space shuttle. In Ramakrishnan, I. V., editor, *Practical Aspects of Declarative Languages*, pages 169–183, Berlin, Heidelberg. Springer Berlin Heidelberg.
- OpenAI and team (2024). GPT-4 Technical Report. *arXiv:2303.08774*.
- Pan, L., Albalak, A., Wang, X., and Wang, W. (2023). Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In Bouamor, H., Pino, J., and Bali, K., editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.
- Pearl, J. and Mackenzie, D. (2018). *The book of why : the new science of cause and effect*. Allen Lane.
- Pednault, E. P. D. (1994). ADL and the State-Transition Model of Action. *Journal of Logic and Computation*, 4(5):467–512.
- Peot, M. A. and Smith, D. E. (1992). Conditional nonlinear planning. In Hendler, J., editor, *Artificial Intelligence Planning Systems*, pages 189–197. Morgan Kaufmann, San Francisco (CA).
- Pereira, R. F., Oren, N., and Meneguzzi, F. (2020). Landmark-based approaches for goal recognition as planning. *Artificial Intelligence*, 279:103217.

- Petrick, R. P. A. and Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, AIPS'02, page 212–221. AAAI Press.
- Petrick, R. P. A. and Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, ICAPS'04, page 2–11. AAAI Press.
- Phillips, N. D., Neth, H., Woike, J. K., and Gaissmaier, W. (2017). Fftrees: A toolbox to create, visualize, and evaluate fast-and-frugal decision trees. *Judgment and Decision Making*, 12(4):344–368.
- Pontelli, E., Son, T. C., and Elkhathib, O. (2009). Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56.
- Pryor, L. and Collins, G. (1996). Planning for contingencies: a decision-based approach. *J. Artif. Int. Res.*, 4(1):287–339.
- Puig, X., Ra, K., Boben, M., Li, J., Wang, T., Fidler, S., and Torralba, A. (2018). Virtualhome: Simulating household activities via programs. In *International Conference on Computer Vision and Pattern Recognition*, pages 8494–8502.
- Radford, A. and Narasimhan, K. (2018). Improving language understanding by generative pre-training.
- Rahman, M. A., Hopner, N., Christianos, F., and Albrecht, S. V. (2021). Towards open ad hoc teamwork using graph-based policy learning. In *International Conference on Machine Learning*, pages 8776–8786.
- Rahwan, I., Ramchurn, S. D., Jennings, N. R., McBurney, P., Parsons, S., and Sonenberg, L. (2003). Argumentation-based negotiation. *The Knowledge Engineering Review*, 18(4):343–375.
- Ramírez, M. and Geffner, H. (2009). Plan recognition as planning. In *International Joint Conference on Artificial Intelligence*.
- Ramírez, M. and Geffner, H. (2010). Probabilistic plan recognition using off-the-shelf classical planners. *Proceedings of the AAAI Conference on Artificial Intelligence*.

- Ravula, M., Alkoby, S., and Stone, P. (2019). Ad hoc teamwork with behavior switching agents. In *International Joint Conference on Artificial Intelligence*.
- Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence*, 13(1):81–132. Special Issue on Non-Monotonic Logic.
- Reiter, R. (1987). Nonmonotonic reasoning. *Annual Review of Computer Science*, 2(Volume 2, 1987):147–186.
- Ribeiro, J. G., Martinho, C., Sardinha, A., and Melo, F. S. (2022). Assisting unknown teammates in unknown tasks: Ad hoc teamwork under partial observability. *International Conference on Autonomous Agents and Multiagent Systems*.
- Ribeiro, J. G., Rodrigues, G., Sardinha, A., and Melo, F. S. (2023). Teamster: Model-based reinforcement learning for ad hoc teamwork. *Artificial Intelligence*, 324:104013.
- Rudin, C. (2019). Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead. *Nature Machine Intelligence*, 1:206–215.
- Rudin, C., Chen, C., Chen, Z., Huang, H., Semenova, L., and Zhong, C. (2022). Interpretable machine learning: Fundamental principles and 10 grand challenges. *Statistics Surveys*, 16:1–85.
- Russell, S. and Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson.
- Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *International Joint Conference on Artificial Intelligence*.
- Santos, P. M., Ribeiro, J. G., Sardinha, A., and Melo, F. S. (2021). Ad hoc teamwork in the presence of non-stationary teammates. In Marreiros, G., Melo, F. S., Lau, N., Lopes Cardoso, H., and Reis, L. P., editors, *Progress in Artificial Intelligence*. Springer International.
- Schulz, C. and Toni, F. (2016). Justifying answer sets using argumentation. *Theory and Practice of Logic Programming*, 16(1):59–110.
- Shanahan, M. (1993). Explanation in the situation calculus. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI’93*, page 160–165, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

- Shortliffe, E. H., Davis, R., Axline, S. G., Buchanan, B. G., Green, C., and Cohen, S. N. (1975). Computer-based consultations in clinical therapeutics: Explanation and rule acquisition capabilities of the mycin system. *Computers and Biomedical Research*, 8(4):303–320.
- Simon, H. A. (1956). Rational Choice and the Structure of the Environment. *Psychological Review*, 63:129–138.
- Simon, H. A. (1990). Invariants of human behavior. *Annual review of psychology*, 41:1–19.
- Simon, H. A., Klahr, D., Kotovsky, K., and on Cognition (21st : Carnegie-Mellon University), C. S. (1989). Complex information processing : the impact of herbert a. simon.
- Smith, D. E. and Weld, D. S. (1998). Conformant graphplan. In Mostow, J. and Rich, C., editors, *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 889–896. AAAI Press / The MIT Press.
- Smith, R. G. (1980). The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29:1104–1113.
- Son, T. and Balduccini, M. (2018). Answer set planning in single- and multi-agent environments. *Künstliche Intelligenz*, 32.
- Son, T. C. and Pontelli, E. (2004). Planning with preferences using logic programming. In Lifschitz, V. and Niemelä, I., editors, *Logic Programming and Nonmonotonic Reasoning*, pages 247–260, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Son, T. C., Pontelli, E., and Nguyen, N.-H. (2010). Planning for multiagent using asp-prolog. In Dix, J., Fisher, M., and Novák, P., editors, *Computational Logic in Multi-Agent Systems*, pages 1–21. Springer Berlin Heidelberg.
- Son, T. C. and Sakama, C. (2010a). Reasoning and planning with cooperative actions for multiagents using answer set programming. In Baldoni, M., Bentahar, J., van Riemsdijk, M. B., and Lloyd, J., editors, *Declarative Agent Languages and Technologies VII*, pages 208–227, Berlin, Heidelberg. Springer Berlin Heidelberg.

- Son, T. C. and Sakama, C. (2010b). Reasoning and Planning with Cooperative Actions or Multiagents using Answer Set Programming. In *Declarative Agent Languages and Technologies VII*, volume 5948 of *LNCS*, pages 208–227. Springer Berlin Heidelberg.
- Sridharan, M., Gelfond, M., Zhang, S., and Wyatt, J. (2019). REBA: A Refinement-Based Architecture for Knowledge Representation and Reasoning in Robotics. *Journal of Artificial Intelligence Research*, 65:87–180.
- Sridharan, M. and Meadows, B. (2018). Knowledge Representation and Interactive Learning of Domain Knowledge for Human-Robot Collaboration. *Advances in Cognitive Systems*, 7:77–96.
- Sridharan, M. and Meadows, B. (2019). Towards a theory of explanations for human–robot collaboration. *KI - Künstliche Intelligenz*, 33:1–12.
- Sridharan, M., Meadows, B., and Gomez, R. (2017). What Can I Not Do? Towards an Architecture for Reasoning about and Learning Affordances. In *International Conference on Automated Planning and Scheduling*, pages 461–469.
- Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer*. MIT Press.
- Stone, P., Kaminka, G., Kraus, S., and Rosenschein, J. (2010). Ad Hoc Autonomous Agent Teams: Collaboration without Pre-Coordination. In *AAAI Conference on Artificial Intelligence*, pages 1504–1509.
- Subrahmanian, V. and Zaniolo, C. (1995). Relating stable models and ai planning domains. In Sterling, L. S., editor, *Logic Programming: The 12th International Conference*. The MIT Press.
- Tate, A. (1977). Generating project networks. In *International Joint Conference on Artificial Intelligence*.
- Todd, P. M. and Gigerenzer, G. (2012). *Ecological Rationality: Intelligence in the World*. Oxford University Press.
- Trieu, L. L., Son, T. C., and Balduccini, M. (2021). exp(asp): Explaining asp programs with choice atoms and constraint rules. *Electronic Proceedings in Theoretical Computer Science*, 345:155–161.

- Turner, H. (1997). Representing actions in logic programs and default theories a situation calculus approach. *The Journal of Logic Programming*, 31(1):245–298. Reasoning about Action and Change.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.
- Vered, M., Kaminka, G., and Biham, S. (2016). Online goal recognition through mirroring: humans and agents. In Forbus, K., Hinrichs, T., and Ost, C., editors, *Fourth Annual Conference on Advances in Cognitive Systems*, Advances in Cognitive Systems. Cognitive Systems Foundation.
- Vered, M. and Kaminka, G. A. (2017). Heuristic online goal recognition in continuous domains.
- Vered, M., Pereira, R., Magnaguagno, M. C., Kaminka, G. A., and Meneguzzi, F. (2018). Towards online goal recognition combining goal mirroring and landmarks. In *Adaptive Agents and Multi-Agent Systems*.
- Viegas Damásio, C., Analyti, A., and Antoniou, G. (2013). Justifications for logic programming. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning - Volume 8148*, LPNMR 2013, page 530–542, Berlin, Heidelberg. Springer-Verlag.
- White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt.
- Woike, J., Hoffrage, U., and Martignon, L. (2017). Integrating and testing natural frequencies, naïve bayes, and fast-and-frugal trees. *Decision*, 4(4):234–260.
- Wu, F., Zilberstein, S., and Chen, X. (2011). Online Planning for Ad Hoc Autonomous Agent Teams. In *International Joint Conference on Artificial Intelligence*, page 439–445.
- Xu, P., Zhang, Y., Hao, L., and Yan, Q. (2025). DETEAMSK: A Model-Based Reinforcement Learning Approach to Intelligent Top-Level Planning and Decisions for

Multi-Drone Ad Hoc Teamwork by Decoupling the Identification of Teammate and Task. *Aerospace*, 12(7):1–32.

Yang, Y., Xiong, S., Payani, A., Shareghi, E., and Fekri, F. (2024). Harnessing the power of large language models for natural language to first-order logic translation. In Ku, L.-W., Martins, A., and Srikumar, V., editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand. Association for Computational Linguistics.

Yourdshahi, E. S., Pinder, T., Dhawan, G., Marcolino, L. S., and Angelov, P. P. (2018). Towards Large Scale Ad-hoc Teamwork. In *IEEE International Conference on Agents*, pages 44–49.

Zach, R. (2019). *Boxes and Diamonds: An Open Introduction to Modal Logic*. Open Logic Project.

Zand, J., Parker-Holder, J., and Roberts, S. J. (2022). On-the-fly strategy adaptation for ad-hoc agent coordination. In *International Conference on Autonomous Agents and Multiagent Systems*, page 1771–1773.

Zintgraf, L., Devlin, S., Ciosek, K., Whiteson, S., and Hofmann, K. (2021). Deep interactive bayesian reinforcement learning via meta-learning. In *International Conference on Autonomous Agents and Multiagent Systems*.