Expanding Task Diversity in Explanation-Based Interactive Task Learning

by

Aaron Mininger

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2021

Doctoral Committee:

Professor John E. Laird, Chair
Professor Joyce Chai
Professor Chad Jenkins
Professor Colleen Seifert

Aaron Mininger

mininger@umich.edu

ORCID iD: 0000-0003-1464-0650

©Aaron Mininger 2021

ACKNOWLEDGMENTS

I would like to thank my research advisor, John Laird, whose mentorship and hard work has been a significant contribution. I am very grateful for the support and grace he has shown me over the years. I also appreciate his great ability to bring people together, both within the lab and within the broader field. A major thank you to my committee and their work and flexibility. I am also grateful for the help of my undergraduate advisor, William Birmingham, in making it into graduate school.

A project the size and scope of Rosie is only made possible due to the many collaborators who have contributed to it over the years. Thank you to Shiwali Mohan, whose work laid the foundation for this entire dissertation, and to James Kirk, who was crucial to a lot of the core implementation. Thank you to John Laird and Peter Lindes for their work with language comprehension, Mazin Assanie for his improvements to chunking that made this work possible, and for Edwin Olsen and Rob Goeddel for their robotic and simulation tools. I am also grateful to the community of students within the Soar lab, both older and newer, who have aided in countless ways.

There are so many wonderful people whose support has helped me achieve this goal; all my friends throughout the years at Grace Church, the Kimballs for housing me in the midst of a pandemic, Shawn and Becca for their love and friendship. Special appreciation to Ben, who helped me through hard times and gave me inspiration for the future. Lastly, I could not have done this without the love of my family, even when I did not make it easy. Thank you to my extended family, for their encouragement and unwavering faith, and to my sister, Emily, and mom and dad, who mean the world.

TABLE OF CONTENTS

Chapter

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

ABSTRACT

The possibility of having artificial agents that can interact with humans and learn completely new tasks through instruction and demonstration is an exciting prospect. This is the goal of the emerging research area of Interactive Task Learning. Solving this problem requires integrating many capabilities across AI to create general robot learns that can operate in a variety of environments. One particular challenge is that the space of possible tasks is extremely large and varied. Developing approaches that cover this space is a difficult challenge, made more so by having to learn from a limited, though high-quality, number of examples given through interaction with a teacher.

In this dissertation, we identify three major dimensions of task complexity (diverse types of actions, task formulations, and task modifiers), and describe extensions that demonstrate greater learning capabilities for each dimension than previous work. First, we extend the representations and learning mechanism for innate tasks so the agent can learn tasks that utilize many different types of actions beyond physical object manipulation, such as communication and mental operations. Second, we implement a novel goal-graph representation that supports both goal-based and procedural tasks. Thus the instructor can formulate a task as achieving a goal and let the agent use planning to execute it, or can formulate the task as executing a procedure, or sequence of steps, when it is not easy to define a goal. This also allows interesting cases of a task that blends elements of a procedure and goal. Third, we added support for learning subtasks with various modifying clauses, such as temporal constraints, conditions, or looping structures. Crucially, we show that the agent can learn and generalize a canonical version of a task and then combine it with these various modifiers within a task hierarchy without requiring additional instruction.

This is done in the context of Rosie – an agent implemented within the Soar cognitive architecture that can learn completely new tasks in one shot through situated interactive instruction. By leveraging explanation-based generalization and domain knowledge, the agent quickly learns new hierarchical tasks, including their structure, arguments, goals, execution policies, and task decompositions, through natural language instruction. It has been used with various robotic platforms, though most of the learning demonstrations and evaluations in this work use a simulated mobile robot in a multi-room, partially-observable environment. In the end, we show that the agent can combine all of these extensions while learning complex hierarchical tasks that cover extended periods of time and demonstrate significant flexibility.

CHAPTER 1

Introduction

We are in the early stages of a technological revolution as artificially intelligent agents with interaction capabilities are becoming ever more prevalent. The first wave was virtual personal assistants embedded in phones or smart devices that can answer questions and perform software-based tasks. These capabilities are being extended to more embodied devices that can interact with the physical world, such as appliances and autonomous vehicles. Eventually, one can envision general-purpose, personal robots being deployed in a variety of settings, including domestic, healthcare, and industrial. These have the potential to greatly improve people's lives by taking care of menial labor, giving more capabilities to those who are physically impaired, and improving manufacturing efficiency and thereby lowering the costs of goods.

Developing such general-purpose robots that can operate across complex and diverse environments is an extremely difficult challenge. One particularly challenging aspect is that the environmental and task diversity makes it almost impossible to pre-program or pre-train these robots with all the behavior desired of them. In addition, people will want to direct, customize, and extend their capabilities over time, and do so easily without needing to be an expert programmer or providing hundreds of demonstrations. Fortunately, humans already possess great ability to teach others quickly through instruction and demonstration, and it is the goal of the research problem of Interactive Task Learning (ITL) to leverage this teaching ability to develop agents that learn new tasks from humans through natural forms of interaction.

Interactive Task Learning is an emerging A.I. challenge problem, defined as "any process by which an agent improves its performance on some task through experience, when [that experience] consists of a series of sensing, effecting, and communicating interactions between [the agent], its world, and crucially other agents in the world" (Gluck & Laird, 2019). Importantly, one of those other agents should have the goal of improving the first agent's performance on the task. Examples of ITL include watching a person demonstrate how to make tea, having someone move a robotic arm to show it how to close a box, or listening to an instructor give directions for mailing a letter. One subclass of ITL is learning through Situated Interactive Instruction (SII), where the instructor and the agent are *situated* in a shared environment, both the instructor and agent are engaged in a bidirectional *interaction*, and where the primary form of communication is natural language *instruction*. This is a powerful way of learning, as language enables a skilled instructor to provide curated, high quality information that is specific to the current situation and precise learning goals of the agent.

However, learning a task through Situated Interactive Instruction presents a number of difficult challenges. Since human instruction is time-consuming and the number of examples are limited, the agent must learn quickly and extract a large amount of useful information from few examples. Building an ITL agent that can learn from instruction requires the integration of a number of diverse capabilities that themselves are difficult, including perception, action, natural language understanding, interpretation, grounding, and dialog management. In addition, the agent will require a large amount of existing knowledge about the world and itself to understand and learn from instructions. This challenge is made more difficult by the sheer complexity and diversity of tasks that humans will want to teach. The agent will need to learn tasks with wildly different characteristics, featuring diverse types of objects, concepts, actions, and instructional strategies. However, much of the current research in instructional ITL is limited to simple tasks that do not feature this complexity, with many being variations on pick and place tasks (Frasca et al., 2018; Suddrey et al., 2016; She et al., 2014).

In this dissertation, we investigate how an agent learning from Situated Interactive Instruction can acquire diverse tasks that vary along three main dimensions of complexity and diversity: *action types*, *task formulations*, and *task modifiers*. First, tasks can vary in the types of primitive actions that are involved. A simple task such as *'Move the red block onto the green block'* only involves physical actions, whereas a task such as *'Tell Sam a message'* involves other types of actions, as it must remember the message (mental), find Sam (perceptual), and say the message (communicative). Second, tasks can vary in how they are formulated, such as achieving a goal (clearing the table), performing a procedure (following a brownie recipe), or maximizing an objective function (finding the fastest route to the store). Third, a single task can be modified in different ways with additional clauses, such as specifying when a task should be performed (in fifteen minutes), whether it should be performed (if the room is empty), or how many times. By developing an approach that learns tasks with these diverse characteristics, we significantly expand the space of learnable tasks beyond previous approaches.

The main research questions addressed by this thesis are as follows:

1. *What is a useful characterization of the space of learnable tasks for ITL?*

   Answering this question involves defining what a task is and identifying different dimensions of task complexity and diversity that span the space of possible tasks. These dimensions serve as a way to measure the task learning capabilities of an agent and method to compare different agents.

2. *How can diverse types of primitive actions be integrated into an ITL agent?*

   General-purpose agents will need to learn and execute tasks that involve more than just the physical manipulation of objects. Many common tasks include actions that involve perceptual acts (checking if the lights are on), communicative acts (making an announcement), or mental acts (recalling that milk is stored in the fridge). Integrating these types of actions will require identifying representations and models for primitive actions that work across different action types.

3. *How can an ITL agent learn tasks with diverse formulations?*

   Given the large diversity of task characteristics, a single type of task formulation (which includes the specific representations, learning methods, and execution approaches used) is unlikely to be effective for all tasks. Answering this question will require integrating different task formulations within the same agent using a unified representation that supports blending and mixing different formulations.

4. *How can an ITL agent generalize task knowledge across diverse variations?*

   A single task can be modified by additional clauses and terms that specify, modify, and constrain the task (e.g. where, when, or how the task should be done). So while there may be a simplest, canonical version of a task, an agent should be able to learn and perform these variations with minimal additional instruction. A solution will require significant generalization and reasoning capabilities.

5. *What organization of knowledge and processing supports one-shot ITL?*

   Developing learning methods for acquiring general task knowledge through a single interaction is a challenging problem, especially if they need to cover a diverse range of tasks. This requires that the learning methods are online, real-time, incremental, and general. Addressing this question involves constructing semantic representations of the knowledge obtained through interaction, and then operationalizing that knowledge so it can be utilized efficiently.

In this work, we focus on extending the task representations and learning methods that encode the semantics of a task and that support a large space of possible tasks. We leave the question of how to support high-quality, flexible interaction to other work. While the agent does learn from mixed-initiative, situated interactive instruction, the natural language used is restricted and sometimes esoteric. Questions such as how to make the agent more explainable, robust, and correctable are left for future work.

One incidental problem that we needed to solve was how to maintain a representation of the world that incorporated information from perception, instruction, task knowledge, semantic knowledge, and past experiences. Although this is not our core research goal, it still was an important problem to address to support the rest of the learning and processing. We include our approach in Chapter 5, as we believe it offers some interesting insights and methods. In the following section, we describe our approach for addressing these questions.

1.1 Research Approach

This dissertation addresses these research questions by extending the task learning capabilities of an Interactive Task Learning agent known as *Rosie*, which incorporates ideas from Huffman & Laird (1995) and an initial implementation by Mohan & Laird (2014). Rosie learns many different types of knowledge from Situated Interactive Instruction, including perceptual features and spatial relationships (Mohan et al., 2012), hierarchical state-based concepts (Kirk & Laird, 2019), games and puzzles (Kirk & Laird, 2016), goal-based tasks (Mohan & Laird, 2014), and procedural tasks (Mininger & Laird, 2018). Below are three central features of the SII approach to ITL used in Rosie.

1.1.1 Architectural Approach

A general task learning agent requires many different cognitive capabilities such as high-level reasoning, planning, and decision making. Cognitive architectures provide fixed mechanisms that support these kinds of capabilities. Rosie is developed in the Soar cognitive architecture (Laird, 2012), which contains a symbolic working memory and metric spatial memory, procedural and declarative long term memories, and several learning mechanisms including chunking, reinforcement learning, and deliberate semantic memory storage. Many of these are integral to Rosie's approach to ITL. In particular, Soar supports *impasse-driven learning*, where missing knowledge leads to impasses that allow the agent to reason about what caused the impasse and take steps to acquire the missing knowledge. Soar's chunking mechanism, based on explanation-based learning (DeJong & Mooney, 1986; Mitchell et al., 1986), allows the agent to learn new procedural knowledge that avoids the impasse in the future. This method of learning, along with Soar's emphasis on efficiency and reactivity, makes Soar an excellent architecture for developing robotic agents that learn from SII. A more detailed description of the Soar architecture and how its mechanisms are used by Rosie is provided in Chapter 4.

1.1.2 Integrated Approach

Addressing the problem of Interactive Task Learning requires integrating various AI capabilities together into an agent that produces end-to-end behavior. ITL is at the intersection of research in perception and action, interaction and language comprehension, and learning and planning. Developing approaches to task learning should not happen in isolation, but should be integrated with all of these general capabilities. Thus we develop and evaluate Rosie with regards to end-to-end behavior. Rosie has been applied to a number of real-world and simulated robotic domains, where it learns various tasks. Although we do take steps to simplify the domains, for example, restricting the grammar and vocabulary, types of actions it can perform, or perceptual difficulty of the environment, the task learning capabilities are developed and evaluated within the entire system. Integrating different components together can uncover additional issues and research challenges that do not appear in isolation. For example, when integrating Rosie with real-world robotic domains, we found that the agent needed additional capabilities to maintain a consistent world model in partially observable environments (Chapter 5).

1.1.3 Unified Approach

When adding task learning capabilities to an existing agent, there are different approaches one could take. For example, one could adopt a modular approach where different modules or components in a system handle different types of tasks. We choose to take a unified approach, where new capabilities are integrated within existing representations, problem spaces, and methods of learning as much as possible. This makes it easier for the agent to compose these capabilities, and to combine heterogeneous types of tasks in interesting ways, including vertically (a task of one type with a subtask of another), horizontally (a task with subtasks of different types), and blended (a task with characteristics of multiple formulations). The ability to compose different capabilities can lead to more task diversity, though it also increases the overall system complexity as a capability cannot be developed, tested, and evaluated in isolation. This unified approach is demonstrated throughout the task learning extensions described in this dissertation, including the common task representations used across diverse types of actions (Chapter 7) and a goal graph used across different formulations (Chapter 8) and modifiers (Chapter 9).

1.2 Contributions

By applying our research approach to address the questions posed earlier, this dissertation offers several contributions to the field of interactive task learning and has broader applications to agent design within cognitive architectures. These contributions are summarized as follows:

1. An agent that demonstrates the ability to learn a broader space of high-level tasks from one-shot instruction than other related work along the three dimensions of task complexity. The traits that particularly distinguish the work in this dissertation are the ability to use and represent mental and communicative actions as first-class tasks, the ability to represent tasks as achieving a goal *or* following a procedure, and the ability to learn temporal and conditional modifiers with both types of tasks.

2. Extensions to explanation-based policy learning that enable its use with more actions involving communication and mental operations, without requiring extensive modelling or domain knowledge.

3. A task representation that unifies both goal-based and procedural formulations. These have complimentary strengths and weaknesses, so supporting both broadens the space of learnable tasks, gives the instructor more flexibility, and allows the most suitable formulation to be used with any particular task.

4. A dual representation of task knowledge and a way of organizing processing that involves first constructing declarative structures from knowledge contained in natural language instruction and then interpreting that knowledge within the current situated context to compile it into procedural rules. This allows the agent to utilize the flexibility of declarative representations with the efficiency of productions. The novel aspect of our approach is that the intermediate structure is completely embedded in the rules.

5. A method for an agent in a cognitive architecture to integrate information from perception, natural language instructions, task knowledge, semantic world knowledge, and past experiences into a cohesive and stable representation of the world that supports the learning goals of the agent within large, partially observable environments.

These contributions are evaluated through the various methods below.

1.3 Evaluation Methodology

A major claim of this dissertation is that the extensions expand the space of tasks that an ITL agent can learn along these three dimensions of task complexity beyond both previous work and other related work. Characterizing this space of learnable tasks is challenging in and of itself, and a goal of this dissertation is to provide a framework for better defining and comparing what tasks different approaches can learn through Interactive Task Learning. The evaluation of this claim will consist of two major parts: demonstrating the agent's task learning abilities through experiments and comparing its capabilities to other related work. The following sections describe the different methods and experiments that will be used when evaluating this dissertation.

1.3.1 Case Studies

Case studies are a valuable tool to show how the agent learns a specific task and describe the internal processing and learned knowledge. These can demonstrate that the agent does successfully learn a task, although it is limited to a single instance. They are more useful to bring clarity to how the approach works and what is learned. They can also highlight cases where our integration of these new task learning capabilities in a unified manner leads to interesting and sometimes novel interactions between them (Section 8.4).

1.3.2 Task Transfer Experiments

One of the most significant strengths of our approach is its ability to generalize from one task instance and transfer that knowledge to many other variations on that task. When we claim that the agent successfully learned a task, we do not only mean that the agent can replicate that same exact task again in the future, but that it can perform it with different objects, modifiers, and starting conditions without needing further instruction. The agent may need to initiate further instruction if a task variation requires additional knowledge not taught in the first instance, though it is valuable to see when the agent is able to acquire that knowledge on its own. To measure the agent's generalization capabilities, we perform transfer experiments where we give one training instance of a task, then ask the agent to perform a set of task variations. This can include randomly generated starting conditions (Section 7.5) or different task modifiers (Section 9.6). We measure both whether it is ultimately successful at performing the task and how many additional instructions are needed.

1.3.3 Qualitative Analysis and Comparison

In this work, we develop a taxonomy for measuring the space of tasks that an ITL system can learn, including the three areas of task complexity addressed in this thesis. We perform an analysis of our approach according to this framework, making sure to identify how it improves beyond the previous version. This is a manually guided exploration of where the task-learning boundaries to our approach lie.

A significant challenge in evaluating work in ITL is that there are not many commonly used data sets, task lists, benchmarks, or environments that can be used to compare different approaches. We propose that having a framework for measuring the space of tasks that an ITL agent can learn can serve as a way of comparing the capabilities of different approaches.

1.4 Road map

The first two chapters give background on the problem of learning from instruction. In Chapter 2, we further describe the characteristics of learning from Situated Interactive Instruction and how it constraints the learning problem and suggests agent desiderata. We then discuss related work in Chapter 3.

The next three chapters describe how Rosie works and learns tasks. It provides the details necessary to understand the contributions and extensions of this thesis. Chapter 4 gives an overview of the Rosie agent. This includes describing the agent architecture, language comprehension, and interaction management. In Chapter 5, we describe how the agent constructs and maintains a model of the world, given unreliable perceptual information. The agent takes an active role in anchoring and can recover from certain errors. An overview of how the agent represents and executes tasks is given in Chapter 6.

The subsequent three chapters are the most central to describing the work of this thesis. They define the different dimensions of task complexity that this thesis addresses and our contributions in extending the learning capabilities of the agent along each. Chapter 7 tackles learning diverse types of actions beyond physical manipulation. Chapter 8 describes extending our approach to learning diverse types of task formulations, including procedural, optimization, and maintenance tasks. Chapter 9 includes our extensions to the agent's task representations and generalization capabilities in order to handle diverse task modifiers.

We conclude with a large-scale demonstration of the agent learning the task of doing a guard/patrol task in a barracks (Chapter 10), which pulls together all the capabilities in the previous chapters, and summarize our contributions and identifies limitations and future work in Chapter 11.

CHAPTER 2

Learning from Situated Interactive Instruction

Interactive task learning has a number of fundamental characteristics that define, constrain, and differentiate the learning problem. These features of the learning problem identify important desiderata for ITL agents to satisfy. They also distinguish ITL from other approaches and applications of machine learning. In this chapter, we identify the fundamental aspects of interactive task learning and how they impact the learning problem. During these discussions, we identify a number of important agent desiderata that can be used to evaluate the performance of an ITL agent. We close by distinguishing the features and desiderata that are the particular focus of this work. This analysis is influenced by Gluck & Laird (2019), especially chapters 1, 15, and 17.

2.1 Interactive Learning

Task learning is interactive if both the teacher and learner are engaged in a collaborative and cooperative activity with the purpose of having the learner achieve certain learning goals. Both parties must be actively engaged in the process, taking the initiative when appropriate, and adapting to how the interaction is unfolding. The teacher chooses which examples to present, what knowledge to communicate, and the method of instruction, and adjusts these according to feedback from the agent and evaluation of its learning progress. The agent continually tries to apply the knowledge, requesting additional information and asking for clarification or disambiguation as needed. Both parties must work towards maintaining a shared common ground and addressing obstacles as they arise. In the following sections, we go through several important features of interactive learning and the desiderata each one imposes on the agent.

2.1.1 Interaction is in Real Time

Unlike batch learning from large data sets, learning from interaction happens in real time. This offers a number of advantages and challenges. The teacher observes the agent and can offer time-critical feedback and corrections. They can also adapt their teaching strategy as necessary. The learner can apply the new knowledge as it attempts to learn and execute the task and ask for targeted help if problems arise. However, this presents a very challenging learning problem, as new knowledge must be incorporated in real time and be immediately available for use. The real time constraints leads to the following agent learning desiderata:

D1: *Immediate*
To fully take advantage of the interactive mode of learning, the agent should immediately and continually apply its knowledge to the current situation. This allows the instructor to monitor its behavior and provide corrective feedback if needed. Thus the agent's learning mechanisms must produce immediate, actionable results and should mostly use *online* learning techniques.

D2: *Incremental*
Another consequence of real time learning is that the agent is only exposed to one execution trace at a time. Therefore it may be unable to completely learn the task during the first interaction and must do so over multiple episodes. Therefore the task learning must be incremental and be able to update previous knowledge during subsequent executions.

D3: *Fast*
A third constraint from real time interaction is that the learning algorithms must be computationally fast enough to not impede the interaction progress or annoy the instructor. The agent needs to remain reactive and should be able to respond to an instruction, either through an action or dialog, within a few seconds. This could be combined with slower, more computationally intensive learning mechanisms that retrospectively analyze the training experiences during idle periods.

2.1.2 Interaction is Collaborative

The other main characteristic of interaction within the context of ITL is that the interaction is a collaborative process between the teacher and learner. Both must contribute to the interaction and support each other, or it will be less effective. From the agent's perspective, this means that it must include capabilities that give the instructor more options, more information, and greater flexibility. It should accommodate different levels of teaching expertise and preferences. A crucial capability for the teacher is the ability to build a mental model of the agent, its capabilities, and knowledge. This can be difficult for human teachers and students. It is much more difficult if the learner is an artificial agent with a different embodiment and background knowledge. To ease this burden, an ITL agent should be explainable, robust to errors, and correctable. We discuss these various desiderata below.

D4: *Mixed-Initiative*

One significant way to ease the instructor's burden is for the agent to support mixed-initiative interaction, where it initiates new interactions to acquire missing knowledge or additional examples. This allows the instructor to teach without having a perfect mental model of the agent and its knowledge, as they can rely on the agent to ask questions if they use an unknown concept or word. This also requires some form of meta-reasoning to identify missing, uncertain, or incomplete knowledge.

D5: *Flexible*

Another desired quality of the agent is that it supports multiple ways of teaching and interacting. Instead of requiring that the instructor has perfect knowledge of the single proper way to instruct the system, the agent should support different methods of instruction and be flexible to different teaching styles.

D6: *Multi-Modal*

One way of giving the instructor more teaching options is to support multi-modal interactions. Some modalities will be better suited for teaching a given task than others, so having multiple options is beneficial. Language is a rich and useful medium for teaching higher-level concepts and identifying what aspects of the environment are most important. Gestures can direct attention or emphasize what is most salient. Demonstrations can help teach lower level motor actions that would be difficult to phrase using language.

D7: *Explainable*

It is useful for the agent to expose its knowledge to the teacher to allow them to develop a better mental model of the agent and assess its current knowledge. This could be through specialized interfaces, or simply through mechanisms for answering instructor questions. These could include describing what it currently perceives, listing the actions it can do, or explaining what it thinks it should do next.

D8: *Robust*

Agents should be robust to errors made by the instructor during teaching or by its own interpretation of the instructions or demonstrations. Having some measure of suprisal or detecting if an instruction 'makes sense' in the current context would be useful. Being robust also involves simple error handling or rolling back some partial learning if it leads to an invalid state.

D9: *Correctable*

Along with being robust to errors, the agent should afford ways of correcting knowledge that it has learned. It should have mechanisms for allowing the instructor to interrupt its execution and provide alternative knowledge. Supporting explanation is crucial here to allow the teacher to identify the incorrect knowledge.

2.1.3 Interaction is Limited

One of the primary features of learning from interaction is that the data is high-quality, but sparse. Instructor time is limited, and most people will not have the patience to give numerous repetitive examples. However, the expectation is that the instructor has the explicit goal of teaching the agent well, and so those examples will be high-quality, tailored to the current learning experience, and accurate. These features distinguish learning through interaction from other learning problems that involve larger but less curated data sets. For an agent to learn from sparse but high-quality data, it must support the following:

D10: *Efficient*

Since the agent receives only a few examples, it must maximize the knowledge gained from each one. This suggests that the agent should utilize other capabilities, such as planning, reasoning, exploration, and experimentation, to augment its learning.

D11: *Aggressive Generalization*

Another consequence of needing to learn quickly and efficiently is that the agent should have the ability to aggressively generalize from the few examples to a large space of possible applications. It must also be able to transfer knowledge learned in one situation to another.

D12: *Extensive Background Knowledge*

It is unlikely that an agent will be able to learn quickly from few examples if it has no prior knowledge of the domain or its own capabilities. This is especially true for learning higher-level tasks that span extended timescales. Therefore, in order to support rapid learning and aggressive generalization, an agent needs to rely on extensive background knowledge such as primitive actions, object ontologies, or commonsense knowledge bases.

2.2 Situated Learning

Task learning is *situated* when it occurs in the same environment as task execution. Any interactions or demonstrations share the same context as the task being attempted by the learner. Not all interactive learning must be situated; the instructor could describe what to do in some future or hypothetical scenario. However, situated learning has a number of important benefits. Having a shared environment with the teacher helps tremendously with comprehending instructions and establishing common ground. Language is notoriously under constrained; statements can be ambiguous with multiple valid interpretations. Comprehending language within a specific context can help resolve these ambiguities. A classic example is the statement *'Move the cup to the left of the fork'*, where the prepositional phrase attachment is ambiguous (*to the left of the fork* could either identify which cup to move or specify the destination). However, this may not be ambiguous in the current context given the objects in the environment.

Furthermore, since they share an environment, both parties can use gestures and deictic references to establish common ground and direct attention. Even if the agent cannot resolve certain ambiguities, it can detect them and ask for clarification. Being situated also means the agent can immediately apply new task knowledge as it is learned. This simplifies the teaching as the instructor only needs to explain the task within the current context and not some hypothetical future scenario. The instructor can also immediately resolve any problems or failures that occur as they arise.

Learning through situated interaction emphasizes a number of the desiderata identified earlier. The learning must be immediately applicable (**D1**) and fast (**D3**). To fully take advantage of the shared environment, the agent should support mixed-initiative interaction (**D4**) to address problems that arise during execution.

2.3 Task Learning

A core focus of interactive task learning is developing agents that can learn a wide range of novel tasks that are not pre-trained (Gluck & Laird, 2019). This stands in contrast to much of the current work in machine learning, which is focused on developing agents with a very high level of skill on a singular task. Whether it is learning to master Go, recognize images, or control a drone, the task is usually fixed, and the agent maximizes its performance on that particular task. By contrast, interactive task learning is focused on breadth by developing agents that learn a wide range of different tasks. Addressing this broad scope imposes a number of agent requirements as follows.

D13: *Diverse Types of Knowledge*
Agents that can operate within their environments, engage in interaction with an instructor, and learn a wide range of tasks require an extensive set of capabilities integrated together, including perception, motor control, natural language comprehension, spatial reasoning, planning, and action modeling. All of these involve different types of knowledge and will likely involve heterogeneous representations. Ideally, these should also be learnable through interaction.

D14: *Compositional Knowledge*
Not only should an agent be able to represent different types of knowledge, but it should be able to compose them to learn more complex concepts and structures. For example, it should be able to compose previously learned tasks within an even higher-level task.

D15: *Diverse Task Formulations*
As discussed in the introduction, tasks vary wildly in their goals, objects, concepts, actions, and executional strategies. We predicate that a singularly focused learning algorithm will not be sufficient for the breadth of learning that is desired. Instead, we must develop techniques for integrating different task formulations (the representations, learning mechanisms, and execution strategies used) within a single agent.

D16: *High Performance*
In the end, the agent should not only learn the structure of the task, but be able to perform the task accurately and reliably. This will likely require refining the task knowledge over time through experience, practice, and expert advice.

2.4 Discussion

The previous sections define a large number of desiderata for ITL agents, and there are likely additional ones not listed above. Addressing them all is beyond the scope of this work. Instead, we will highlight those that are the most important for our particular research focus: learning a wide range of high-level tasks from one-shot instruction. Therefore the most important desiderata are that the learning is efficient (**D10**) and uses aggressive generalization (**D11**) to perform the task correctly in the future (**D16**). Our goal is for the agent to demonstrate a high degree of transfer to many different task variations from a single teaching example. This is a main reason we use symbolic representations and an explanation-based learning approach. These afford the significant generalization capabilities that Rosie demonstrates. This rapid learning and transfer also requires extensive background knowledge (**D12**) and compositional representations (**D14**). A major contribution of this dissertation is demonstrating how an agent can display this impressive generalization across a wide range of tasks and through diverse task formulations (**D15**). Rosie uses many different types of knowledge (**D13**), and has been used to learn many of them in previous work, but for this research we will assume that most of it is innate.

Learning from situated interactive instruction also constrains the learning approach by requiring that it be immediate (**D1**), incremental (**D2**), fast (**D3**), and mixed-initiative (**D4**). However, we are not focused on the desiderata that improve the overall quality of the interaction. We assume the instructor is an expect and knows the proper way to teach the agent and the correct language to use. While the instructions will be reasonable natural language sentences, there might only be one or two correct (and sometimes esoteric) ways of describing a given concept. Thus the instruction is not multi-modal (**D6**) or very flexible (**D5**). We will not investigate how to recover from instructional or learning errors (**D8**, **D9**) or how to provide explainability mechanisms (**D7**), although using a cognitive architecture and declarative representations should support these capabilities. In the concluding chapter, we discuss future work for addressing some of these desiderata.

CHAPTER 3

Related Work

Research into developing agents that learn from humans is a very wide field that encompasses very different learning approaches, interaction modalities and strategies, and types of knowledge being targeted. In this chapter, we provide a broad overview of the most closely related work. We are mostly concerned with learning task knowledge, so we mostly restrict this discussion to work involving learning actions or tasks from interaction. We first cover methods of learning to ground language to internal action representations, then move on to learning from demonstration. In the final section, we cover methods of learning tasks through instruction, and identify the most closely related work that will be used throughout this dissertation for comparison.

3.1 Learning to Ground Actions

One area of related work is learning to ground language to actions. These approaches are given training pairs of natural language commands and action sequences and learn a model that directly maps language to internal action representations. Some approaches learn a parser to do the translation. The approach of (Branavan et al., 2010) learns to map natural language instructions to actions in a Windows desktop environment. Chen & Mooney (2011) use examples of instructions paired with an action sequence to learn a semantic parser that produces navigation plans. Artzi & Zettlemoyer (2013) demonstrated a method for learning a combinatory categorical grammar (CCG) from situated examples that translates natural language instructions into an executable expression. Similarly, Matuszek et al. (2013) also learned a CCG to map commands to a robot control language. More recently, Thomason et al. (2020) used situated dialog to improve the performance of a semantic parser over time and can generate queries through active learning to gain additional examples.

Other approaches learn a model that performs the mapping. Tellex et al. (2011) train a probabilistic graphical model that is instantiated by a particular command and inference over the model produces an executable plan. It includes the hierarchical and compositional structure required to perform tasks with a robotic forklift. Misra et al. (2017) train a single neural model that maps a visual observation and instruction text to actions in a 2D environment using reinforcement learning to guide exploration. Misra et al. (2016) develop a dataset of household manipulation tasks that pairs natural language with task logs. They develop and train a model that grounds instructions to robotic instructions given the environment context, task constraints, and is robust to noisy or incomplete instructions. Work by Arumugam et al. (2017) uses deep neural networks to map natural language commands to the proper level in a planning hierarchy. The major contribution was showing that the model could handle commands at different levels of granularity.

These approaches take a more data-drive approach to learning and usually learn offline from large datasets. So even though they involve natural language, they are not learning from situated instruction. These tend to require a lot of data and examples to train. For example, Misra et al. (2017) train a model that learns to move individual blocks within a simple 2D blocks-world environment, and still requires over 15,000 examples. As discussed in the previous chapter, the challenge of learning from a few situated examples is a very different learning problem and will require different techniques.

3.2 Learning from Demonstration

One major approach to ITL is learning from demonstration (LfD: Ravichandar et al. 2020; Argall et al. 2009), also referred to as programming by demonstration (PbD: Billard et al. 2008) or imitation learning. A demonstration can take many forms, including kinesthetic teaching (Calinon et al., 2007; Ahmadzadeh et al., 2016; Elliott et al., 2017; Chu et al., 2016; Akgun et al., 2012; Phillips et al., 2016), teleoperation (Havoutis & Calinon, 2019; Spranger et al., 2018; Zhang et al., 2018; Rosen et al., 2018; Kukliński et al., 2014), or observations (Dillmann, 2004; Vogt et al., 2017; Liu et al., 2018; Mollard et al., 2015). Learning from observations, or imitation learning, requires addressing the correspondence problem (Nehaniv & Dautenhahn, 2002), or how to map the observed actions of another to one's own embodiment. This is not as much of an issue if learning from kinesthetic training or teleoperation, but they are typically more difficult for a teacher than performing a demonstration. A central challenge to learning from demonstrations is the *data description problem* (Cypher & Halbert, 1993), where the observation of an action does not include information about *why* the action was performed. This is why demonstrations are often paired with natural language instructions for the purposes of feedback, clarification, or explanation (Forbes et al., 2015; Hayes & Scassellati, 2014; Nicolescu & Mataric, 2003; Wang et al., 2018).

Learning from demonstration is a related problem, but generally attempts to tackled a different learning problem than this dissertation. Learning from demonstration is useful for learning fine-grained manipulation actions such as opening a drawer. Often they result in learning continuous policies or reward functions. These types of actions are not currently learnable by our agent, and would be difficult to describe only using language. However, it is a hard problem to extract the relevant features from high-dimensional and noisy demonstrations, and as a result these approaches tend not to generalize as well from a single example.

3.3 Learning from Situated Interactive Instruction

Situated interactive instruction (SII) has been used to learn a number of different concepts beyond tasks, including object references and perceptual semantics (Matuszek et al., 2012; Thomason et al., 2016; Amiri et al., 2019), block structures (Perera et al., 2018), spatial relations (Spranger & Steels, 2015; Gong & Zhang, 2018; Chao et al., 2011), navigation routes (Lauria et al., 2002), and planning models (Cantrell et al., 2012). Ideally, a general interactive task learning agent should have the capability to learn all of these different types of knowledge. Some earlier work showed Rosie learning perceptual properties and spatial relations (Mohan et al., 2012), but for this dissertation we assume that the agent is able to fully ground any object or spatial references.

3.3.1 Early Work

Although the specific area of interactive task learning has been recently identified and defined as a general research problem, work on learning task knowledge from instruction has been undertaken since the early years of AI. In 1972, Winograd developed SHRDLU, a program that could understand natural language commands to manipulate a simulated blocks-world environment and learn simple concepts such as specific block configurations (Winograd, 1972). Other early work includes UNDERSTAND (Simon & Hayes, 1976), which could turn natural language instructions describing variations of the Tower of Hanoi problem into internal problem-space representations. There was also the Instructable Production System (Rychener, 1983), a research project with the goal of learning productions from instruction that was unsuccessful but helped develop ideas used in the creation of Soar. None of these learned complete task representations from scratch.

Work by Crangle & Suppes (1994) developed foundational ideas about the notion of an instructable robot that could learn new tasks from instruction. Their work was more focused on how to understand natural language commands. It demonstrated an impressive ability to parse a wide range of instructions to a simulated mobile robot. These included spatial relations, conditions, temporal phrases, and termination clauses. However, this was a fixed parser from instructions to execution plans and did not involve learning new tasks. They did show the ability to translate English instructions into a procedure for adding numbers, though this was closer to interactive programming where the commands were interpreted into a LISP procedure. Any interaction was not mixed-initiative, their agent did not ask questions of the teacher.

One of the earliest examples of learning hierarchical tasks from instruction was Instructo-Soar (Huffman & Laird, 1995), which could learn a full task problem-space representation in a simulated blocks-world environment. This was a precursor to Rosie, and provided some key ideas. One major one was the ability to learn all the different types of task knowledge through instruction, although the language was very restricted and this system was only applied to a simulated blocks world domain where there was complete, noise-free perception.

3.3.2 Comparison Work

In order to compare and evaluate this thesis within the broader field, we identify the closest related work and assess their task learning capabilities along the three dimensions of complexity. This analysis will be performed within each of the three related chapters (7, 8, and 9). In this section, we identify the set of related approaches that will be studied and give a brief description of each. The major deciding factor for inclusion was whether a primary research goal was learning completely novel tasks from situated interactive instruction and demonstrations with an integrated agent that demonstrated end-to-end behavior.

Several approaches learn tasks in a tabletop environment with a robot that can manipulate objects in front of it, such as the PR2[1] or the Baxter[2] platforms. First, we will compare against the prior Rosie work (Mohan & Laird, 2014). It learns hierarchical blocks-world tasks such as moving or 'storing' blocks with a robotic tabletop arm. These tasks were formulated as achieving a singular goal and learning a state-based policy over innate and learned subtasks. We present a number of significant improvements over this work – implementing the approach within a mobile robot, adding many new types of innate tasks (including mental and communicative actions), extending the task representation to include procedural tasks, and adding support for temporal, conditional, and repetitious clauses.

Frasca et al. (2018) demonstrate one-shot task learning from instruction within the DIRAC architecture (Scheutz et al., 2019). Their approach learns manipulation task procedures such as passing objects between people using a PR2 robot. A key distinguishing

[1] https://www.willowgarage.com

[2] https://www.rethinkrobotics.com

feature is their ability to identify different roles in the task and transfer task execution knowledge from other actors in the task example. Although the DIRAC architecture is able to represent more general task structures, the given work only demonstrated learning fixed linear procedures. It could learn hierarchical tasks, though was only demonstrated on simple manipulation tasks, so the true generality of the approach is still unproven.

Suddrey et al. (2016) teach tabletop pick and place tasks with a Baxter robot, such as clearing a table or storing an object. They learn an hierarchical task network (HTN) solely from natural language instructions and not only learn parameterized arguments, but preconditions and postconditions. This allows their agent to use forward planning to improve flexibility. However, their approach cannot learn true goal-based tasks, and was limited to a fully-observable environment where the only actions were opening, closing, and moving a gripper.

She & Chai (2017) demonstrate an approach for learning verb semantics represented as a goal hypothesis space refined through observing multiple demonstrations involving both instructions and actions. It compares the initial and final states to generate possible goal predicates, and maintains a version space with multiple predicate subsets. This improves on earlier work (She et al., 2014) that only learns a single goal for blocks world tasks, but it does not attempt to learn hierarchical tasks. Having this multiple hypothesis space allows for better dealing with uncertainty and supports more flexible planning. However, the problem of identifying what state changes should constitute the goal is largely side-stepped by our approach of having the instructor directly describe the goal. The downside is that it requires more precise instruction and is less flexible if there are multiple variations of the goal.

Work by Mohseni-Kabir et al. (2019) combines learning hierarchical task networks from Mohseni-Kabir et al. (2015) with the ability to learn new action primitives. Their approach, simultaneous learning of hierarchy and primitives (SLHAP), was demonstrated with a task of rotating tires on a car using a simulated PR2 robot. The instruction helps the robot identify trajectory boundaries when learning primitives. The most relevant aspect is the ability to learn a multi-level HTN through a single interaction. The language is somewhat limited, though the agent is proactive with suggesting possible task groupings. The HTN itself is very rigid and supports almost no generalization or flexibility apart from the specific object arguments.

Other related work involved learning navigation tasks with a mobile robotic platform. For example, Meriçli et al. (2013) teach a mobile robotic agent (CoBot, Rosenthal et al. 2010) tasks such as getting coffee and following landmarks. It learns a graph structure called an Instruction Graph (IG) that represents procedural control flow between primitive actions. This can include do-until loops and conditional statements, and was able to be modified or corrected post-training. However, the learned tasks are not very flexible, the instructor must explicitly teach every contingency. It also does not do much generalization. For example, it learns a task of getting coffee, but could not apply that knowledge to getting a different drink or even starting at a different place in the world. The task learning is also not compositional, it cannot use one task within another.

A similar agent from Gemignani et al. (2015) learns navigation tasks such as delivering an object or checking a room for a person. It uses a method that first maps natural language instructions to an intermediate task description language, then transforms that into a formal task execution representation (Petri Net Plans or PNP; Ziparo et al. 2011). This approach was shown to learn parameterized procedural graph structures that included looping and branching structures for a mobile service robot. It is more general than the approach by Meriçli et al. (2013), as specific locations and objects can be parameterized. However, it also does not learn hierarchical tasks, and the graph structure is not flexible.

Learning from SII has also been used to teach software based tasks. These are still considered to be situated, since the agent has direct access to the software state and the instructions and demonstrations are directly related to to state. PLOW (Allen et al., 2007) learned tasks that involved extracting information from websites, such as finding businesses within some distance of an address. It mostly learned linear procedures, although it could repeat actions for each instance in a collection.

Work by Tom Mitchell and others applied ITL to software and personal assistant tasks. For example, LIA (Labutov et al., 2018) is an intelligent personal assistant that can be taught world knowledge and conditional procedures through natural language. We will focus our analysis SUGILITE (Li et al., 2020), a smartphone agent that can learn tasks by combining instructions and GUI demonstrations within apps. It can learn compositional concepts, such as how to find the current temperature using the weather app, but it is unclear whether the actual procedures can be combined hierarchically.

CHAPTER 4

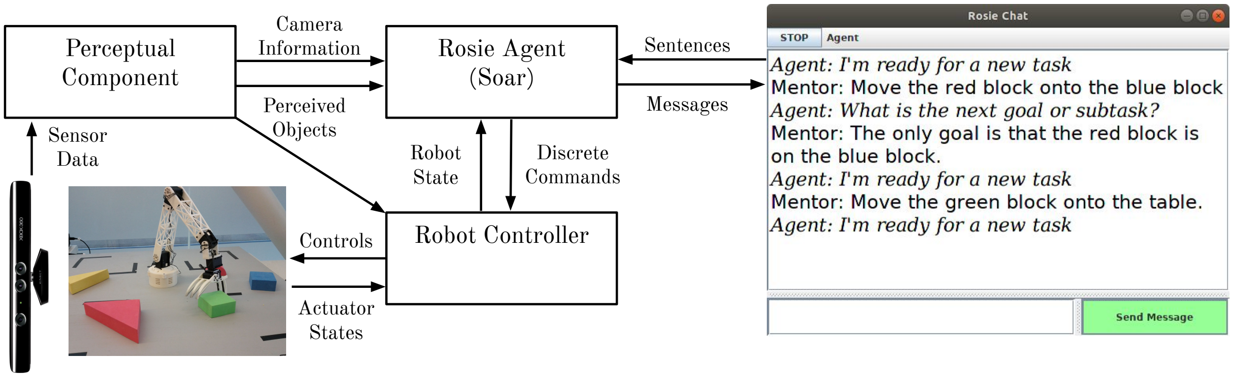
Agent Overview

The task learning extensions presented in this dissertation are implemented in the context of Rosie – a general cognitive agent that learns from Situated Interactive Instruction. Rosie has been the product of the combined effort of a number of researchers at the University of Michigan exploring the learning of various types of knowledge from instruction. This research has included learning perceptual features and spatial relationships (Mohan et al., 2012), hierarchical state-based concepts (Kirk & Laird, 2019), games and puzzles (Kirk & Laird, 2016), goal-based tasks (Mohan & Laird, 2014), and procedural tasks (Mininger & Laird, 2018). It has also been used as an application for researching grounded language comprehension (Lindes et al., 2017) and human-robot interaction (Ramaraj et al., 2019). Rosie has been deployed in a number of different environments and embodiments, including a tabletop arm, a mobile robot, a fetch robot, a toy forklift robot, and several simulated domains.

In this chapter, we give an overview of the Rosie agent architecture, including how it is integrated with its environment and organized within the Soar cognitive architecture (Section 4.1). We then describe the way processing is organized in Rosie and the computational model it uses to engage in interaction (Section 4.2). We conclude with an analysis of how the knowledge and processing organization supports the agent desiderata (Section 4.3).

4.1 Agent Architecture

The Rosie agent has been deployed across a number of different environments and embodiments, but the overall agent organization (and core agent software) remains the same (Figure 4.1). While underlying perceptual and control systems differ between environments, the ways they interface with the agent do not. The perceptual component produces an egocentric, object-oriented representation from its sensory data that it provides as input to the agent, along with information about its sensors and actuators. The agent uses this information to



Agent: I'm ready for a new task
Mentor: Move the red block onto the blue block
Agent: What is the next goal or subtask?
Mentor: The only goal is that the red block is on the blue block.
Agent: I'm ready for a new task
Mentor: Move the green block onto the table.
Agent: I'm ready for a new task

Figure 4.1: The Rosie agent in relation to the external environment.

construct a representation of the world. This process, which is a contribution of this thesis, is described in Chapter 5 and Mininger & Laird (2019).

Rosie performs actions in the world by sending discrete commands to the robot controller. These commands include arm manipulations such as `pick-up(`o_4`)` or `place(`o_7`, `*on*`, `o_9`)`, navigation commands such as `drive-to(`x`, `y`)` or `turn(`*angle*`)`, and object interactions such as `open(`o_3`)` or `turn-off(`o_6`)`. The controller then reports back the status of the command, such as *executing, succeeded, failed.*

The primary way that the instructor interacts with the agent is through a chat interface, shown in Figure 4.1. When the instructor enters a sentence, it is sent to the agent, which performs language comprehension and parsing to produce a grounded message structure. When the agent produces an outgoing message structure, it is translated into natural language through predefined templates. In certain simulated domains, the instructor can also select an object by clicking on it in the simulator interface.

4.1.1 The Soar Cognitive Architecture

The Rosie agent is implemented within the Soar cognitive architecture (Laird, 2012), shown in Figure 4.2. Soar provides various memories and learning mechanisms that support cognitive processes. It has a symbolic *working memory* that contains representations of the current state of the world, the agent's goals, and other information required for the the current computation. This knowledge is represented as a directed graph, where each edge (starting node, edge label, and ending node) is the fundamental representational unit known as a *working memory element* (WME). Working memory also contains specific regions (or *buffers*) that are used to interface with other memories and input/output.

Soar also has a short-term memory called the *spatial visual system* (SVS), that stores the metric representations of entities in the world produced by perceptual processing. SVS

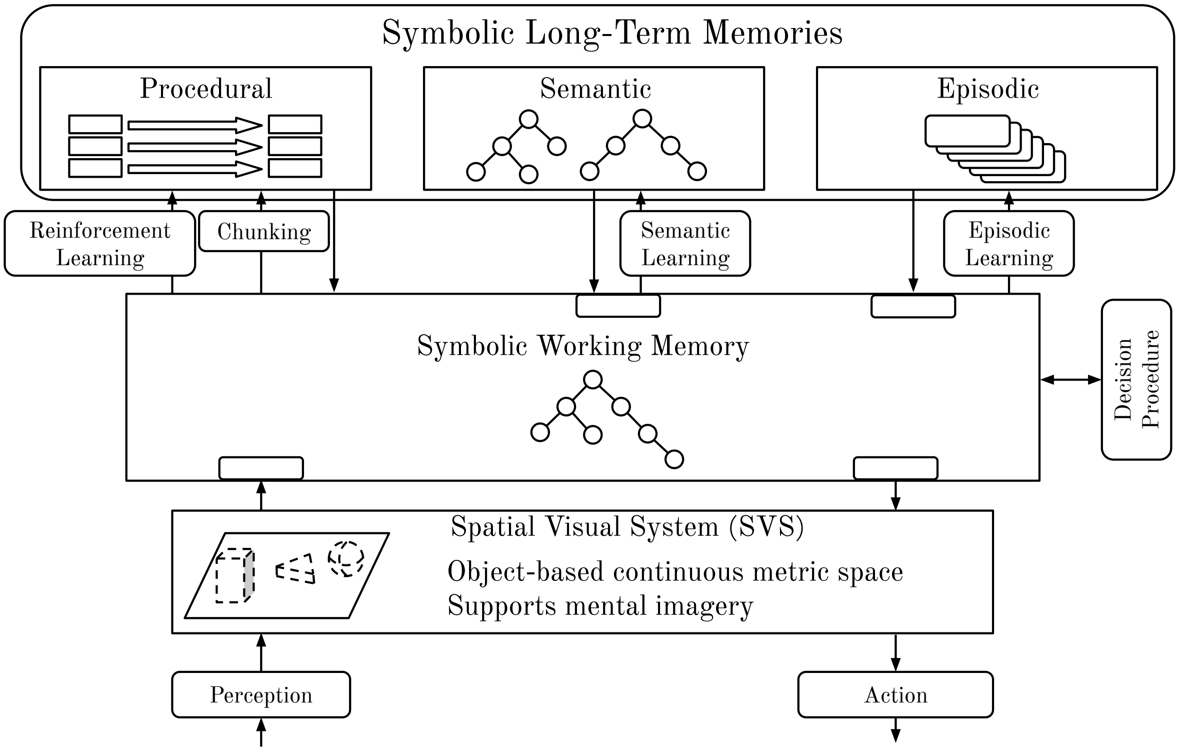


Figure 4.2: The Soar Cognitive Architecture (adapted from Laird 2012)

uses a scene-graph representation of objects and their geometric properties and supports general-purpose queries that the agent can use to extract relational information about the scene. Together, these support complex reasoning involving continuous representations that would be prohibitively difficult using symbolic procedural rules alone.

In addition to these short-term memories, Soar has three long-term memories. *Semantic memory* is a declarative long-term memory that contains general knowledge about the world, represented as a directed graph. Knowledge in semantic memory can be added as a priori knowledge during initialization or deliberately stored by the agent during execution. The agent can then retrieve this knowledge by constructing a *cue* in a specific buffer in working memory. Soar matches this cue against all of semantic memory and retrieves the best match to working memory, weighted by an activation value determined by recency and frequency of access.

Episodic memory is also a declarative long-term memory that stores a history of the agent's past experiences. Soar automatically stores an ordered sequence of episodes (working memory states) that the agent can query to retrieve the context of an event. Similar to semantic memory, the agent can initiate a cued retrieval where the architecture returns the best partial match, biased by recency. This result is added to working memory as a complete reconstruction of the top state at that time. The agent can also extract relative temporal information, such as the episodes preceding or following some event.

The third long-term memory is a rule-based *procedural memory* that contains the agent's knowledge of skills, procedures, and reasoning capabilities. This can include how to access its long-term memories, learn new knowledge, and take actions in the world. Procedural rules are if-then statements that match against working memory and then fire to apply their results. Rules that match simultaneously fire in parallel. Decision making is done at a level above rules, with special structures called *operators*. An operator represents a potential internal or external action that the agent can take. Soar has a decision cycle built around operators which goes through the following phases:

- **Input Phase:** The input buffer is updated with new perceptual information.
- **Elaboration Phase:** Rules fire to elaborate the state, propose operators, and evaluate proposed operators.
- **Decision Phase:** Soar selects an operator based on elaborated preference knowledge.
- **Application Phase:** Rules apply the operator and modify working memory or construct output/retrieval commands.
- **Output Phase:** Output, SVS, and long-term memory commands are processed.

During this decision cycle, if the agent is missing sufficient knowledge to select or apply an operator, Soar will reach an *impasse* and create a new substate intended to overcome the impasse. For example, if the agent is lacking knowledge of which operator to select next, it will enter an operator tie impasse and the agent can perform a search or further operator evaluation to decide which is the best one to choose. Alternatively, if the agent lacks knowledge to immediately apply the operator, it will enter an substate where it can decompose the operator into smaller, more manageable steps. In certain circumstances, the *chunking* learning mechanism compiles the reasoning done in the substate into a new rule. Chunking is based on explanation-based generalization (Mitchell et al., 1986). It analyzes the causal connections as defined by rule firings to determine the working memory structures of the parent state that was used in the substate to resolve the impasse and create the result. Those structures become the basis for the conditions of a new rule that can match in future situations to immediately create the result instead of requiring the substate processing.

4.1.2 Knowledge Organization

In this section, we describe how Rosie's knowledge is distributed across the different memories in Soar (Figure 4.3). Soar's working memory holds the current representation of the world,

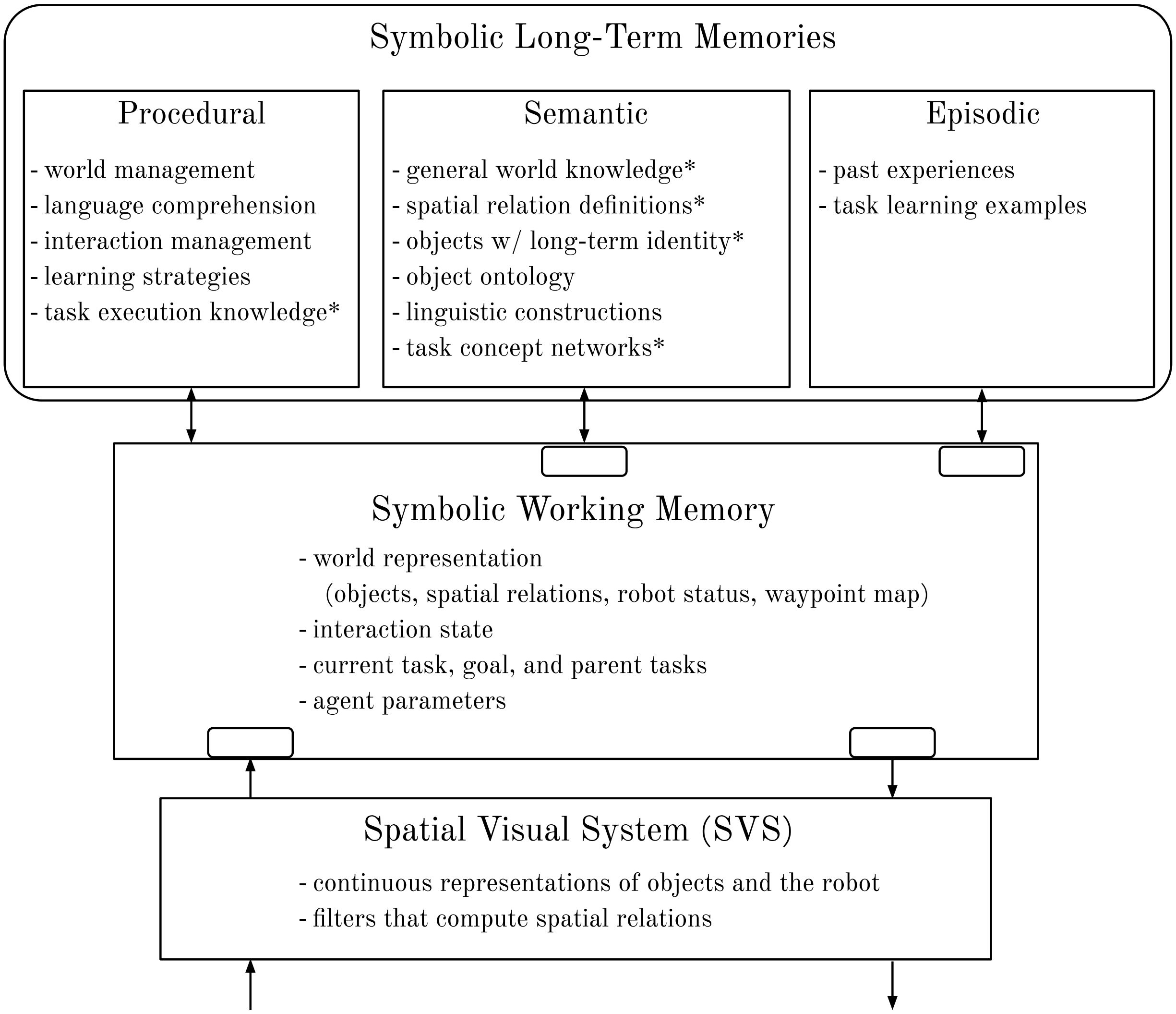

Symbolic Long-Term Memories
Procedural
- world management
- language comprehension
- interaction management
- learning strategies
- task execution knowledge*
Semantic
- general world knowledge*
- spatial relation definitions*
- objects w/ long-term identity*
- object ontology
- linguistic constructions
- task concept networks*
Episodic
- past experiences
- task learning examples
Symbolic Working Memory
- world representation
(objects, spatial relations, robot status, waypoint map)
- interaction state
- current task, goal, and parent tasks
- agent parameters
Spatial Visual System (SVS)
- continuous representations of objects and the robot
- filters that compute spatial relations

Figure 4.3: How Rosie's knowledge is organized across the Soar Architecture. An asterisk indicates knowledge that Rosie can learn.

including objects, spatial relations between them, and the robot's current state. The agent has access to metric information about these objects and the robot embodiment in SVS, and it creates filters in SVS to compute spatial relationships between objects. Working memory also contains a representation of the current state of the interaction, including the current dialog context and purpose. During task execution, working memory contains a representation of the current task, its goal, and all parent tasks in the task decomposition.

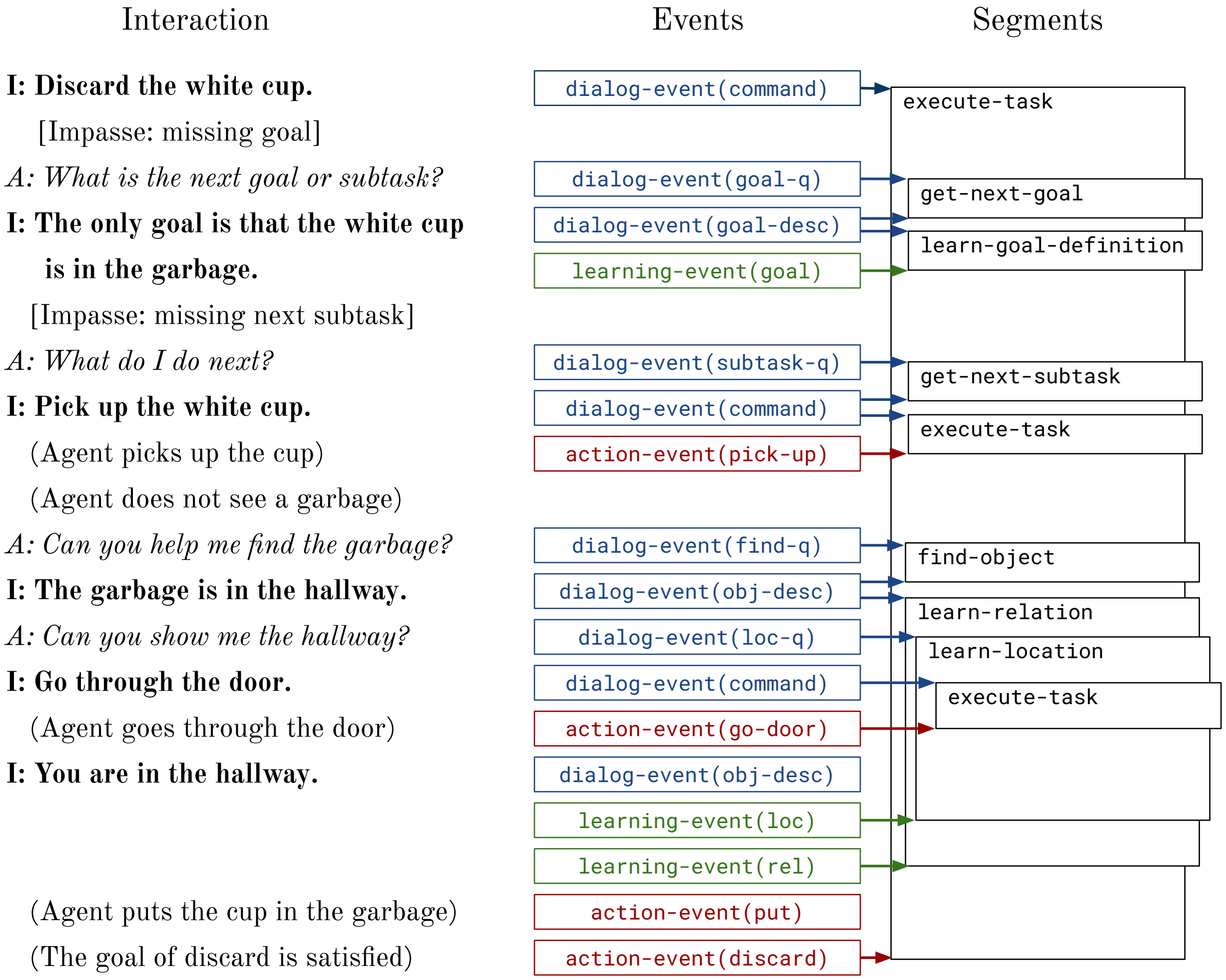
Procedural memory contains rules that implement the various types of processing Rosie performs during interactive task learning, including how to construct and manage the world representation, perform language comprehension, manage the current interaction state, and acquire missing task knowledge. Most of the task knowledge required to execute a particular task (either learned or pre-existing) is compiled into procedural rules, including task preconditions, decomposition, policy, and postconditions.

Semantic memory contains general knowledge about the world, such as knowledge of perceptual features (red is a color), the object category ontology (table is a type of furniture), and definitions of spatial relations (above means aligned in x and y axes and greater in the z axis). It stores information about certain objects with long-term identity, such as rooms and people, and constructions used during language comprehension. Semantic memory also contains structured knowledge about each task in a representation called a Task Concept Network (TCN). This network includes knowledge about a task's structure, goal, and subtasks. During execution, this knowledge is deliberately interpreted to produce behavior, which is then compiled into procedural rules through chunking, as described in Section 6.3.

Finally, episodic memory automatically captures the history of the agent's experiences, encoded as sequential episodes. Rosie primarily uses episodic memory to replay a task after completion, which leads to learning additional rules. This is referred to as *retrospective learning* and is described more in Section 6.4.3. For example, it can reconstruct the initial state when the task began and replay the subtasks to learn the task's preconditions. Rosie also uses episodic memory to reconstruct parts of the world from its past experience when it enters a room (Section 5.3.1) and one of its innate mental tasks (recall) can access episodic knowledge during execution.

4.2 Agent Processing

Processing in Rosie is organized around the interaction between the instructor and the agent. The agent has an explicit representation of the current interaction state and is always trying to carry the interaction forward. Thus, Rosie's behavior is driven by the instructor – it will not perform actions unless they will help accomplish the purpose of a given instruction.



Interaction

I: Discard the white cup.

[Impasse: missing goal]

A: What is the next goal or subtask?

I: The only goal is that the white cup is in the garbage.

[Impasse: missing next subtask]

A: What do I do next?

I: Pick up the white cup.

(Agent picks up the cup)

(Agent does not see a garbage)

A: Can you help me find the garbage?

I: The garbage is in the hallway.

A: Can you show me the hallway?

I: Go through the door.

(Agent goes through the door)

I: You are in the hallway.

(Agent puts the cup in the garbage)

(The goal of discard is satisfied)

Figure 4.4: An example of teaching discard between the Instructor (I) and Agent (A), along with the events that occur and how those events change the interaction stack.

However, the interaction is still *mixed-initiative*, since both the instructor and Rosie can change the state of the discourse by initiating new interactions. Consider the task *"Discard the white cup"* shown in Figure 4.4. As Rosie attempts to execute the task, it reaches impasses due to missing knowledge and initiates new interactions to obtain that knowledge. This greatly increases the flexibility of the dialog – the instructor does not have to have a complete model of the agent's internal knowledge and state when giving a command. Instead, the agent initiates interactions when it requires more information.

In order to support this flexible, mixed-initiative interaction, Rosie requires a representation of the current interaction state and knowledge of how to change this state as new sentences are communicated by itself or the instructor. The approach that Rosie uses was developed by Mohan et al. (2013) and is an extension to Collaborative Discourse Theory

(CDT; Grosz & Sidner 1986, Rich & Sidner 1998) that supports interactive learning. The state of discourse is represented by a stack of *segments*, where a segment is a contiguous sequence of *events* that serves a specific *purpose.* An event represents some change; in Rosie we consider three types of events: a change in the environment due to the agent's behavior (*action-event*), a change in the state of the interaction (*dialog-event*), or a change in the agent's knowledge (*learning-event*). The purpose of a segment is paired with a *satisfaction* – a set of events that indicate the purpose has been achieved. For example, when the instructor gives the command *"Discard the white cup,"* a new segment is created with the purpose of executing the discard task and an `action-event(discard)` satisfaction (Figure 4.4). Or when the agent asks for the goal, a new segment is created with the purpose of acquiring the goal of discard and a `learning-event(goal)` satisfaction. If an event occurs that corresponds to a segment's satisfaction, that segment is removed. Segments are hierarchical, so the segment for learning the goal is pushed onto the stack above the one for discard, and it is also contributing to its parent's purpose.

In the following sections, we describe the processing steps when a new sentence is given by the instructor, and how that interacts with the discourse stack. Throughout we will use the example of the task *'Discard the white cup.'* shown in Figure 4.4.

4.2.1 Language Comprehension

When the instructor gives a new instruction, the agent incrementally parses it as it is placed onto the input link. The language comprehension process combines both syntactic and semantic knowledge to generate a grounded message structure that the rest of the agent can use. This approach (Lucia; Lindes et al. 2017), was developed by Peter Lindes, with contributions from John Laird, and is based on Embodied Construction Grammar (ECG; Bergen & Chang 2013) from cognitive linguistics research. The ECG grammar theory defines two items: *constructions* and *schemas.* A construction pairs form and meaning, where the form can be a single word or an entire type of clause or sentence. Constructions build upon each other recursively. Each connects the syntactic form to its meaning, encoded as schemas. A schema is an organized collection of *roles*, or slots, that hold information. Thus when a construction matches, it defines which schemas to invoke and how to assign values to the various roles. Lucia is implemented in Soar procedural rules and works incrementally to build up a nested structure of constructions and their schemas. Along the way, certain items are grounded to the world. Finally, once a construction matches the entire sentence, the agent performs an interpretation process where it takes the information from the schemas and generates a message structure that the rest of the agent can use and act upon.

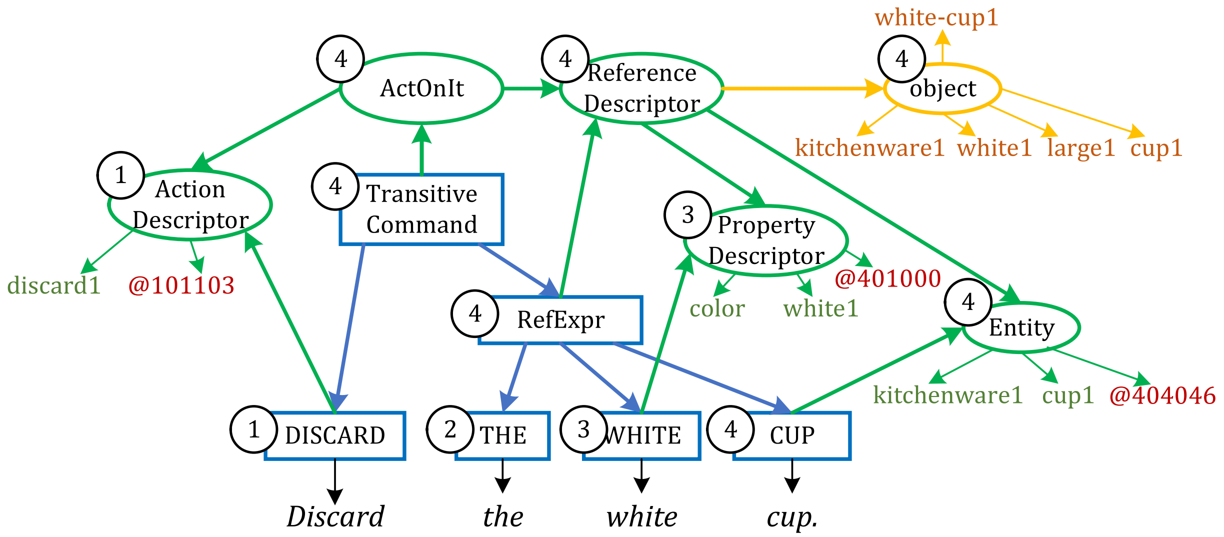


Figure 4.5: Result of language comprehension for the sentence *'Discard the white cup.'* (Adapted from Lindes et al. 2017)

```
construction TransitiveCommand
   subcase of Imperative
   constructional
      constituents
         verb: ActionVerb
         object: RefExpr
   meaning: ActOnIt
      constraints
         self.m.action <---> verb.m
         self.m.object <---> object.m
```

```
schema Action
   roles
      action
      location

schema ActOnIt
   subcase of Action
   roles
      object
```

Figure 4.6: Example ECG construction and schemas used during *discard* to match an action command (Adapted from Lindes et al. 2017)

```
(<msg> ^message-type command ^action <act> ^arg1 <obj7>)
  (<act> ^handle discard1 ^item-type action)
  (<obj7> ^handle obj7 ^item-type object ^predicates <o7preds>)
     (<o7preds> ^category cup1 ^color green1 ^is-visible1 visible1 ...)
```

Figure 4.7: The end result of language comprehension – an internal message structure. Note that `<obj7>` is a grounded object in the world representation.

We demonstrate this process using our example of *"Discard the white cup."* Through the comprehension process, it generates the hierarchical structure seen in Figure 4.5. Along the way, it encounters the referring expression *'the white cup'* that it tries to ground to the objects in the world representation. It does find such an object, which is placed into the proper role in the `ActOnIt` schema. Once it has comprehended the entire sentence, the agent performs the interpretation step to generate a message structure with type *command* (Figure 4.7).

For our experiments and demonstrations, we assume that the agent possesses the knowledge required to correctly and unambiguously parse all of the necessary words and phrases. So when learning new tasks, it will know the part of speech information about the verbs used, but no semantic information. Other work has shown how new parsing knowledge can be acquired (Lindes et al., 2017).

4.2.2 Interpretation

Once the comprehension phase has produced a message structure (Figure 4.7), Rosie interprets this message within the current dialog context. The agent has procedural knowledge of how to handle each message type and how they modify the interaction stack. For example, when the agent receives a message with type `command`, it will push a new segment onto the stack with the purpose of `execute-task` (Figure 4.4). Or when the agent receives a message with type `goal-description` in the context of a segment with purpose `get-next-goal`, it will terminate that segment and create a new one with the purpose of `learn-goal-definition`.

4.2.3 Purpose-Driven Behavior

The top segment on the interaction stack represents the current focus of the agent and its current interaction goal. Processing in Rosie is centered around trying to satisfy the current purpose. It is programmed with procedural knowledge of what to do for various purposes (e.g. executing a task, answering a question, or learning a new concept). While doing so, certain events may occur that will change the interaction state. If an event satisfies the purpose of the current segment, it is popped from the stack. Or when the agent asks a question a *dialog event* will cause a new segment to be pushed, such as when the agent asks for the goal via a `goal-query` (Figure 4.4).

4.2.4 Impasse-Driven Learning

Often when the agent is trying to satisfy the purpose of the current dialog segment, it will reach a point where it is missing some knowledge required to continue. In Soar, this manifests as an impasse, and it provides an opportunity for learning. Instead of failing or quitting, the agent can reason about why the impasse occurred and attempt to resolve it. Sometimes the agent can resolve the impasse on its own. For example, at line 6 in the discard example, the agent knows the goal of discard but does not know what action to perform next. It hits an impasse, where the agent does look-ahead search to figure out it needs to find the garbage next. Rosie will also learn a new rule through chunking that contains this knowledge. However, the agent may not always be able to acquire the missing knowledge on its own. In those situations, it initiates a new interaction to obtain that knowledge from the instructor and pushes a new segment onto the dialog stack with the purpose of learning that knowledge. When that learning event occurs, the segment is popped and the agent returns to where it left off.

This tight integration between execution and learning works well for Situated Interactive Instruction, as the learner and instructor engage in a mixed-initiative dialog as the agent makes progress on learning the task. It also gives greater flexibility to the instructor, as they can choose to either start with simple concepts and work up to more complex ones, or start with a high-level task and have the agent ask questions as it encounters missing knowledge.

4.3 Discussion

We have described the overall Rosie agent architecture, how it is represented within the Soar cognitive architecture, and how the processing is structured. The structure of Rosie directly supports several of the desiderata from Chapter 2. First, using symbolic, declarative memories to represented task knowledge allows for the learning to be incremental (**D2**), since it is easy to modify and extend declarative structures in Soar. These memories also support efficient learning (**D10**), since knowledge from a single interaction can be directly converted into internal structures, and they are innately compositional (**D14**). Through task execution, this declarative knowledge is compiled into procedural knowledge through chunking, which is both fast (**D3**) and supports extensive generalization (**D11**).

The processing in Rosie is first and foremost organized around interaction. When a new instruction occurs, Rosie immediately interprets the instruction and works to learn and apply any knowledge contained within (**D1**). Soar's *impasse-driven learning* supports mixed-initiative instruction (**D4**), as the agent can detect missing task knowledge and initiate new interactions to obtain it (**D4**). This also serves to make the instruction more flexible (**D5**), as the teacher can choose whether to teach concepts in a top-down or bottom-up manner.

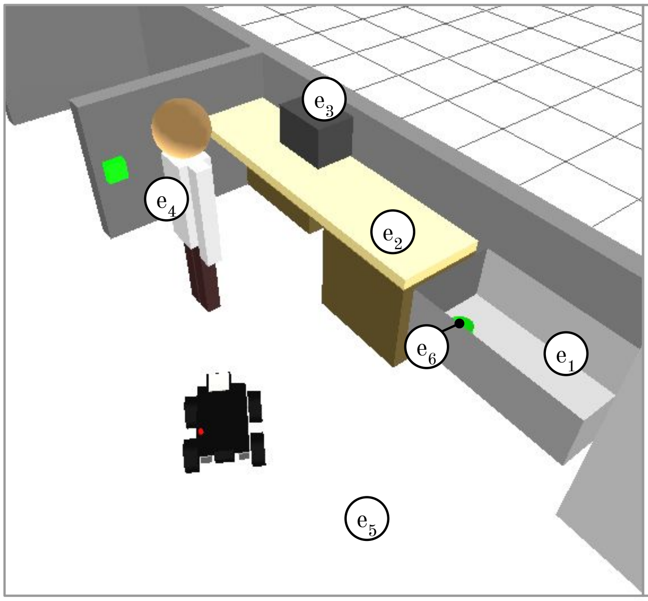
Most of the structure and processing of Rosie was developed in previous work (Mohan et al., 2012, 2013). The specific contributions of this thesis include the method for constructing and maintaining the world representation (Chapter 5), extensions to task representations, and procedural knowledge for learning and executing tasks (Chapter 6).

CHAPTER 5

Constructing the World Representation

If a robotic agent is to support interactive task learning while situated in an environment, it must construct and maintain an accurate representation of the world. Our approach to ITL requires an object-based representation with object positions, attributes (color, shape, etc.), relations (above, to-the-right, etc.), and other semantic information. Producing this representation is challenging, as it needs to be *accurate* – the knowledge about the world needs to be correct, *robust* – it is free of noise and errors and is stable over time, and *comprehensive* – it contains all the information that the agent needs to accomplish its tasks. In order to satisfy these characteristics, the agent needs to take an active role in building a model of its environment. Although it does get information about its environment from perception, this information is not a sufficient representation for several reasons. First, it may contain noise or errors which could cause problems during task learning. Second, if the agent is performing tasks in a complex, multi-room environment that is only partially observable, it needs to remember and reason about objects that it cannot currently perceive. Third, if the agent is doing hypothetical reasoning or learning tasks that have non-perceptual aspects, it needs to incorporate this information into its model of the world.

In this chapter, we describe how our agent constructs and maintains the world representation it uses for learning and executing tasks. In doing so, it must handle the three cases above while bridging the divide between low-level non-symbolic information and more abstract symbolic representations This requires *anchoring*: "creating and maintaining the correspondence between symbols and sensor data that refer to the same physical object" (Coradeschi & Saffiotti, 2003), a special case of the symbol grounding problem. Anchoring can be both bottom up, where new percepts create and update anchors, and top down, where the system ties an object description to its corresponding percepts. We present a novel approach for an agent in a symbolic cognitive architecture with access to a spatial short-term memory to use spatial reasoning and domain knowledge to participate in the anchoring process. In doing so, the agent detects and corrects certain anchoring errors and



e_1 : sink, furniture, object, drain, receptacle, confirmed, not-reachable, not-visible
e_2 : counter, furniture, object, surface, confirmed, not-reachable, visible
e_3 : microwave, appliance, object, activatable, not-activated, openable, not-open, confirmed, not-reachable, visible
e_4 : mary, person, confirmed, reachable, visible
e_5 : kitchen, room, location, confirmed, visible, current
e_6 : mug, kitchenware, object, fillable, empty, grabbable, not-grabbed, green, confirmed, not-reachable, not-visible

in(e_1, e_5) in(e_2, e_5) in(e_3, e_5) in(e_4, e_5) in(e_6, e_5)
in(e_{robot}, e_5) in(e_6, e_1) on(e_3, e_2)

stopped(e_{robot}) current-waypoint(wp_{01})
current-location(e_5)

Figure 5.1: An example of the simulated mobile robot in the kitchen (left) and the corresponding unary properties and relations in the world representation.

constructs a stable world representation which is crucial for effective task learning.

Section 5.1 specifies the world representation used by the agent. Section 5.2 describes a novel approach for an agent in a symbolic cognitive architecture with access to a spatial short-term memory to use spatial reasoning and domain knowledge to participate in bottom-up anchoring and detect and correct certain errors. Section 5.3 describes how the agent does top-down anchoring by unifying information from long-term memory, instruction, and task knowledge with the world representation. Section 5.4 details an experiment with a real world tabletop arm and demonstrates how this approach can maintain object identity in a challenging perceptual environment.

5.1 World Representation

Our approach to interactive task learning uses a predicate-based world representation that contains information about the current state of the agent and its environment. These predicates are defined over a set of *entities*: $E = \{e_i\}$, where an entity represents some individual thing: an object, a person, a room, a message, etc. There is also an entity corresponding to the robot itself. Predicates over one entity are called properties, such as `red(`e_1`)`, `visible(`e_2`)`, or `grabbable(`e_3`)`, and predicates over two entities are called relations, such as `in(`e_1`, `e_2`)` or `right-of(`e_3`, `e_4`)`. An example of this world representation can be seen in

Figure 5.1. These predicates are stored in Soar's working memory. Entities that represent objects, such as a cup or table, also have metric information about their pose and bounding box volume stored in Soar's spatial memory (SVS). The agent has a simple object category hierarchy in its semantic memory, so when an entity of a given category is added to the world, all supercategories are also added (e.g., if an entity is added with `table(`e_i`)`, the properties `furniture(`e_i`)` and `object(`e_i`)` are also added). In addition, certain categories are associated with affordances predicates, and these are also added (e.g., for a counter the predicate `surface(`e_i`)` would be included). This predicate representation of the world that we use throughout the paper is an abstraction of the actual Soar structures, these can be seen in Appendix A.

5.2 Bottom-Up Anchoring

Maintaining this world representation requires *bottom-up anchoring* – unifying perceptual updates with the existing *anchors* (data structures that contain both symbolic and perceptual information about objects in the world). Bottom-up anchoring combines the problems of *data association*, updating the appropriate anchor with new measurements, and *tracking*, maintaining the anchoring correspondences over time as objects move (Elfring et al., 2013). These two problems are often tackled together through *multitarget tracking* or *MTT* (Vo et al., 2015) – locating multiple objects or targets in a video feed and tracking them over time. This is challenging because real-world robots often encounter objects that move, change in appearance, look identical, or occlude one another. In addition, perception often involves noise and errors. We have developed a taxonomy of anchoring errors that include five distinct types:

- **E1 False anchor:** An anchor is created for a non-existent object, usually due to sensor noise.
- **E2 Missing anchor:** An anchor does not exist for a known object (e.g., from occlusion).
- **E3 Misidentified anchor:** A object percept is not mapped to the correct anchor but is used to create a new one (e.g., due to a tracking error).
- **E4 Merged anchor:** One anchor exists for multiple objects (e.g., due to segmentation errors).
- **E5 Fragmented anchor:** Several anchors exist for one object (e.g., due to segmentation errors).

A common system organization in cognitive robotics is to have the agent that is responsible for the high-level reasoning and planning receive the final world representation as input from external perceptual and anchoring components. In this approach, the anchoring process is opaque to the agent, which only receives the symbolic portion of the anchors as the end result. The advantages of a separate, external anchoring component are that it can be fast, continuously active, and not interrupt the agent's deliberate reasoning. However, this separation can lead to two major issues.

First, the agent can have high-level knowledge that would be useful during anchoring. This can include knowledge of the agent's current actions and goals, object affordances and properties, and knowledge of environmental regularities. For example, consider a game of cups and balls where someone places a ball under a cup and then moves the cup. The agent cannot rely on direct perceptual information to track the ball, but it can use domain knowledge about containers to infer the ball has also moved. Or consider a robot that moves a block with its arm and during movement the block is occluded from the camera by the arm. The agent has high-level knowledge about its goal of placing that block at a given region that could be used to reacquire the track when the object reappears at the intended destination. A third example concerns an agent using a knife to cut an apple. If the agent has appropriate actions models, it can predict that one complete object will become two halves. These examples suggest that there needs to be a channel for knowledge to flow back into the anchoring component. However, many approaches to anchoring do not have mechanisms for incorporating knowledge in this way.

The second issue with this hard separation between the anchoring and cognition is that the agent lacks the information needed to detect and handle the anchoring errors described above. For example, suppose there is a misidentified anchor, where an object is assigned a new id. If the agent only has access to symbolic information, it is difficult to determine if an error actually occurred or if one object appeared at the same time another disappeared. Or consider if one block is placed upon another and occludes the bottom one, causing the anchor to become missing. If that occluded block is necessary for the agent to detect a goal as being satisfied, it will fail to detect it. In cognitive systems research, the possibility of these errors is sometimes overlooked. However, if a real-world robotic agent is relying on consistent, long-term object identity in order to complete its objectives, encountering anchoring errors can significantly degrade its performance. Improving the accuracy of the bottom-up anchoring can help, but no purely bottom-up approach is likely to be perfect and, as described above, the agent can have access to knowledge that can correct some of these errors that occur when low-level assumptions about the world are violated.

To overcome these issues, we developed a novel approach for an agent in a symbolic cognitive architecture with access to a spatial short-term memory to use spatial reasoning and domain knowledge to participate in the anchoring process. A key insight of our approach is that it maintains two representations of the world. As is common, a perceptual component performs performs standard bottom-up perception and anchoring to produce the *perceptual world representation* that is provided as input to the agent. However, this input not only contains the symbolic information stored in working memory, but also metric information in spatial memory. Crucially, the agent maintains a second *belief world representation* that it uses during planning and reasoning. It compares these two representations to identify discrepancies between them and then attempts to reconcile them. This process can update the belief world with new information or detect an anchoring error and attempt to reconcile it. Error handling can be internal (e.g., just ignore the error) or external (e.g., telling perception to merge two anchors). The agent uses knowledge about the world, object affordances, and its current tasks and actions to aid in this process. By using domain knowledge and participating in the anchoring process, the agent maintains its own world representation. This representation is more *accurate* (avoids certain errors), *robust* (ignores noise), and *comprehensive* (maintains information not currently perceived) than that given by perception.

5.2.1 Related Work

Previous research on detecting and recovering from anchoring errors (Bouguerra et al., 2006; Broxvall et al., 2005) has focused on top-down anchoring, where an ambiguous object description matches multiple percepts. Most of the work related to anchoring focuses on how to avoid producing errors, as opposed to correcting for errors that do occur. This can be done through better object persistence, object tracking, or object/motion models. The work of Loutfi et al. (2005) reduces missing anchor errors (**E2**) by adding object persistence to maintain anchors for objects that are out of view. The system of Blodow et al. (2010) also maintains beliefs about objects it cannot currently observe and then uses probabilistic anchoring to identify them when they come back into view, even after they have moved, to guard against misidentified anchors (**E3**). The work of Elfring et al. (2013) represents a major step in integrating a sophisticated tracking algorithm with anchoring, using a Multiple Hypothesis Tracker algorithm to improve tracking reliability during occlusion and detect clutter (false positives). This reduces the occurrence of anchoring errors **E1**, **E2**, and **E3**. Heyer & Graser (2012) use relative anchors to represent relations between two reference anchors. These help in noisy environments where detecting one object improves the chance of detecting the other. They also incorporate knowledge about actions to update the anchors of objects being moved. Persson et al. (2019) described a probabilistic reasoning system that infers the state of occluded objects and continue tracking through occluded movements. Nitti et al. (2014) implemented a relational particle filter that draws on commonsense world knowledge during tracking, such as when one object is inside another. These approaches to anchoring usually assume perfect perceptual segmentation (none of the percepts are merged or fragmented), so they are susceptible to errors **E4** and **E5**.

There has been a large body of work in multiple target tracking apart from anchoring (Vo et al., 2015; von Hoyningen-Huene & Beetz, 2009; Khan et al., 2006). These approaches can handle complex tracking scenarios with merged measurements and occlusions. Our approach is complimentary in that it aims to provide knowledge in situations where the model assumptions are violated (e.g., an object is fully occluded while the robot moves it) or where a tracking error occurs.

5.2.2 Perceptual Requirements

The Rosie agent has been integrated into a number of real-world and simulated domains, with different perceptual systems. Our approach does not assume any specific perceptual or anchoring algorithms, but does instill the following requirements on them.

Camera: Perception provides the camera's position, and for partially observable environments, a bounding volume in SVS approximating the 3D view region that the camera can perceive (ignoring occlusions).

Global Reference Frame: All positions and orientations provided by perception (e.g. for the robot, objects, and waypoints) are assumed to be properly localized within a fixed, global reference frame.

Robot: Perception maintains a 3D bounding volume of the robot in SVS that includes its position and orientation. If the robot has an arm, perception reports the id of any currently grabbed object.

Waypoints: For navigable environments, the agent uses a fixed waypoint graph for navigation. Perception reports the waypoint associated with the robot's current location (each waypoint corresponds to a region).

Objects: The perceptual component performs object detection, recognition, and anchoring and provide a set of objects as input into the agent (the *perceptual world representation*). These objects must include consistent identifiers tracked over time, position and bounding volumes that are placed into SVS, and classified predicates that are placed into working memory via the input-link. If perception recognizes object categories, the most specific one

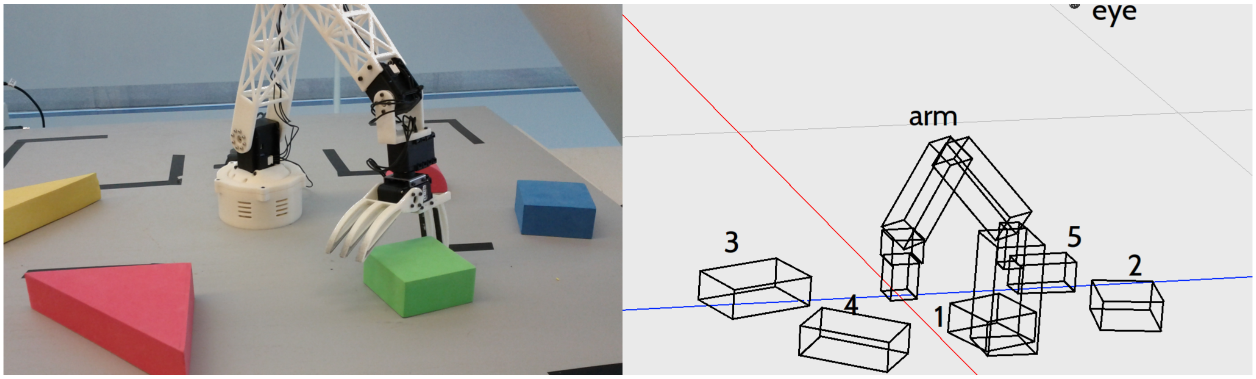


Figure 5.2: An example of the tabletop arm (left) and the corresponding spatial information put into SVS by the perceptual system (right).

should be included in the set of predicates.

People: If the perceptual system can detect people, it should represent them in the same way as objects, with the category predicate of person (both object and person are subcategories of entity in the category hierarchy), and a name or id of the person.

Anchoring: While our approach does not assume any specific anchoring algorithm, it does require that the anchoring algorithm must support three operations to correct anchoring errors:

- **move-anchor**(x_i, pos)**:** Changes the position of the anchor with the given id x_i.
- **change-anchor-id**(x_i, x_j)**:** Changes an anchor id from x_i to x_j.
- **merge-anchors**($x_i, x_j, ...$)**:** Merge the anchors with the given ids together (keep id x_i).

Figure 5.2 shows what is put into SVS for the tabletop environment, including bounding boxes for each object, the arm itself, and the camera's position.

When an update is received from perception, the symbolic portion of the objects (the identifying symbol and set of predicates) is placed onto the input-link in working memory and the metric portion (bounding volume information) is put into Soar's Spatial Visual System (SVS). This is a short-term spatial memory that provides a scene graph representation of objects and their bounding volumes. Metric information about the camera and robot is also put into SVS. For example, with the tabletop arm robot, it contains a volume for each segment of the arm.

SVS provides the necessary representations and computational infrastructure to support complex reasoning over metric data, such as computing intersections or occlusions. This reasoning would be very difficult to do using procedural rules alone. It uses a scene graph

Distance(x, y)	The distance between the object centroids.
Volume(x)	The volume of x's bounding volume
Intersects(x, y)	True if the bounding volumes of x and y intersect
Overlap(x, y)	The percentage of x's bounding volume that intersects y's
Occlusion(x, p)	The approximate percentage of object x visible from point p

Table 5.1: The SVS filters used to extract spatial knowledge about objects.

New-Object(x)	$x \in W_P$ and $x \notin W_B$
Missing-Object(x)	$x \notin W_P$ and $x \in W_B$
Moved-Object(x)	$distance(pos(x, W_P), pos(x, W_B)) > c_{move}$
Grown-Object(x)	$volume(x, W_P)/volume(x, W_B) > c_{grown}$
Shrunken-Object(x)	$volume(x, W_P)/volume(x, W_B) < c_{shrunk}$
Different-Predicate(x, p)	$p(x) \in W_P$ and $p(x) \notin W_B$
Different-Waypoint(w)	$cur\_wp(w) \in W_P$ and $cur\_wp(w) \notin W_B$

Table 5.2: The discrepancies that can be detected between the perceptual and belief world representations.

representation of objects together with a set of filter queries that extract symbolic, relational, and metric information from the scene graph representation and serve as an abstraction over it. Table 5.1 lists the filters used in this paper. The agent also calculates an `in-view(x)` predicate. In a fully observable environment (such as a tabletop arm), it is always true. In a partially observable environment, it is computed using `intersects(x,` o_{view}`)`, with the view volume provided by the perceptual system.

5.2.3 Discrepancy Detection

The agent has access to both the perceptual world representation W_P and belief world representation W_B in working memory and SVS. To detect differences between these two, the agent has procedural rules that continually compare the two world representations and detect discrepancies. These rules fire in parallel with other procedural knowledge and do not interrupt or interfere with the agent's deliberate reasoning. Only when a discrepancy is detected will the agent perform deliberate reasoning to determine its cause and resolve it. Table 5.2 shows a list of these discrepancies.

The constants c_{move}, c_{grown}, and c_{shrunk} are environment dependent parameters and determine which differences are considered significant. Due to perceptual noise, the position and volume values constantly change, even for a stationary object. Setting a low threshold causes the agent to spend more time reasoning about unimportant differences and decrease efficiency, but a high threshold may cause the agent to miss important changes and be less accurate. We use hand-tuned thresholds and reserve learning them as an area for future research. In the tabletop domain, $c_{move} = 2\text{cm}$, $c_{grown} = 1.2$, and $c_{shrunk} = 0.8$.

5.2.4 Discrepancy Resolution

Once a detector is triggered, the agent does deliberate reasoning (implemented as operators in Soar) about the detector. First, it uses SVS filters to gather more information and identify *why* the detector was triggered – whether it represents an actual change in the environment or some perceptual/anchoring error. Second, it takes the appropriate actions to resolve the error by using knowledge about physical properties of objects to guide reasoning. Implicit in this reasoning are assumptions that two objects cannot occupy the same physical space, objects occlude one another, objects maintain spatial and temporal continuity (i.e., do not teleport or vanish), and objects do not merge, split apart, or change in size. Obviously, these assumptions are sometimes incorrect (cutting an apple will split the object) or appear to be violated due to incomplete sensing (an object can appear to teleport when moved by a person). However, we have found that for our domains, they are sufficient for most cases. Below, we describe how the agent handles each of the five anchoring errors.

A **false anchor error (E1)** occurs when the perceptual system creates an anchor for an object that is not actually there and usually results from perceptual noise. To reduce the chance of accepting false anchors, when the agent sees a `new-object(x)` detector, it notes the time it appears and waits some time c_{new}. If the new object is still present, it is added to the belief world. Having a longer time makes the agent less susceptible to false positives, as they would have to span multiple perceptual updates, but less reactive to new objects. We set the constant c_{new} to be one second, which is a good compromise for our environments.

A **missing anchor error (E2)** occurs when the perceptual representation does not contain information for an known belief object. In this case, the `missing-object(x)` detector will be triggered and there will not be any corresponding `new-object` detectors. The agent must determine if the object is actually no longer present or simply not currently visible. To do this, it checks whether the object is being occluded (`occlusion(x, eye)` $> c_{occlusion}$) or no longer `in-view(x)`. If either of these conditions is true, the agent concludes the object must still be present, so it keeps the belief anchor. In this way, agent deals with partial observability by maintaining anchors for objects that are not current visible. Here we use $c_{occlusion} = 0.2$.

A **misidentified anchor error (E3)** occurs when a new anchor is incorrectly created from a percept instead of the appropriate existing anchor being updated. In this case, the `missing-object(x)` and `new-object(y)` detectors are triggered. The agent checks to see if the bounding volume of the new anchor overlaps the missing one. If `overlap(y, x)` $> c_{overlap}$ (and there are no conflicting properties), then it merges the anchors by sending a `change-anchor-id(y, x)` command to the perceptual system to correct the tracking error. Here we use $c_{overlap} = 0.5$.

A **merged anchor error (E4)** occurs when an anchor incorrectly contains percepts that correspond to multiple objects. This can happen if the perceptual component fails to segment two or more objects into distinct percepts. The agent can correct for this if at one point the anchors were not merged. When a `missing-object(x)` detector triggers, the agent checks for overlap between the missing object and the current bounding volume information for perceptual objects. If there exists a perceptual object y where `overlap(x, y)` $> c_{overlap}$, the missing object's bounding volume is contained inside another's volume, and the agent deduces that the percepts for object y belong to both x and y. Thus the object is no longer labeled as missing.

A **fragmented anchor error (E5)** occurs when multiple anchors represent a single object. This can happen if the perceptual component over-segments an object into multiple fragments. The cognitive component can correct for this if at one point the anchor was not fragmented. One or more `new-object(x)` detectors will be triggered, and the agent checks whether there is some belief object y whose bounding volume contains the new object. If `overlap(x, y)` $> c_{overlap}$, then it will send the command `merge-anchors(x, y)` to the perceptual component.

If the agent detects none of the above errors, then it updates its belief world representation with the new perceptual information. In the case of `new-object(x)` or `missing-object(x)`, the anchor is either added or removed from W_B. In the other cases where the position, volume, or predicates differ between the two anchors, the cognitive component checks for occlusion (`occlusion(x, eye)` $> c_{occlusion}$). If the object is not occluded, it updates the belief anchor with the new information and otherwise ignores the discrepancy. This means the agent only updates a belief about an object if it has an unobstructed view.

Resolving discrepancies requires deliberate reasoning, which competes with the agent's other tasks. How much this happens depends on the environment and perceptual system. If the environment is relatively static and the perceptual system is reliable, then the agent only occasionally needs to update its belief. In most cases, resolving a discrepancy only takes a few decision cycles ($<$ 10ms) and there are hundreds between perceptual updates. However, the agent can become overwhelmed if there are hundreds of changes per second. In such cases, it can choose to delay the perceptual processing and give priority to other cognitive activities, at the cost of decreased reactivity.

5.3 Top-Down Anchoring

In addition to bottom-up anchoring, the agent also needs to do top-down anchoring, where symbolic entity descriptions are unified with the world representation and grounded to perceptual information if possible. For our ITL agent, there are three main situations where this occurs. First, when the agent re-enters a location, it retrieves an episode from episodic memory for the last time it was there, and reconstructs the world representation for that location. Second, during language comprehension the agent needs to do reference resolution to connect a referring expression to an entity in the world representation. Third, when the agent is performing a learned task which involves implicit arguments (arguments that are used in the task but not included in the task command), it needs to connect entity descriptions in its task knowledge to the world. These are described further in the following sections.

5.3.1 Previous Location Episodes

In a simple environment, the agent can keep track of all the physical objects. However, with a mobile robot operating across multiple rooms in real-world environments, the capacity to track everything is quickly exceeded. To address this limitation, we assume that it is reasonable to only keep track of the objects in the robot's immediate surroundings or those related to the its current goals. Our approach involves having a known map of the environment, which is divided into convex regions called *locations*, usually a room or a part of a hallway. The robot's current location is included in perceptual updates. When the agent moves to a different location, it sees this as a context switch and clears its belief world representation, except for those entities that are involved in the current task.

This approach reduces the number of objects being tracked at any one time and allows the agent to work within multiple rooms and with their associated objects. However, this comes at the cost of deleting information that might be needed later. For example, suppose the agent stores a certain mug inside a cupboard and then leaves the room. If later it is back in that room and needs to find the mug, it would have to search the entire environment. Soar has a long-term episodic memory that automatically records the history of working memory and includes the world's symbolic information. However, it does not include the spatial information; in Soar there is no long-term spatial memory.

To overcome this limitation within the current architecture, when the agent leaves a location and before it removes the entities from its belief world representation, it performs queries to SVS to extract the metric information (position, rotation, and scale) from each physical object and adds it to working memory. This snapshot is then automatically recorded in episodic memory. When the agent enters a new location, it performs an episodic memory retrieval cued on the last time it was in that location. This retrieves an episode that includes this snapshot of spatial memory and provides enough information to reconstruct the previous world representation for that location. The room might have changed since it was last visited, but any incorrect beliefs can be fixed as the robot drives around.

5.3.2 Linguistic References

During interaction with the agent, the instructor will reference different entities. The agent must eventually resolve those references to a suitable entity. To do so, the agent needs to first do *reference resolution*; taking a linguistic reference and either identifying an existing representation or creating a new representation for the referenced entity. Our approach is inspired by the *Givenness Hierarchy* (Gundel et al., 1993), which describes how to associate the form of the reference expression (e.g., pronoun, definite noun phrase) with a cognitive status (e.g. in-focus, uniquely identifiable). We use this cognitive status to determine which candidate entities to consider.

In our system, we handle four of the cognitive statuses (Table 5.3). The *in focus* status is indicated by the word *it* and indicates an entity in the current center of attention. The candidate set is the set of entities previously mentioned in the dialog within the current task (DL), with a preference for more recent entities that match usage constraints (for example, if 'it' is used as the direct object of 'move', the referent must be a physical object and not a location). The *activated* status is indicated by the word *this*, and indicates an entity that has some recent salience. Currently, this is only used when the instructor 'points' to an object (clicks on a graphical interface). The *uniquely identifiable* status is indicated by a noun phrase with the word *the*, and is the most common form that our agent encounters. This status is also assigned by default if no form is given. Here the instructor is indicating that he expects that the agent can resolve the reference to a unique entity based on the constraints in the reference. Here, the agent looks through the following candidate sets in order until it finds an entity that satisfies all the constraints: entities referenced in previous dialog (DL), currently visible entities (VIS), all entities in short term memory (STM), and all entities in long term memory (LTM). If no match is found, it generates a NEW representation containing the constraints in the referring expression. Finally, the *type identifiable* status is

Cognitive Status	**Linguistic form**	**Candidate Set**
In Focus	*it*	DL
Activated	*this, that, this NP*	ACT
Familiar	*that NP*	*not used*
Uniquely Identifiable	*the NP*	DL > VIS > STM > LTM > NEW
Referential	*indef. this NP*	*not used*
Type Identifiable	*a NP*	NEW

Table 5.3: Candidate entity sets considered during reference resolution based on the linguistic form of the referring expression: previous dialog (DL), activated (ACT), visible (VIS), short term memory (STM), long term memory (LTM), and a newly created representation (NEW).

indicated by a noun phrase with the word *a*, and indicates that the instructor is referring to a class of entities, not a specific one. In this case the agent generates a NEW representation that includes the constraints in the reference. The end result of this process is that the reference has been resolved to an internal representation, either existing or newly created. If a given reference is ambiguous and resolves to multiple world objects, one is chosen at random.

This reference resolution is performed during language comprehension, which produces an internal message structure that the rest of the agent understands. If an entity in this structure is already in the world, the agent does nothing more. However, if the entity is not already in the world (e.g., for a LTM or NEW entity), the agent will attempt to anchor it to an existing one. It will look for an entity that matches all of its predicates. If one exists, the agent will anchor the message entity to the matching entity and substitute the matching entity into the message structure. If no match is found, the new entity is added to the world without having any metric information in SVS. for example, suppose the instructor says *'Fetch a stapler.'* The referring expression *'a stapler'* has an indefinite article and thus is given the cognitive status of *Type Identifiable* and a new representation is created with the single predicate of `stapler(`e`)`. The agent will then try to anchor this new entity. If a stapler does exist in the world, then this new representation will be anchored to it and the fetch command will be connected to that specific stapler. Otherwise, the new and ungrounded entity will be added to the world. If the stapler later becomes visible, it will be anchored to the ungrounded entity.

5.3.3 Implicit Task Arguments

Some tasks involve arguments that are not explicitly stated in the task command. For example, for the task *'Discard the paper'*, the goal is that the paper is in the garbage can.

The garbage can is an implicit argument of this task. When the agent begins to perform a task, it retrieves all implicit entity arguments from the Task Concept Network in semantic memory and performs top-down anchoring to add them to the world. This process is the same as for new entity references in the previous section. An implicit entity argument is represented as a collection of predicates. If there is an existing entity in the world that has all of those predicates, the implicit argument is anchored to that existing entity. Otherwise, it is added to the world as a new, ungrounded entity. This allows the agent to plan and reason about task entities that it cannot see.

5.4 Empirical Evaluation

We have used this approach to support agents that can learn tasks from instruction in several domains, including the tabletop arm, a mobile robot that operates across multiple rooms (Mininger & Laird, 2016), a simulated indoor kitchen environment, and a toy forklift robot. Although these environments differ in their embodiments and perceptual processing, our method provides a stable and reliable representation of the world across these different domains. To evaluate this approach, we focused on the tabletop environment.

5.4.1 Experimental Design

We evaluated performance on an end-to-end task in the tabletop environment with an arm that can manipulate foam blocks. A suitable task must require that having an accurate belief world representation is necessary to perform the task well and that the agent will be exposed to common perceptual challenges likely to induce various anchoring errors. We chose the task of measuring two nonvisual properties of objects (temperature and weight), finding the object with the highest or lowest value of that property, and placing it onto a goal region. This is referred to as a *superlative request* (e.g., find the heaviest block). These properties are simulated and can only be attained through a measurement action involving placing the block on a certain region on the table (e.g., the scale). This task is difficult because the agent must know the values for all the objects before being able to achieve the goal. Every time it fails to track an anchor, it must remeasure the object by moving it back onto the relevant measurement region.

Our evaluation metric is the number of movements it takes to satisfy each superlative request. The more accurately the agent can maintain the world representation, the fewer remeasurements are needed. This evaluation does not directly record how many anchoring errors the system makes, but the performance on this task is directly tied to how well the

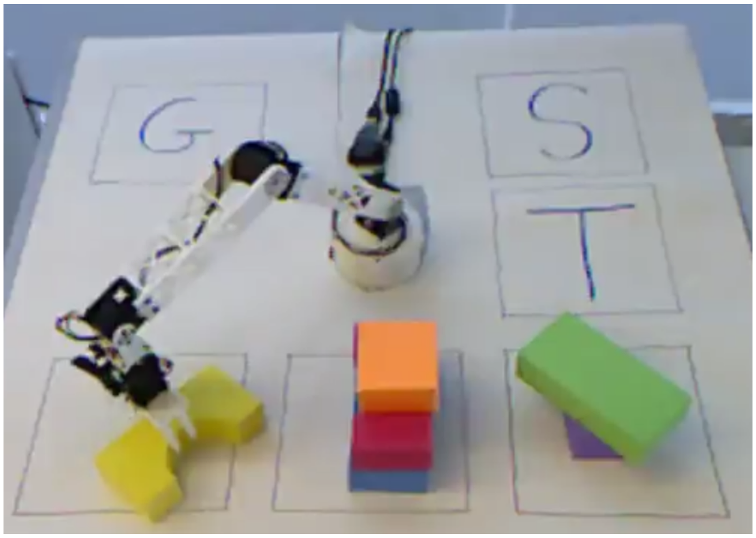


Figure 5.3: The second domain variation D2 with three bins to stack objects and goal, scale, and thermometer regions (labeled G, S, and T).

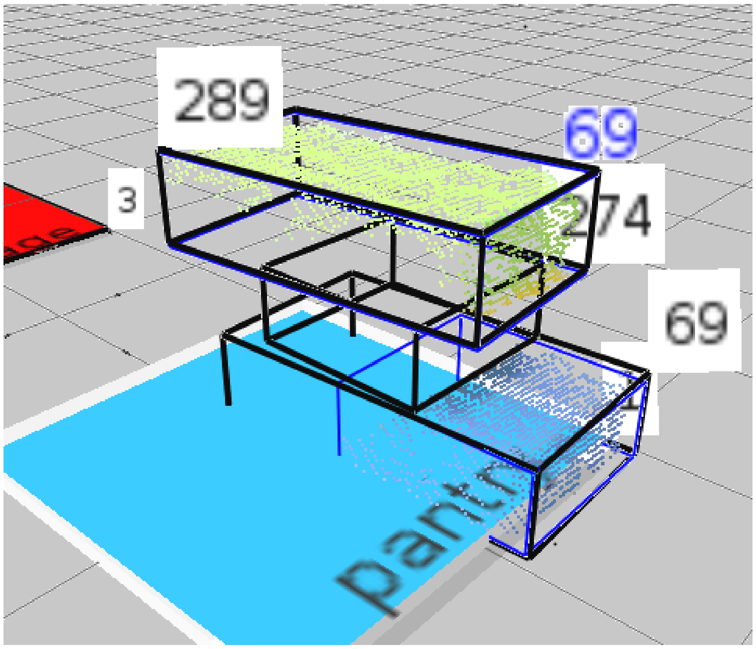


Figure 5.4: The point cloud data from the Kinect camera during stacking showing total occlusion of the middle block and partial occlusion of the bottom one. The thick black borders are the belief volumes for each object in SVS (it never resized them due to detected occlusion).

agent can detect and recover from errors. We evaluate performance in this way because ground truth is difficult to obtain and we want to examine the influence of errors on end behavior. If the agent does manage to measure and track all six objects, it will always identify the correct block. For some agent and domain combinations this task is extremely difficult, so we consider the run a failure once the agent has done thirty moves without finding the correct block.

The blocks are placed onto a tabletop with the robotic arm at the center. Different areas of the table are designated as named regions where the arm can put objects. The *goal* region is where the agent places the requested block, whereas the *scale* and *thermometer* regions are used to gather weight and temperature information. Figure 5.3 labels these regions as G, S, and T, respectively. There are also several *bin* regions where the agent must place blocks it is not using. If there are multiple blocks on a bin, the agent must stack them. This is difficult due to the partial or total occlusion for the lower blocks (Figure 5.4).

We evaluated four versions of the agent across four versions of the domain. Each agent includes all the capabilities of the previous versions plus an additional one (Table 5.4.1). This gives a qualitative comparison of how adding error handling capabilities improves the overall performance. We tested each agent version across four variations of the domain, which vary in complexity and perceptual difficulty (Table 5.4.1). In each one, the task of finding objects is the same, but the objects and configuration of the table differ.

We ran each combination of agent and domain using the same script of find requests. We gave the agent two requests for the same type of superlative, with five random moves in between. For example, it would find the lightest block, randomly move five blocks, then find the lightest block again. These intermediate moves provided more opportunities for anchoring errors to occur. We report the average number of moves required to satisfy the second superlative request. The better the agent is at handling anchoring errors, the fewer remeasurements the agent will make, since it can *immediately* satisfy the request if all the anchors were successfully maintained.

5.4.2 Experimental Results

Figure 5.5 shows the performance of each agent type in the different domain variations averaged across four runs. These results are summarized below.

- Agent **A1**, which uses only the perceptual world representation to make decisions, always fails, even the easiest variation. This is because the anchoring component makes assumptions during tracking that are broken when the arm moves an object (due to arm occlusion, the block disappears and reappears somewhere else). When this occurs, the

A1. No Error Handling	The agent relies exclusively on perception for planning
A2. +Action Knowledge	The agent notifies the anchoring component when it moves a block
A3. +Object Permanence	The agent can reason about occlusions and maintain anchors to handle errors related to **E1**, **E2**, **E3**.
A4. +Segmentation Reasoning	The agent does not assume one anchor per object, and can handle errors relating to **E4** and **E5**.

Table 5.4: Different agent variations, each one includes the capabilities of those previous.

D1. No Occlusion 6 bins and 6 blocks	This has one bin per block, so no stacking or occlusions, making segmentation easy. There should be minimal anchoring errors.
D2. Partial Occlusion 3 bins and 6 blocks	The agent must stack blocks, and so it has to deal with partial occlusion of objects lower on the stack (Figure 5.3). Stacking greatly increases the chances of fragmented segmentations (**E5**), tracking errors (**E3**), and noise (**E1**).
D3. Total Occlusion 3 bins and 6 blocks Only 1 bin visible	In this version, we artificially restrict perception so that the agent can only view objects in one bin at a time. The agent must deliberately select which bin it can observe. Thus the perceptual anchor set will be missing anchors for occluded objects (**E2**).
D4. Merged Percepts 3 bins and 6 blocks 2 block colors	Because there are only 2 colors, stacked blocks of the same color are often segmented as one object. The anchoring component will produce anchors that correspond to multiple merged objects (**E4**).

Table 5.5: Different versions of the domain that vary in difficulty.

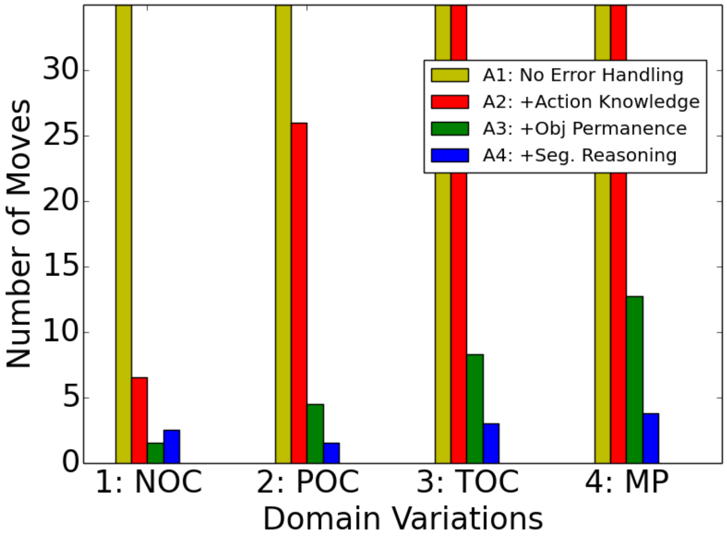


Figure 5.5: The number of moves to satisfy each request for a previously seen superlative averaged across four runs. The best possible score is 1. Bars above 30 indicate failure.

agent loses the nonvisual information needed for the task.

- Agent **A2**, which uses knowledge about its actions to inform the anchoring component when it moves an object, can maintain correct anchors through movement, but cannot recover from anchoring errors. Thus, it succeeds in variation **D1** but has a very difficult time on the harder variations where there are many more anchoring errors.
- Agent **A3**, which does have the knowledge of object permanence, can deal with errors due to partial and total occlusions in the harder variations. It successfully completes the task across all variations. However, in **D3** and **D4**, there are more errors due to anchors being merged or fragmented that causes it more difficulty.
- Agent **A4**, which has the complete set of knowledge (including how to handle errors relating to merged and fragmented anchors), performs well in all of the domain variations, with only a small decrease in performance on the harder ones.

These results show that the error handling capabilities of the agent makes the system more *robust*. Without those capabilities, the number of anchoring errors make the task impossible for the agent to perform, as demonstrated by **A1**. As error handling capabilities are added, the overall performance of the system improves. Another desired characteristic is that the perceptual reasoning is *efficient*. It is important that the extra processing required to handle the errors and maintain the separate belief world representation does not significantly slow processing. The agent must remain reactive to new updates and changes in the environment. To evaluate the overhead of our approach, we measured the average time per arm movement that the agent spent processing the perceptual updates and maintaining the belief world representation. The basic agent **A1** spent 410 ms per arm movement, while the full agent **A4** spent 331 ms per movement. Qualitatively, this shows that the error-handling capabilities does not slow the agent, and may sometimes make it faster by letting it ignore perceptual noise. Since each arm movement takes around fifteen seconds, the agent spends less than three percent of its time handling perceptual updates, most of which are done while the arm is moving when the rest of the agent is idle.

5.5 Discussion

In this chapter, we demonstrated an approach for allowing an agent in a cognitive architecture to construct and maintain the world representation it uses for learning and executing tasks. It does this by integrating information from many different sources, including perception, language comprehension, task knowledge, semantic memory, and episodic memory. A key idea is keeping perceptual updates separate from the belief representations (in working memory and SVS), and performing deliberate reasoning to attend to differences and resolve them. This results in a more stable world representation that supports the long-term task learning that Rosie performs.

This approach has been sufficient for the environments that we have used Rosie for, but there would be some significant challenges in migrating to more complex real-world domains. First, this approach assumes that objects are stationary, or move fairly slowly. SVS does not have any dynamics, so it does not represent velocities or accelerations. Adding these could help with prediction and tracking. Second, the world representation is still fairly high-level, and does not account for uncertainty or non-rigid objects. Discovering methods of combining lower level perceptual techniques with high-level symbolic representations is still a major unsolved research area.

CHAPTER 6

Executing and Learning Tasks

Before we discuss the specific extensions implemented to address the three dimensions of task complexity, we first give an overview of how Rosie executes and learns tasks. Our approach is a successor to Mohan & Laird (2014), although it involves a complete re-implementation and involves several crucial differences as will be described below. The core idea is to represent a task as a *problem space* (Newell, 1993) – a space defined by states (sets of symbolic structures) and operators that transform the state. The *problem* is to find a sequence of operators (path) through the space constrained by path constraints and ending at a goal state. In the context of task execution, the state is the current world representation, operators are other tasks, the goal states are defined by the task goal, and path constraints are imposed by task preconditions. Thus, the core approach is to frame hierarchical task execution as a selection problem over possible subtasks given the current goal. The main strength of this method is that the agent only needs to learn the goal in order to execute a task, as it can use planning to find a sequence of operators (subtasks) to achieve it. However, not every task naturally fits this formulation, as we discuss in Chapter 8. A major contribution is layering a procedural goal graph on top of the problem-space formulation, which supports both goal-based tasks and procedural tasks.

Learning a new task involves acquiring all the problem space knowledge. This knowledge consists of the task representation, the goal representation, and a set of possible subtasks. In order for a learned task to be used as part of another, the agent must also learn its preconditions and postconditions. The key challenge is to learn this from specific instances and generalize to different task variations. Our approach uses explanation-based generalization (EBG; Mitchell et al. 1986) with extensions by Mazin Assanie, to achieve this. EBG operates on the insight that it is possible to form a justified generalization from a single example if it has an explanation of why the example achieves the desired outcome. Explanation-based techniques are the key to allowing Rosie to learn tasks in one-shot with significant generalization and transfer.

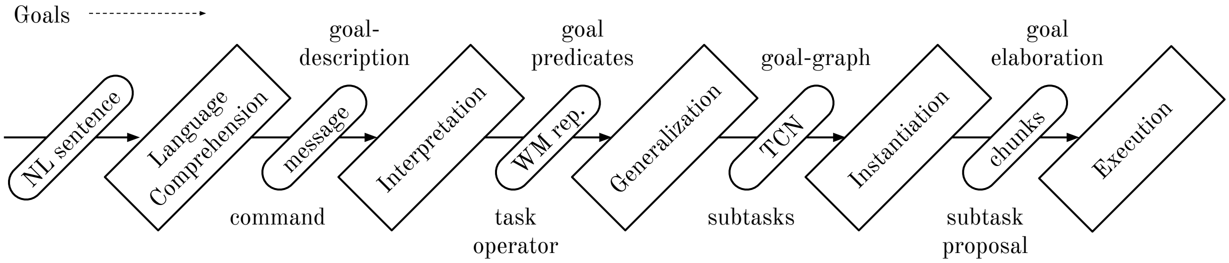


Figure 6.1: The processing stages for going from instructions to procedural task knowledge (rectangles) and the intermediate knowledge (ovals).

The overall process for going from instructions to operationalized task knowledge is shown in Figure 6.1 for both goals and subtasks. The sentence goes through language comprehension to produce a message structure (Section 4.2.1). During interpretation, the message is used to construct a grounded task operator or set of goal predicates. These are then generalized and added to the task concept network by connecting arguments to those in the parent task or adding default versions. Finally, when this knowledge is interpreted within the current task context to create instantiated representations, that deliberate processing is compiled into procedural rules (subtask proposal and goal elaboration rules).

In this chapter, we first compare our work to some common alternative representations suitable for learning hierarchical tasks from instruction (Section 6.1). We then give an overview of how Rosie executes tasks (Section 6.2), learns the Task Concept Network (Section 6.3), and learns the various types of procedural task knowledge (Section 6.4). We close with a summary of the distinguishing features of our approach (Section 6.5).

6.1 Alternate Representations

There are many different options for representing hierarchical tasks that would support learning through instruction. One approach is to learn tasks as STRIPS operators (Fikes & Nilsson, 1971), where operators have predicate-based preconditions and effects. In this framework, executing a task would involve finding a sequence of operators that would achieve the desired effects (goal) of the task. Our task knowledge shares much in common with STRIPS operators, and we do similar forward planning. However, the standard formulation cannot represent the more complex control structures that we learn in this work, such as loops and conditional goals, and tends to operate at one level of abstraction, so it does not satisfy the hierarchical and compositional requirements.

Another common approach is using Hierarchical Task Networks (HTN; Erol et al. 1994), where the network is a tree with the leaves being primitive tasks that are directly executable, and intermediate nodes being more abstract tasks that include knowledge of how to decompose the task. The subtasks at a given node are usually fixed (Mohseni-Kabir et al., 2015), though some approaches do use planning to satisfy subtask preconditions (Suddrey et al., 2016). Our approach is more flexible, as it can represent a task as a sequence of subtasks (in the goal graph), but can also use more complex structures such as conditionals and loops, and can also frame a task as achieving a goal and thus leverage classical planning techniques. An alternate representation is a Hierarchical Goal Network (HGN; Shivashankar et al. 2012), that represents a task as a sequence of subgoals. This is more in line with previous work (Mohan & Laird, 2014), but cannot represent the procedural control structures that we demonstrate in this work.

An alternative to HTN's is to representing a task as a procedural graph structure that encodes the control flow of the task (Meriçli et al., 2013; Gemignani et al., 2015). The main strength is that they can represent complex control structures such as loops and conditional actions. They also are simple to execute, extend, and explain. However, they are less flexible to task variability as any alternate plans need to be directly encoded. Rosie's goal graph representation is inspired by these approaches, but it can also represent goals and supports hierarchical planning.

6.2 Task Execution

When Rosie receives a new task command, it must construct a representation of that task and then execute it, usually by decomposing it into a number of subtasks and recursively executing each one. In the following subsections, we describe this process for executing a task that is fully learned. Throughout we use the same example task of *"Discard the white cup"* from the previous chapter.

6.2.1 Constructing the Task Operator

Once a task command is given and the language comprehension phase produces a grounded `command` message structure, the agent pushes a new segment onto the dialog stack with the purpose of `execute-task` and satisfaction of `action-event(<task>)`. The `command` message structure produced during language comprehension contains all the information needed to instantiate the task, but is not in the proper representation. The agent transforms the command message into a task operator:

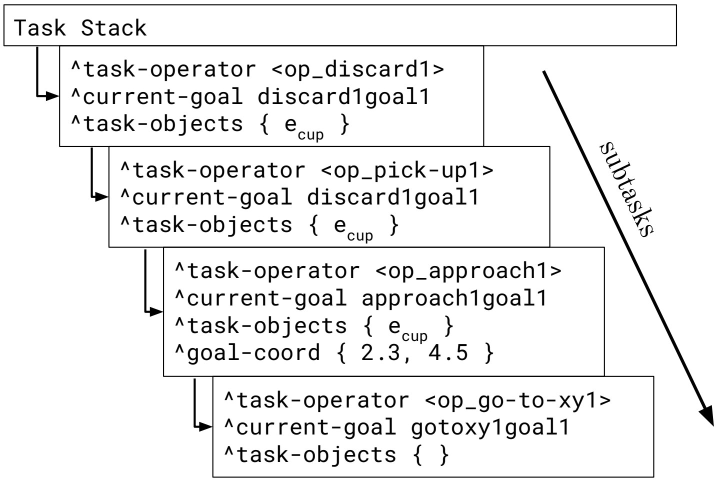

Task Stack
^task-operator <op_discard1>
^current-goal discard1goal1
^task-objects { e_{cup} }
^task-operator <op_pick-up1>
^current-goal discard1goal1
^task-objects { e_{cup} }
^task-operator <op_approach1>
^current-goal approach1goal1
^task-objects { e_{cup} }
^goal-coord { 2.3, 4.5 }
^task-operator <op_go-to-xy1>
^current-goal gotoxy1goal1
^task-objects { }
subtasks

Figure 6.2: The task stack at a specific moment during the discard task.

```
(<discard> ^name op_discard1
           ^item-type task-operator
           ^task-handle discard1
           ^arg1 <arg1>)
  (<arg1> ^arg-type entity
          ^id <cup>)
```

Note that `<cup>` is the root identifier of an entity in the world representation and has its own substructure. In our representations, we use the term *handle* to refer to a uniquely identifying string for a concept, such as `discard1`. This allows the agent to retrieve the precise concept from smem when needed. For clarity, we usually write such a task as `discard(`e_{cup}`)`. This is an abstraction over the actual Soar representation shown above. A full specification of Soar representations for tasks, goals, and TCN's is provided in Appendix A.

6.2.2 Managing the Task Stack

After constructing the task operator, the agent pushes a new segment onto the *task stack*. The task stack is a structure in working memory that stores the current task decomposition. It is comprised of *task segments*, each containing the task operator, current goal handle, and any entities involved in the task. The task stack is a novel extension from the previous work. Deliberately representing this stack helps overcome two problems that occur when using operators to represent tasks. First, it provides a place to store task information if the agent switches to other processing (such as language comprehension) and returns later. Second, it ensures a more deliberate way of handling task lifetimes than regular Soar operators, where the operator goes away if the preconditions are no longer met. Using a declarative structure allows the agent to perform extra steps to cleanly exit each task. This does come at the cost of slightly reduced reactivity.

6.2.3 Executing the Task

Once a task is pushed onto the stack, the agent selects the task operator and a substate is created. This substate represents a problem space with the goal of executing the task. The general processing that takes place in this problem space is described in Algorithm 1. The problem space has access to the $World$ and the $TaskStack$ from the top state.

Algorithm 1 Pseudocode description of the execute task problem-space

```
 1: function EXECUTE-TASK($Task$)
 2:     uses $World$, $TaskStack$
 3:     delete $Task.StartOfExecution$
 4:     $Task.StartTime \leftarrow World.CurrentTime$
 5:     $Goal \leftarrow$ RETRIEVE-START-GOAL($Task$)
 6:     while !(satisfied($Goal$) and terminal($Goal$)) do
 7:         ATTEND-TO-PERCEPTION($World$)
 8:         if satisfied($Goal$) then
 9:             $Goal \leftarrow$ SELECT-NEXT-GOAL($Task$, $World$)
10:             for $obj$ in IMPLICIT-ENTITIES($Goal$) do
11:                 ADD-ENTITY-TO-WORLD($obj$)
12:             end for
13:         end if
14:         $Subtask \leftarrow$ SELECT-NEXT-SUBTASK($Task$, $Goal$, $World$)
15:         if primitive($Subtask$) then
16:             $Command \leftarrow$ SEND-OUTPUT-COMMAND($Subtask$)
17:             WAIT-FOR-COMPLETION($Command$)
18:         else
19:             $SubtaskCopy \leftarrow$ COPY-TASK-OPERATOR($Subtask$)
20:             PUSH-TASK-SEGMENT($TaskStack$, $SubtaskCopy$)
21:             EXECUTE-TASK($SubtaskCopy$)
22:             POP-TASK-SEGMENT($TaskStack$)
23:         end if
24:     end while
25:     REPORT-ACTION-EVENT($Task$)
26: end function
```

First, the agent performs a few initialization steps (lines 3-5). It first removes the $StartOfExecution$ flag from the task. This flag is added when the task is pushed onto the stack, and is used later by the agent to retrieve the episode from episodic memory representing the beginning of the task. The agent also marks the *StartTime* of the task, which is used for those involving duration (e.g. *'Wait for ten seconds'*), and retrieves the start goal from the goal graph in semantic memory.

Then, the agent repeats a general execution cycle (lines 6-24) until it has satisfied a terminal goal. First, it attends to any perceptual updates that have occurred (Chapter 5), and updates the world accordingly (line 7). Then, if the current goal is satisfied, the agent selects the next goal and adds it to the state, along with any implicit entities (lines 8-13). Once it has a goal, the agent proposes operators representing subtasks it can perform and selects the best one according to a policy (line 14). If that subtask is a primitive action, meaning that it represents an output command the motor system can execute, the agent sends the command and waits for it to be completed (lines 15-17). While waiting, it is also attending to any new perceptual updates that occur. Once the motor system indicates that the command is finished, the cycle will repeat. If the subtask is not a primitive action, then the agent recursively executes that subtask by copying it, pushing it onto the stack, then executing it and going into a further substate (lines 19-21). Once the substate has completed, the subtask segment is popped from the stack.

6.3 Task Concept Network

The general procedure for executing a task shown in Algorithm 1 occurs when Rosie has complete knowledge of a task. Note that most of that knowledge is encoded as procedural rules. For new tasks, Rosie must learn all of the different types of task knowledge. Most of this happens in a two stage process. First, the agent learns a declarative representation of the task in semantic memory called the Task Concept Network (TCN). Then during task execution, that knowledge is interpreted within the current context and it is compiled into rules through Soar's chunking mechanism.

The TCN represents parameterized knowledge about the task, including its representation, goals, and subtasks. An example for the discard task is shown in Figure 6.3. The TCN serves two main purposes. First, it stores the connections between arguments in the task, goal, and subtasks. For example, in Figure 6.3, it shows that the entity argument e_i for `discard(`e_i`)`, `pick-up(`e_i`)`, and the goal `in(`e_i`, `$e_{garbage}$`)` are all the same. When this knowledge is compiled into procedural knowledge, those connections will be embedded in the rules. This is the key to the agent's generalization and transfer. Second, it contains the goal graph, which can represent complex control structures and allows the agent to extend these incrementally. In the following sections, we describe each of the three main parts of

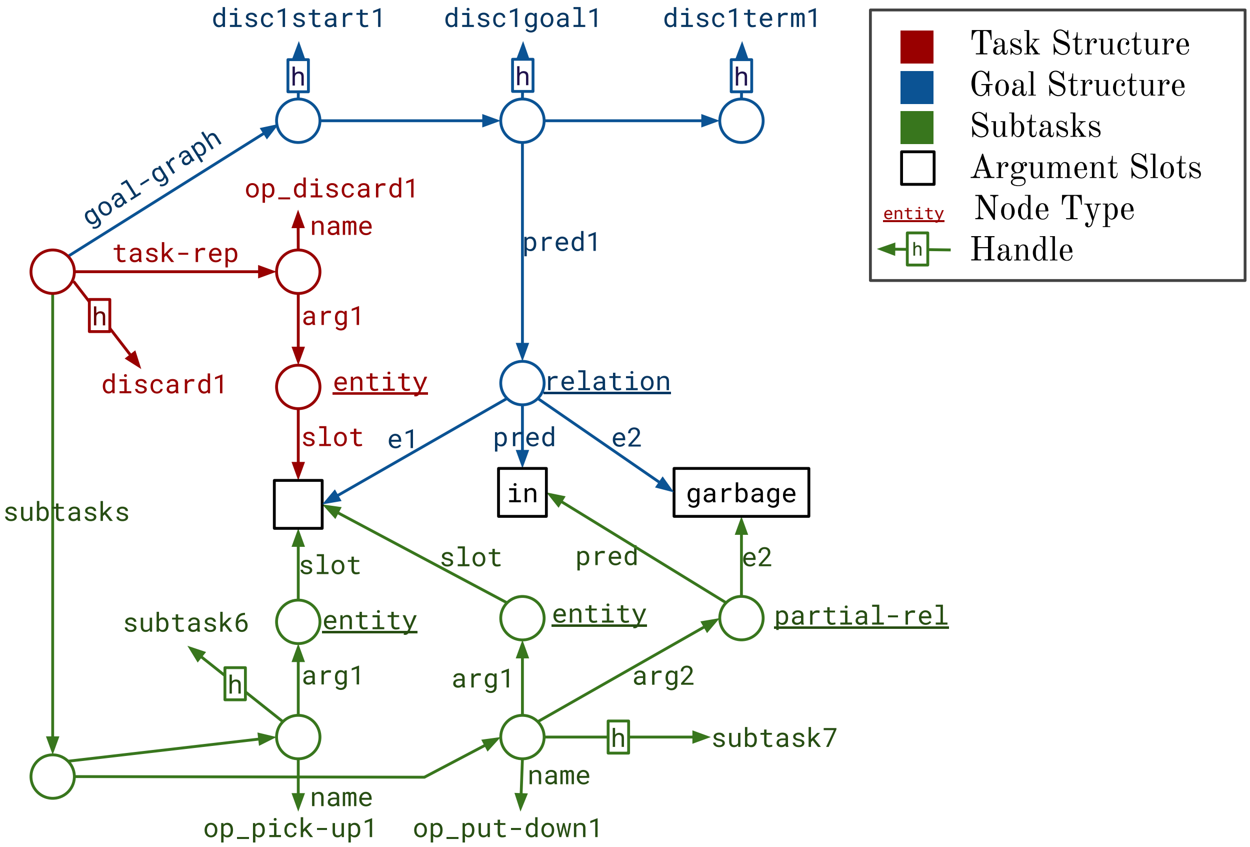


Figure 6.3: The Task Concept Network learned for the task *"Discard the white cup."*

the TCN (the task structure, goal graph, and subtasks) are learned and represented.

Note that throughout this dissertation, we will show graphical depictions of TCN's such as in Figure 6.3. These are slightly simplified from the actual Soar representations and omit minor implementation details. The important structure of the TCN is always faithfully shown. Appendix A gives a full specification of the TCN representations in semantic memory.

6.3.1 Task Structure

The agent creates a new TCN when it receives a task command for the first time. It stores the task representation (the structure of the task operator) by extracting the argument structure from the command, replacing references to specific entities or predicates with *slots*, which are placeholders that will be filled in during a specific task execution. For example, with *discard,* there is a single argument, an entity (the white cup), so the TCN is initialized with the task name and a single entity argument slot (figure 6.3). More complex commands can have multiple arguments, e.g. *'Deliver the green box to the office.'*

6.3.2 Goal Graph

In previous work, tasks were constrained to only having a single goal representing a set of state predicates to establish. We present a more flexible goal representation that can accommodate a wider range of tasks: a *goal graph*.

6.3.2.1 Goal Graph Representation

The goal graph has four types of nodes: *goal nodes* that represents a goal the agent should pursue, a single *start node* that indicates where the agent begins, *intermediate* nodes that are empty placeholders used within more complex graph structures, and *terminal nodes* that indicates when the task is complete. Directed edges between two goals indicate a possible choice of pursuing the latter once the former is satisfied. Thus executing a task involves following a path through the graph starting at the start node and ending at a terminal node. Edges can have conditions, meaning the tail goal can be selected only if the conditions are met. Goal nodes can also represent the purpose of executing one or more subtasks, which are used for procedural formulations (Chapter 8).

This representation offers much greater flexibility for representing different kinds of tasks. A simple goal-based task is easily represented as a single start node, goal node, and terminal node (as in Figure 6.3). A sequential procedure is represented as a single sequence of goals, where each goal is to perform the next action in the procedure. However, the graph structure is general enough to represent more complex control flow such as conditionals and loops (Chapter 9).

6.3.2.2 Learning Individual Goals

Whenever the agent detects that the current task goal is satisfied, it proposes possible `select-next-goal` operators to choose the next goal to pursue. If there are no such operators, the agent encounters a state no-change impasse and asks the instructor *"What is the next goal or subtask?"* Suppose the instructor replies *"The goal is that the cup is in the garbage,"*. Through natural language comprehension, this is interpreted as a set of one or more predicates: `{in(`e_{cup}`, `$e_{garbage}$`)}`. The agent creates a new goal node in the TCN with an edge from the current, satisfied goal. The goal contains several different arguments (in this example `in`, e_{cup}, and $e_{garbage}$), and for each the agent needs to map them onto the task arguments.

Here the agent uses the heuristic that if an argument appears in both the task command and the goal, then it is mapped to the same slot. For example, e_{cup} appears in both the task command and the goal, so the corresponding arguments in the task and goal representations are mapped to the same slot. This is referred to as an *explicit argument*, since it is explicitly mentioned in the task command. This mapping is a key factor in the agent's ability to generalize to other task variations. If the same command is used with a different object (e.g., *"Discard the newspaper"*), the TCN shows the connection between the argument and the goals and subtasks. If a goal argument is not present in the task command (e.g. $e_{garbage}$), it is assumed to be fundamental to the goal. It is referred to as an *implicit argument*, and is added as a new slot with default values. These arguments will not be generalized, i.e., the goal of discard will always reference the garbage.

The agent then constructs a representation of the goal in working memory by matching the mapping information in the TCN against the world and task instance. Through this process, a new procedural rule is learned through chunking that can elaborate a goal representation. This process is described in Section 6.4.1

6.3.3 Task Decomposition:

To actually execute the task, the agent must decompose it into subtasks. Each subtask is either a previously learned task or a hand-coded task that the agent initially knows how to execute (such as `pick-up` or `go-to`). In the task problem space, the agent reaches a decision point where it must select the next subtask to execute given the current goal. If the agent lacks valid subtask proposals, it encounters a state no-change impasse. This presents the opportunity to learn a new subtask in one of two ways.

The first method that the agent tries to discover the next subtask is to perform a forward search. It creates an internal copy of the state and proposes all known task operators. It then does an iterative deepening search up to some cutoff depth, and simulates each operator using action models. If the search is successful, it returns the next subtask to perform. If it fails, the agent asks the instructor *'What should I do next?'* and the instructor can respond with the next step (e.g. *'Pick up the soda.'*).

Once the agent has a representation of the next subtask, it adds it to the TCN using the same method of mapping implicit/explicit arguments as with goals (see Figure 6.3). It then uses the TCN representation to construct a new subtask operator and propose it. Through this process it learns a new proposal rule (Section 6.4.2). This subtask proposal ends the impasse and the agent selects and executes it.

6.4 Procedural Knowledge

In order to fully incorporate the knowledge gained through a training interaction, the agent must compile it into procedural knowledge. This makes the agent much more efficient, as it summarizes the deliberate processing required to retrieve and analyze the declarative structures into rules. It also compiles the results from the forward search into policy rules. This all occurs through Soar's chunking mechanism that is based on explanation-based generalization. In the following sections, we describe how the goal and subtask knowledge in the TCN is operationalized and how policy rules, preconditions, and postconditions are learned. We provide summarized examples of the rules learned for each type of knowledge. As with the depicted representations, these abstract rules hide certain details for the sake of clarity. The actual Soar productions for each example in the chapter are shown in Appendix A.

6.4.1 Goal Elaboration

Once the agent has selected the next goal in the goal graph, it needs to elaborate a the goal onto the state. However, the entities in the TCN representation are represented as slots and default values and are not grounded to specific entities in the world. Through deliberate reasoning, the agent instantiates a copy of the goal and maps arguments to entities in the task structure (for explicit arguments) or the world (for implicit arguments). Using the discard example, it sees that the goal has a single relation predicate. The predicate name is implicit and defaults to `in`, the first entity maps to arg1 of the discard task operator (e_{cup}), and the second entity has no such mapping, but it matches the entity $e_{garbage}$ in the world through the default property `garbage`. The result of this deliberate processing is that the instantiated goal {in(e_{cup}, $e_{garbage}$)} is added to the state.

As a side effect of this processing, a rule is learned through Soar's chunking mechanism that incorporates the mapping logic in its conditions. This rule elaborates a grounded goal structure onto the state given the current task and goal handle. In the case of *discard*, it learns a rule summarized below. Note that this rule has variablized the argument, so it will apply to discarding any object.

```
IF:
  task=discard($e_i$) $\wedge$ current-goal=disc1goal1 $\wedge$ garbage($e_j$)
THEN:
  desired={ in($e_i$, $e_j$) }
```

6.4.2 Subtask Proposal

Subtask operator structures are instantiated in the same way as with goals. The agent matches the subtask representation in the TCN against the current task and world representations to variablize explicit arguments. However, with task operators there may be additional constraints that belong in the proposal rule, such as only proposing to open a door if it is closed. Therefore, once the subtask operator is created, it is matched against all possible tasks that are valid in the current state. If a match is found, its preconditions will be included in the learned chunk. In either case, the subtask operator is proposed within the parent task problem space which results in the agent learning a subtask proposal rule. In the case of discard, it would learn a proposal rule for the *pick-up* subtask summarized as:

```
IF:
  task=discard($e_i$) $\wedge$ holding-object=false
  $\wedge$ grabbable($e_i$) $\wedge$ not-grabbed($e_i$) $\wedge$ confirmed($e_i$)
THEN:
  propose pick-up($e_i$)
```

Note that this rule only represents the preconditions under which the subtask is valid. It does not include policy knowledge of when the subtask should be selected in order to achieve a particular goal. Learning this knowledge is covered next.

6.4.3 Task Policy

Once the task is successfully completed, the agent knows which subtasks are used in the parent task and has an example sequence of those subtasks, but it does not have a specific policy for *when* to do each subtask. It learns a policy mapping a state to the appropriate subtask $\Pi : S \rightarrow T$ through a retrospective analysis of the training example. Working backwards, it retrieves the state of the world when each subtask was proposed from episodic memory, including the parent task and current goal. It then internally simulates executing the subtask within the reconstructed state, and any subsequent subtasks, until it achieves the goal. It then elaborates a best preference to the subtask, which causes a chunk to be learned.

During the chunking analysis, Soar has access to a trace of rule firings that started with the subtask operator and ended with a satisfied goal. This serves as an explanation for why that particular subtask was useful in that context. By backtracing through the rule firings, Soar identifies exactly what portions of the state was required to generate the result, and these are pulled into the chunk preconditions. All other irrelevant parts of the state can be ignored. For example, in the discard task the agent retrieves the state of the world right before it proposed put-down(e_{cup}, in($e_{garbage}$)) and then simulates the operator using its action model. The result is the relation in(e_{cup}, $e_{garbage}$), which satisfies the goal. The learned policy rule is summarized as:

```
IF:
   task=discard($e_i$) $\wedge$ desired=in($e_i$, $e_j$)
THEN:
   prefer put-down($e_i$, in($e_j$))
```

6.4.4 Preconditions

The knowledge it learns from the above methods is sufficient for the agent to perform the task in the future. However, to truly learn hierarchical tasks, the agent must be able to use and plan with a learned task as a subtask for other, as yet, unseen goals. Thus it must learn the preconditions (when it can perform a task) and postconditions (what the task accomplishes), so in the future the subtask can be used in other higher-level tasks. Currently, the agent only learns these for tasks that involve a single goal node.

To learn the preconditions, the agent again retrieves the initial state when the task began, and simulates performing the task using the policy. If it succeeds in reaching the goal, it learns a rule to propose the task in a similar manner to learning the policy. The rule will incorporate the features of the state that had to be present in order for the entire task to succeed. For discard, this would be:

```
IF:
   grabbable($e_i$) $\wedge$ not-grabbed($e_i$) $\wedge$ confirmed($e_i$)
   $\wedge$ garbage($e_j$) $\wedge$ confirmed($e_j$)
THEN:
   propose discard($e_i$)
```

These rules tend to be overly-specific, since they are sensitive to the specific conditions in the initial state. For example, the above would not apply if the object was already grabbed, or the garbage was in a different location. This is usually not a problem, the only consequence is the instructor might have to teach alternative use cases.

6.4.5 Postconditions

The agent does not have sufficient domain knowledge to model all of the side effects of each subtask. For example, when going to the garbage, it cannot model which new objects may become newly visible or which objects are no longer visible. Thus, when the agent learns how the subtask changes the world (postconditions), it only includes what the agent *intends* to accomplish. It uses the learned representation of the goal as the postconditions. So in the case of discard, the action model would be to add the `in` predicate:

```
IF:
   execution-type=internal ∧ operator=discard($e_i$) ∧ garbage($e_j$)
THEN:
   +in($e_i$, $e_j$)
```

6.5 Discussion

In this chapter, we provided an overview of how Rosie represents, learns, and executes tasks. There are several core ideas that distinguish our approach. First, much of the task knowledge exists in dual representations – the declarative Task Concept Network and procedural rules. The TCN supports deliberate reasoning over the task structure and provides an easier way for the agent to insert and extend task knowledge. The learned rules support fast and efficient execution and allow new tasks to be used during planning.

Second, the representations support both the classical problem space and planning formulations (individual goal nodes) and procedural control structures (goal graph). Having both options gives the agent must greater flexibility, as shown in Chapter 8.

Third, the use of EBG allows for very aggressive generalization to learn rules that capture the argument structure in the TCN (allowing for explicit arguments to be variablized) and policy rules that only test the necessary aspects of the state. Together, these characteristics are the key to having incremental (**D2**), efficient (**D10**), and compositional(**14**) learning.

CHAPTER 7

Diverse Actions

When we consider the goal of having an interactive task learning agent be capable of learning a truly diverse and vast space of tasks, one major determining factor is what primitive actions are available to the agent. For one such as Rosie that builds up hierarchical tasks from its primitives, the agent is not able to learn tasks that contain features absent from the primitive actions. Even in systems that can learn new primitive actions, higher level tasks are still limited by the types of primitive actions that can be learned.

Human language presents an extensive diversity of action types. Consider the fifty most common English verbs (Davies, 2009), shown in Figure 7.1. Those that commonly describe some sort of action invoke physical manipulation (make, put, get), movement (go, turn, run), perception (see, find), communication (tell, ask), and mental state (think, know). It is likely that people will want collaborative agents to understand and use these types of actions. For example, consider having a robot ask someone a question and report back the answer. Most of the above action types appear in some form during the execution: finding that person, asking the question, remembering their answer, navigating back to the original person, and repeating the answer. However, many approaches to interactive task learning are demonstrated with tasks that primarily involve performing physical actions in the world, such as moving blocks, rotating tires, or making a smoothie. While there is certainly a large amount of research that needs to be done with these types of tasks, the other types of actions have been mostly overlooked in the context of ITL.

In this chapter, we describe the implementation of a set of innate tasks that cover a wide range of behaviors that include manipulation, movement, perceptual, communicative, and mental aspects. These innate tasks form the building blocks that the instructor can use to teach complex hierarchical tasks. A significant contribution is adding mental and communicative actions as proper tasks in a manner compatible with an explanation-based learning approach without requiring complete action models.

We also improve on previous work by unifying innate and learned task representations.

be	have	do	say	go	get	can	know	will	would
make	think	see	come	take	want	could	look	use	tell
find	give	need	should	work	try	let	call	may	mean
feel	ask	talk	keep	leave	put	like	help	start	become
happen	show	seem	might	hear	believe	play	turn	run	live

Table 7.1: The fifty most common English verbs in the Corpus of Contemporary American English.

This allows the agent to leverage the existing planning and learning capabilities to significantly improve the flexibility and adaptability of the agent within complex environments. In a complex open-world environment, the state of the agent and the world can vary wildly when the agent begins to perform a task. Consider the situation where the agent decides to pick up an apple. It may be in any number of different scenarios: the apple might be within arm's reach, across the room, inside the fridge, blocked by a cereal box in front of it, or in an unknown location. With an agent such as Rosie, in only a few of these situations will the motor system be able to immediately act upon a command to pick up an object. Otherwise, the agent needs to perform additional actions to carry out even simple-seeming tasks such as picking up a block or turning off a light. We show that by having a proper set of innate tasks, along with a consistent task representation and existing planning capabilities, Rosie is able to adapt to a wide range of such situations and learn strategies for successfully performing the desired tasks, even without complete models of its actions. This happens in real time as the agent is learning and executing other tasks, without requiring apriori programming or additional instruction.

In the following sections, we first discuss related work and identify the extent to which they support diverse types of actions in Section 7.1. Section 7.2 describes how innate tasks are represented within the agent, and Section 7.3 lists the innate tasks implemented for each type. We highlight the specific extensions required to support mental and communicative tasks in Section 7.4. Finally, we evaluate our method for representing and using innate tasks in Section 7.5 and provide a demonstration of the agent learning a task with all five types of actions in Section 7.6.

7.1 Related Work

None of the closest related approaches to learning from instructional ITL (Chapter 3) learn tasks that include all five types of actions. Several involve robots that only perform object manipulation actions and assume the world is fully observable (Mohseni-Kabir et al., 2019;

Frasca et al., 2018; Suddrey et al., 2016; Mohan & Laird, 2014). The approach by She & Chai (2017) does assume noisy/incomplete perception and produces plans that can include manipulation, navigation, and perception actions. Both PLOW (Allen et al., 2007) and SUGILITE (Li et al., 2020) operate in software environments, so manipulation actions involve interacting with elements in the current page and perceptual actions involve searching for an element matching some description, such as finding the text field naming an article's author or finding the temperature in a weather app. Meriçli et al. (2013) and Gemignani et al. (2015) utilize mobile robots that perform navigation tasks. Both include a *say* action to speak a fixed message, and can include visual landmarks within tasks. Additionally, both represent manipulation actions such as *pick-up*, but lack a physical arm and instead rely on a person to give/take an object.

While all of these approaches involve natural language instruction, only two can learn to say a message as part of a task, and none have the ability to ask a question within a learned task. None of these approaches formally represent mental operations as proper actions. One aspect that distinguishes this work is the ability to learn tasks from instruction that utilize communicative and mental actions within planning.

7.2 Innate Task Representations

Innate tasks are those that the agent starts with. We use this term instead of primitive action because these innate tasks can themselves be further decomposed into subtasks. In Rosie, *primitive action* refers to a command the agent can send to the motor system. In keeping with our research principle of using a unified approach, a crucial requirement of our work is that innate tasks share the same representations and execution framework as those learned through instruction. Therefore, innate tasks also have a task concept network and utilize planning and learning during execution. The major difference is that an innate task will have additional procedural knowledge to actually execute the task, whether that is sending a command to the output link, doing retrievals, or changing working memory. Otherwise, the agent has no ability to differentiate leaned and innate tasks Since there is no representational difference between learned and innate tasks, when executing a task, the agent has no way of telling if it is innate (i.e. there are no flags it can test for). This is a contribution of this dissertation, as previous work did have a distinction and did not involve any additional learning for innate tasks. As will be shown later, this unified approach leads to significant flexibility and adaptability of the agent. In the following sections, we describe the different types of knowledge required to implement innate tasks.

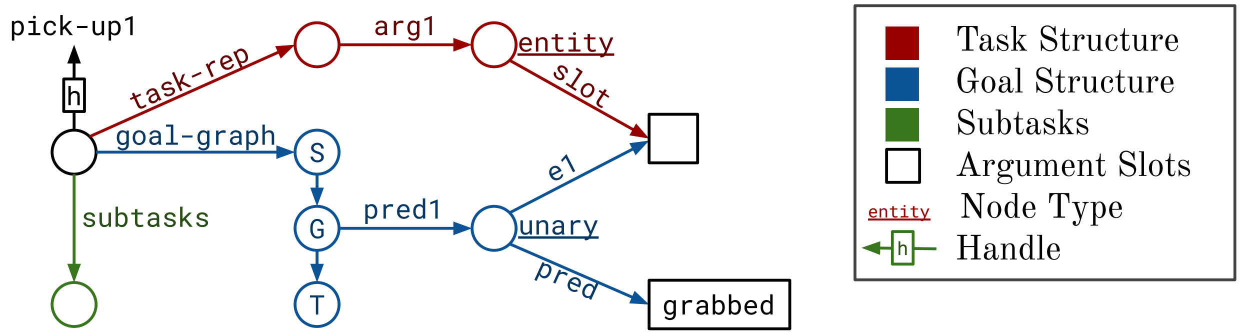


Figure 7.1: The task concept network for the innate task `pick-up`.

7.2.1 Task Concept Networks

Innate tasks share the same task concept network representations as learned tasks. These contain representations of both the task structure and the goal graph. A simplified example for the *pick-up* task can be seen in Figure 7.1. The `task-rep` link leads to the task structure, which in this case comprises a single entity argument `arg1`. The `goal-graph` has a start node (S), a single goal node (G) that represents `grabbed(`e_1`)`, and a terminal node (T). The structure of the TCN indicates that `arg1` and e_1 map to the same entity in the world, since they are linked to the same slot.

Most innate tasks have a single goal that falls into one of two categories. One is a predicate-based goal represented as achieving one or more goal predicates (e.g. `grabbed(`e_1`)` for *pick-up*). The other is to perform an action by sending a command to the motor system via the output link. The latter is used when the goal is to perform a specific motor command (e.g. turning left).

7.2.2 Procedural Knowledge

The main difference between innate and learned tasks is that innate tasks come with additional procedural knowledge that specifies how to execute the task. This procedural knowledge contains steps that cannot be learned, otherwise the task would be teachable through instruction. Most innate procedural knowledge falls into one of the following categories:

- **Semantic Memory Retrievals:** The agent retrieves additional information needed to perform the task from semantic memory. For example, if told to *"Orient north,"* the agent will retrieve concept of *north* to get the associated direction to face.
- **SVS Queries:** The agent extracts additional information about itself or the objects involved from SVS. For example, if told to *"Approach the table,"* the agent will use intersection queries to check several candidate positions around the table to see which

Proposal	not-holding-object(arm) $\wedge$ grabbable(e_i) $\wedge$ not-grabbed(e_i) $\wedge$ confirmed(e_i) $\rightarrow$ propose pick-up(e_i)
Action Model	-not-holding-object(arm) +holding-object(arm) -not-grabbed(e_i) +grabbed(e_i) -predicate$_p$(e_i, e_j) -predicate$_p$(e_k, e_i)

Table 7.2: The proposal conditions for the innate task `pick-up`, and the predicates that are added (+) or removed (-) by the action model.

are unobstructed.

- **Action Commands:** The agent creates the proper command on the output link to be sent to the motor system. This exact representation depends on the specific domain, so there is often a different rule for each embodiment. However, since the domains share a common world representation, this is often the only difference required to implement the same innate task across environments.
- **Subtask Proposals:** The agent breaks the innate task down even further into subtasks. For example, if told to *"Approach the counter,"* the agent will decompose that into *face* and *go-to-xy* subtasks.
- **Failure Handling:** The agent has knowledge to detect failures that occur in certain tasks and handle them appropriately. For example, in the tabletop domain if the agent is trying to pick up a block and drops it, the agent will know to reset the arm and try again.

7.2.3 Proposal and Model Knowledge

For some innate tasks, the agent has knowledge of how to model the task and use it during search. This has two parts: the proposal rule (preconditions) and action model (postconditions). An example of this knowledge for `pick-up` can be seen in Table 7.2. The proposal rule contains a set of conditions (predicates) that must exist in order for the innate task to be proposed. The action model contains a model of how the world changes as a result of performing the task by adding and removing predicates. A list of which tasks have this knowledge is shown in Figure 7.2. Often they are the tasks that have the goal of achieving some predicates in the world.

Category	Task	Prop	Model	Internal	Tabletop	Mobile
manipulation	close	X	X	X	X	X
	give	X	X	X		X
	open	X	X	X	X	X
	pick-up	X	X	X	X	X
	put-down	X	X	X	X	X
	pour	X	X	X		X
	press			X		X
	turn-on	X	X	X	X	X
	turn-off	X	X	X	X	X
	use	X	X	X		X
movement	approach	X	X	X		X
	drive					X
	face					X
	go-to-location	X	X	X		X
	go-to-next-wp		X	X		X
	go-to-wp			X		X
	go-to-xy					X
	orient					X
	stop					X
	turn					X
perceptual	explore			X		X
	find	X	X	X		X
	view	X	X			X
	scan					X
communication	ask		X*	ALL		
	describe			ALL		
	say	X	X	ALL		
mental	recall		X*	ALL		
	remember		X	ALL		
	wait			ALL		

Figure 7.2: A table of all the innate tasks in Rosie, grouped by category and marked with whether the agent has proposal rules or action models for each one and which domains each task is used in. An asterisk indicates that the action model is learned.

7.3 Innate Task Implementations

In total, we have implemented thirty innate tasks, shown in Figure 7.2. In keeping with our analysis above, we have grouped them into the following five categories:

- **Manipulation:** The primary purpose is to change the external environment in some way. Examples: *"Open the door"*, *"Pick up the mug"*.
- **Movement:** The primary purpose is to change the physical embodiment of the robot. Examples: *"Turn right"*, *"Go to the kitchen"*, *"Drive through the door"*.
- **Perceptual:** The primary purpose is to change the sensory information available to the robot. Examples: *"Find the red block"*, *"Scan the room"*.
- **Communicative:** The primary purpose is to exchange information with another agent through communication. Examples: *"Ask John 'What time is are you leaving?' "*, *"Say the meeting location to Ruth"*.
- **Mental:** The primary purpose is to change the agent's internal/mental state. Examples: *"Remember the answer"*, *"Remember the starting location"*, *"Recall the location when the stapler was visible"*.

We provide an overview of each of these categories below. Additional implementation details for each innate task are included in Appendix B.

7.3.1 Manipulation Tasks

Manipulation tasks involve interacting with the objects in the environment to achieve some desired state. Most have the goal of achieving some predicate, e.g., the goal of *close* is that the target is `not-open`. From a representational standpoint, these tasks are fairly simple as they have a one-to-one correspondence with an output command. However, as will be discussed in Section 7.5, those output commands have stricter preconditions than the task itself. This means that the agent will sometimes perform online search and policy learning within the innate task to perform subtasks in order to satisfy additional command constraints (e.g. driving closer to an object).

7.3.2 Movement Tasks

For mobile embodiments, Rosie has a number of innate tasks involving movement. Most of these do not have proposal rules or action models, as the agent lacks sufficient knowledge to predict precisely how a task such as *turn right* will change the world representation. These will not be proposed during planning, although they can be learned as part of a procedural task. The main exceptions are *approach*, which achieves the goal of the target being reachable, and *go-to-location*, which allows the agent to navigate between rooms. To implement navigation, the agent performs A* search over a known waypoint map, where 2D regions are each associated with a specific waypoint. The map has edges between adjacent waypoints, which are either marked as open or as having an intermediate doorway point.

7.3.3 Perceptual Tasks

For partially observable environments, every time the agent performs a task, the state of the world with respect to its perception may be different. Therefore, for the agent to be robust, it must be able to plan and reason with perceptual actions. In Rosie, there are two binary predicates that indicate perceptual status: visible (the entity is currently perceived) and confirmed (the entity has been seen before in the current room). Each predicate is associated with an innate task. The *view* task satisfies the visible predicate for a known object by turning to face it. The *find* task satisfies the confirmed predicate and is more complex.

Often the agent will have an internal representation of some physical entity but not know its location. This representation is sufficient to instantiate a goal or use during internal planning, but at some point the agent will actually need to perform an action involving that entity, such as picking it up. Therefore, the agent must *find* it in the world. There are many different strategies for finding an object that depend on the available knowledge. We have implemented a set of strategies and prioritize them according to the type of knowledge they use and the amount of effort they entail:

- **Short-term Memory:** The agent has knowledge about the location of the object in working memory and can act on that directly, for example, by driving to a different location or opening a receptacle.
- **Local Search:** The agent searches the local environment by performing a scan task (turning 360 degrees). If the entity becomes visible during the scan, the agent will stop early.
- **Long-term Memory:** The agent attempts to retrieve knowledge about an entity's possible location from its long-term memory through a *recall* subtask. There are three versions: recalling the location when the entity was last visible, a person's office, and an object's storage location.
- **Interaction:** If the agent has no knowledge about where to find the object, it asks *'I can't find the stapler, can you help?'* The instructor can describe the location of the object, tell the agent where to go, get the object themself, or decline to help.
- **Global Search:** If none of the above strategies work, the agent searches the entire environment by performing the *explore* task and searching every location.

7.3.4 Communicative Tasks

Adding communicative behavior enables the instructor to teach tasks that involve saying messages, asking questions, and basing behavior on responses. To support such tasks, we extend our category ontology to include messages. Thus abstract message type entities can be included in the world and task representations just as any other object. One challenge is determining the lifetime of these entities, as they are not connected to perception. Our approach is to keep any message entities in the world until the parent task in which they were added is completed. Another challenge that arose was how to represent a whole sentence within a command, e.g. *"Say 'The meeting is at 1:00.'."* Our solution is to treat the entire quoted string as a single value and not try to parse the contents. This has the benefit of being much simpler and allows the instructor to provide any sentence without requiring the parser to handle it or the agent to have general natural language generation capabilities. However, this does reduce flexibility as the sentence itself is fixed. Below, we describe the three innate communicative tasks (*say*, *describe*, and *ask*).

7.3.4.1 Say

The agent can perform the *say* task using a message entity with a known sentence and a target person. It sends an outgoing message on its output link to the chat interface and adds the relation `heard(`e_{msg}`, `e_{person}`)` to the world. This allows task goals such as *"The goal is that Bob heard the message."* Note that the only way for there to be a known sentence is if it was provided by the instructor. The agent cannot propose possible sentences to say on its own, but once the instructor gives a specific instance the agent can do search, reasoning, and planning with it.

7.3.4.2 Describe

The `describe` task is almost identical to the `say` task, except that the first argument is not a message entity but instead an object, person, or location. For example, the agent could *"Describe the current location."* or *"Describe the held object to Anthony."* The agent will generate a natural language description of the argument. `Describe` also adds the `heard` relation to the world, but does not have a proposal rule, as having the agent consider describing every object in the room during search quickly becomes intractable.

7.3.4.3 Ask

The other innate communicative task is `ask`, where the agent asks a question and then adds the response as an entity in the world along with the predicate `answered`. Rosie can handle three types of responses: a quoted sentence, the words *yes* or *no*, or a reference to an entity in the environment. The `answered` predicate allows following subtasks to refer back to the response, such as *"If the answer is yes, then ..."* or *"Pick up the answered drink."*

7.3.5 Mental Tasks

The last category of innate tasks in Rosie involves mental operations. Allowing the instructor to give explicit commands involving the agent's memories allows for much greater flexibility and range of learnable tasks, as the instructor can teach the agent how to use its memories instead of requiring that it already knows how to do so. We have divided the mental actions into two tasks: one involving storage to either working or semantic memory (`remember`) and one involving retrieval from either semantic or episodic memory (`recall`). A caveat is that while the instructions for these mental tasks do use natural language, they can be esoteric and require specific phrasing that an untrained instructor might not use. Future research in HRI would be useful to determine how people would naturally give instructions that involve mental operations.

7.3.5.1 Remember

The *remember* task is used to extend the agent's knowledge. We distinguish between two versions: short-term and long-term. The short-term version is the default and involves connecting two entities in working memory via the command *"Remember <e1> as <e2>."* For example, we say *"Remember the current location as the starting location."*, while teaching the task of fetching an object (it needs to remember where it started to return there later). The agent connects the abstract notion of a starting location with the current location. Then, the agent can later execute the command *"Go to the starting location."* and know where to go. Using `remember` in this way enables the agent to learn a task involving a concept such as the starting location without it being preprogrammed.

The long-term version of *remember* requires the adverb *permanently* and involves three arguments that end up defining an edge to create in semantic memory. For example, the instructor can say *"Permanently remember the fridge as the storage location of soda."* and the agent will create an edge labelled `storage location` from the concept of `soda` to the concept of `fridge`. There will then be a corresponding *recall* command that can retrieve this information, e.g., *"Recall the storage location of a soda."*

7.3.5.2 Recall

The `recall` task is used to retrieve knowledge from the agent's long-term memories. We identify the type of retrieval (semantic vs episodic) by checking if the task involves a *when* clause. If so, we assume an episodic memory retrieval, such as *"Recall the current location when Alice was visible."* To execute the task, the agent constructs an episodic memory cue from the when clause and retrieves an episode when it was last satisfied (e.g., find an episode containing an entity with predicates `alice(`e_i`)` $\wedge$ `visible(`e_i`)`). Then it tries to match the query argument from the command (`current(`e_j`)` $\wedge$ `location(`e_j`)`) and copy it into the world representation along with the predicate `recalled(`e_j`)`.

If the command does not have a when clause, we assume a semantic memory retrieval with two arguments, such as *"Recall the storage location of a soda."* The agent will search for an edge in semantic memory called `storage location` that starts at the concept of `soda`. If such an edge exists, the agent will copy the tail object or concept into working memory along with the `recalled` predicate. If it does not, the agent adds a new entity `{ nothing(`e_i`),` `recalled(`e_i`) }`. This allows us to teach a contingency plan that involves the phrase *"If nothing was recalled then ..."*.

7.4 Learning Mental and Communicative Subtasks

Extending the set of innate tasks to include those with mental and communicative aspects introduces additional challenges when learning tasks that use them. The previous approach to explanation based ITL made several assumptions about the innate tasks that are not satisfied by communicative and mental tasks.

First, the previous approach assumed that the agent has a priori knowledge of task proposal conditions that it can use during search and explanation. However, this is not practical for mental and communicative tasks as it would require the agent to propose and search over every possible thing that it could say or think. Including that sort of knowledge would require sophisticated models of language, the instructor, and its own mental structures, which is likely beyond the current state of the art to represent.

Similarly, the previous approach assumed that the agent has a priori action models for every innate task. Again this is not practical for mental and communicative tasks, as it would require an extensive model of language and memory. Another problem is that the outcome of these tasks is fundamentally determined by information that is not in working memory: long-term memory for the former and the instructor's mental state for the latter. For example, the agent would have no way to correctly predict the outcome of asking a yes

Instr: Store the ketchup.
Rosie: What is the goal?
Instr: The goal is that the ketchup is in the storage location.
Rosie: What do I do next?
Instr: Recall the storage location of the ketchup.
Rosie: [Executes `recall({storage, location}, ketchup)`]
Rosie: [Executes `open(fridge)`]
Rosie: [Executes `pick-up(ketchup)`]
Rosie: [Executes `put(ketchup, in(fridge))`]

Table 7.3: The interaction for teaching Rosie the task of storing an object. It includes performing a recall action.

or no question.

Third, the previous approach to retrospective policy learning assumed that if the agent retrieves the initial state and replays the sequence of subtasks, it will achieve the goal. But this breaks down when the agent lacks appropriate action models. Note that this is also an issue when operating in uncertain environments, as the agent lacks that ability to fully predict how an action will affect the perceptual state, and so its internal model will often diverge from what actually happened.

Thus the lack of a priori proposal rules and action models for mental and communicative innate tasks means that they are incompatible with the previous approach to explanation-based ITL as they cannot be used in search or to generate a valid explanation. A contribution of this dissertation is the development of a novel method for allowing the agent to utilize EBG when learning these types of subtasks, without requiring extensive pre-programmed task models or specialized knowledge. The key insight is that the agent can leverage the given instructions and its experience to learn models on a case-by-case basis. In the following sections we describe three extensions that address each of the above limitations. Throughout we will use the example of learning the *store* task. The dialog for this task is shown in Table 7.3.

7.4.1 Learning Proposal Rules

The first extension is to allow the agent to learn proposal rules for innate tasks when a novel usage is given through an instruction. When the agent asks for help and the instructor gives a new subtask command telling it what to do next, the agent checks to see if it needs to learn a new proposal rule for that subtask. It proposes all known subtasks and checks if any matches the one from the instructor. If there is not a match, then the agent will learn a new rule that proposes the given subtask but restricted to the parent task.

In the *store* example, Rosie learns the goal as `in(ketchup, storage-location)` but cannot connect the storage location to a physical entity and so cannot figure out what to do next. The instructor says *"Recall the storage location of the ketchup"* which provides a new subtask to perform. Rosie tries to match the recall action against all the valid subtasks, but does not find one. Through chunking it learns a new rule that will propose the recall subtask during store:

```
IF:
   current-task=store($e_i$)
THEN:
   propose recall({storage, location}, $e_i$)
```

Note that there are no additional constraints on the proposal rule, which can cause problems since it is always valid (the agent can always do a retrieval action). Therefore, for these types of tasks, we add a restriction that the agent can only do them once.

7.4.2 Learning Action Models

The second extension is to have the agent learn specific action models for certain innate tasks. The challenge is that the result of certain tasks depends on knowledge external to working memory. For Rosie, this happens with the *ask* and *recall* tasks. The result of asking a question depends on the mental state of the instructor, and thus there is nothing in working memory that can predict the result. Similarly, the result of a recall action depends on knowledge in long-term memory. This presents a problem for search and EBG, as their success depends on being able to model how a certain action helps achieve the goal.

Our insight is that the agent does not require a model that unerringly predicts the outcome, but instead only requires a model that produces a valid result for the specific usage. In the context of our *store* example, the agent does not need a model that always adds the correct storage location from the recall action. Instead, it only needs a model that adds *a* storage location that has all the necessary properties. Even this relaxed requirement on the action models is not feasible to preprogram, as it would involve adding comprehensive models of language and memory. But the second insight is that the agent can learn these action models from its experience.

The agent learns a new action model during the retrospective policy learning. During this learning stage, the agent simulates the execution of each subtask and ideally this will modify the state in a way that brings it closer to the goal. In our *store* example, the agent will try and model the *recall* subtask. However, it lacks the knowledge of how this subtask will change the state and encounters an impasse. This provides the opportunity to learn a new action model rule. The agent does an episodic memory retrieval for the past episode that contained the result (for the store task, this would be that it recalled the fridge). It adds the retrieved object to the superstate and this result will cause the agent to learn an action model chunk:

```
IF:
   current-task=store($e_i$) $\wedge$ execution-type=internal
   operator=recall({storage, location}, $e_i$)
THEN:
   +entity{fridge, storage, location, recalled, visible, ...}
```

The execution type signifies that it is doing internal simulation rather than actual execution. Note that the rule adds a very specific instance of the fridge to the state. If the agent was instead reasoning about how to store a mug, the agent will still model storing it in the fridge, even if its semantic knowledge would indicate the cupboard. This presents a problem for the previous approach to retrospective learning, as the simulated execution will diverge from the actual experience. The solution is to learn the policy incrementally, as will be described in the following section.

7.4.3 Incremental Policy Learning

The previous approach to retrospective policy learning required action models accurate enough to completely model a task from start to finish. This conflicts with learning a fixed action model for a particular subtask. For example, if reasoning about storing a mug, the action model that adds the fridge as the storage location will cause a conflict with the agent's actual experience of putting the mug in the cupboard. To address this, we modified the retrospective policy learning to occur incrementally. The agent only tries to learn the policy for one subtask at a time, and therefore there is no issue with having the internal model diverge from the historical experience in subsequent subtasks.

An example of how each approach works for the store task is shown in Figure 7.3. The batch policy learning approach retrieves the initial state S_1 from episodic memory and each subtask T_i, and then models applying each subtask to the state until the goal is reached. When this occurs, the agent can generate an explanation of the whole task and learn policy rules for each subtask. Our new incremental policy learning approach learns each subtask policy individually. Starting at the final subtask T_3 (putting the ketchup into the fridge) it retrieves the state when that subtask began S_3. Applying T_3 achieves the goal, so a rule is learned. It then goes back and simulates subtask T_2 in the retrieved state S_2. Note that it takes two steps to reach the goal, and the second one is selected using the learned policy instead of an episodic memory retrieval. This is why it starts with the final subtask, as earlier subtasks need the learned policy knowledge for later ones. Through this process, it learns policy rules such as:

```
IF:
   current-task=store($e_i$) $\wedge$ storage-location($e_j$) $\wedge$
   not-grabbed($e_i$) $\wedge$ proposed-subtask($t_k$)=pick-up($e_i$)
THEN:
   prefer $t_k$
```

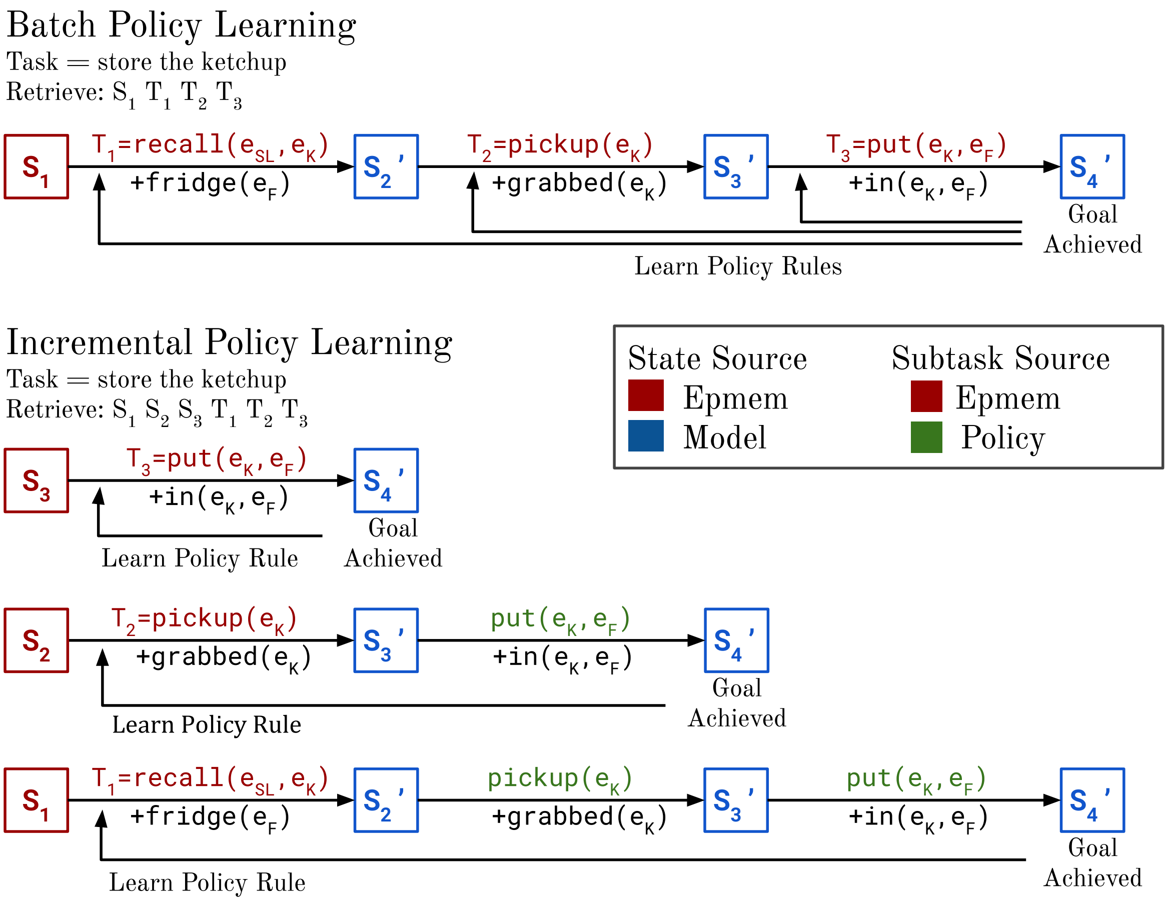


Figure 7.3: A comparison between the batch approach to retrospective policy learning and our incremental approach for the task of storing the ketchup. The former tries to learn everything in one pass, while the latter learns each subtask separately. The entity e_K is the ketchup, e_F is the fridge, and e_{SL} is the storage location.

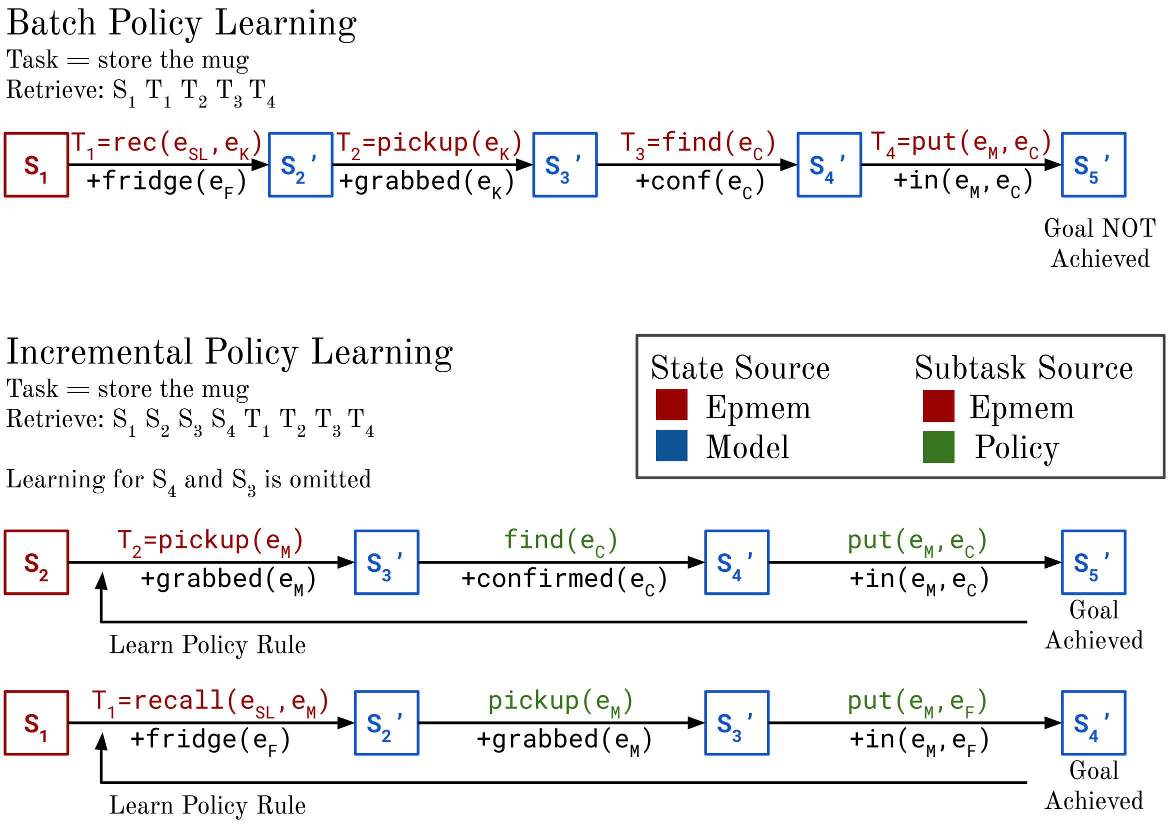


Figure 7.4: Learning the policy for storing a mug (e_M) in the cupboard (e_C) after already learning to store the ketchup. Note that the storage location in S_2 is the cupboard, and the storage location in S_2' is the fridge. Thus the subtasks diverge, but both paths are successful.

The need for an incremental approach can be seen in Figure 7.4 where the agent needs to do additional policy learning after storing a mug (in the cupboard). This time, the agent needed to add the step of finding the cupboard. The batch policy learning fails because the action model for recall says the mug is stored in the fridge, whereas the agent's experience involved placing it in the cupboard. However, the incremental approach can handle this. When starting at state S_2, the storage location is already marked as the cupboard, so the step of finding the cupboard helps achieve the goal. When starting at state S_1, the recall action model adds the fridge, and the subsequence subtasks selected using the policy also achieve the goal. Note that the subtasks diverge from S_1 and S_2, but both paths generate a valid explanation.

This incremental approach has a number of other advantages. First, it reduces the need for highly accurate models of how the agent's actions change the world state. The historical execution sequence can also diverge from the simulated sequence if the environment changed in ways that were not captured by the models. For example, suppose that the agent thought an object was in one location, but it was somewhere else. It would have to take extra actions to find the object that would not be expected given the initial state. The extra actions would not be helpful when generating the explanation in the batch approach and would be ignored. But the incremental approach includes retrieving all the intermediate states where the unexpected actions were needed and learns policies for these contingencies.

7.5 Innate Task Policy Learning

A key claim of our work is that using a unified approach, where innate tasks share the same representations and learning mechanisms as learned tasks, leads to greater adaptability in complex environments without needing further programming or instruction. It improves the ability of the agent to generalize its learned task knowledge to more situations. To realize this benefit, innate tasks conceal lower-level implementation during planning by presenting simpler proposal conditions and action models than are involved with the actual low-level execution. Thus, the proposal conditions for an innate task are less restrictive than a corresponding motor command. For example, the `pick-up` motor command requires that the target object is `visible` and `reachable`, whereas the `pick-up` task preconditions only require that the target object is `confirmed` (previously seen). As a result, when the agent begins one of these innate tasks, it may not be able to immediately perform the desired motor action. It utilizes the same task planning capabilities as described earlier to satisfy those predicates, which leads to learning additional task knowledge for performing pick-up in a wider range of situations.

To demonstrate the benefits of this unified approach and highlight the agent's generalization capabilities, we performed an evaluation that required the agent to learn a new task given a single language instruction and apply it in a wide range of starting conditions in a changing environment. Along the way, the agent sometimes had incomplete or incorrect knowledge. Its planning capabilities allowed it to overcome these obstacles and achieve a high level of success. We also evaluated the claim that our approach improves the agent's generality when compared to a version without the ability to plan within innate tasks.

7.5.1 Experimental Design

We performed the experiment using the Magicbot simulator. The test environment, shown in Figure 7.5, consisted of a single room with six objects and six places to put them: two surfaces (table, counter), two receptacles (sink, garbage), and two receptacles that can be closed (fridge, pantry). To demonstrate the ability of the agent to generalize a single task instruction to a wide range of different contexts, we had the agent perform a series of 100 randomly generated move commands, where the object being moved and the destination are randomly chosen. After the move is complete, we changed the environment by relocating a random object. We also closed one of the two receptacle doors with a 33% chance. This meant the agent could build up a perfect map of the environment and

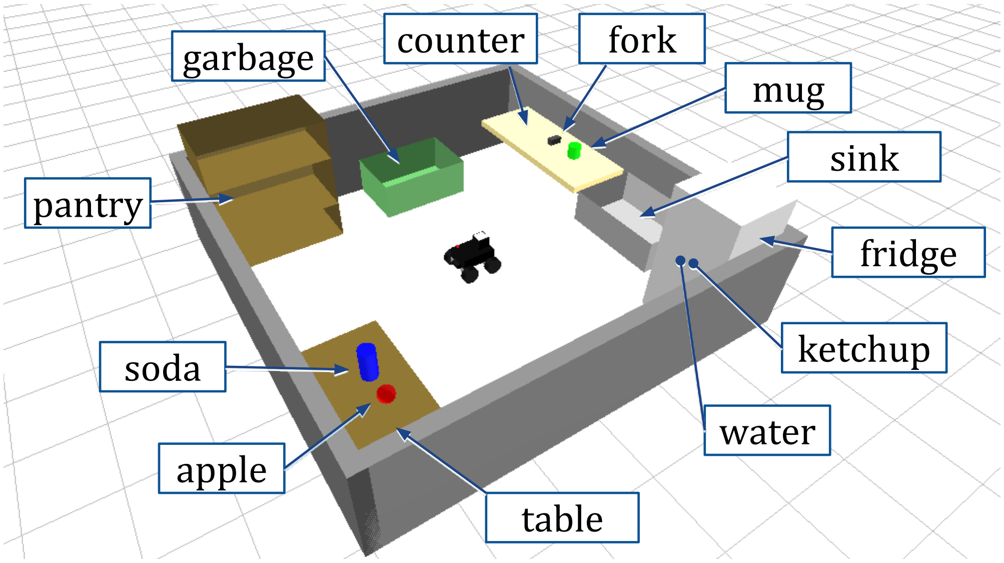


Figure 7.5: The simulated kitchen environment used for the move tasks.

sometimes needed to search for an object. If it ever asked for help finding an object, we described its location (e.g. *"The apple is in the fridge."*).

For the trial of 100 move tasks, we recorded the full subtask trace (task decomposition) for each one. The expectation was that even for this simple task, the variation of the initial conditions in the environment and the incorrect beliefs about where objects are located will result in a number of different task decompositions. Having many unique decompositions indicates there was significant variation between tasks and demonstrates that the agent generalized from a single training example to all of those variations.

We also compared two variations of the agent that only differed in the innate task preconditions. The *normal* version used the technique described above where the preconditions are less restrictive than those required for the lower-level motor commands. In this version, additional search and policy learning was often necessary within innate tasks. The *strict* version had additional preconditions for innate tasks so they could only be chosen if the agent could immediately perform the associated motor command. Usually, the normal version only required that objects be confirmed (previously seen), whereas the strict version also required that they are visible and within reach. The expectation was that planning for the strict version of the agent will be more difficult, as it has to be done at the top level instead of divided between the move task and its subtasks. To measure the planning difficulty, we recorded the time and number of decision cycles spent in search for each task, and the number of rules learned for each task. If planning was more difficult for the strict agent, it should spend longer searching, and if the learned policies were less general, it should have needed to search more often.

Measure	**Normal**	**Strict**
Total runtime (min)	36.26	37.10
Successful tasks	97	97
Unique task decompositions	29	29
Number of searches	25	56
Max search time for 1 task (sec)	0.98	17.26
Number of chunks learned	177	271

Table 7.4: The statistics of the two agent versions doing 100 random move tasks.

7.5.2 Experimental Results

A summary of the results are shown in Table 7.4. Both versions of the agent took around 37 minutes to complete the one hundred move tasks and successfully completed 97 of them without requiring any further task instruction beyond the initial goal description. The three failures were the result of perceptual issues where the object disappeared from sight after a put-down command (simulator placed it outside the view region). Across the one hundred tasks, the agent ended up achieving the goal through 29 unique task decompositions, showing that there was a significant amount of variation among the tasks. In one scenario, the agent thought the fridge was open and picked up the target object, turned around, then saw that the fridge was closed. It had to re-plan and put down the object, open the fridge, then pick it back up. Completing this one task involved 26 steps. In other scenarios, the object had been moved from where the agent thought it was, also requiring the agent to re-plan. In both scenarios, it was able to recover and accomplish the task without requesting help from the instructor. These results demonstrate the ability of agent to generalize from a single teaching example.

When we compare the two versions of the agent, the results show that there is no difference in the final task outcomes, both are able to handle the different variations and successfully perform 97 of the tasks. However, there are significant differences in the planning statistics. The strict agent needs to search for the next subtask over twice as many times as the normal agent (56 vs 25). It ends up learning more policy rules as well (271 vs 177). These data indicate that the chunks learned by the strict agent are less general. In addition, the maximum search time for the strict agent is an order of magnitude greater than for the normal agent (17.26 vs 0.98 seconds). This shows that the planning problem is more difficult for the strict agent.

To further illustrate that point, we graph the number of decision cycles spent performing search for each of the 100 move tasks (Figure 7.6). The tasks are temporally ordered on the x-axis (first task is on the left, last task is on the right). If the agent does not spend any cycles performing search for a given task, it indicates that the previously learned task knowledge is sufficient to execute that instance. The left graph shows that for the agent with normal preconditions, the knowledge learned during the first fifteen move tasks was general enough to cover the vast majority of subsequent tasks. It only needed to perform search in novel scenarios three times in the final 85 tasks. In contrast,

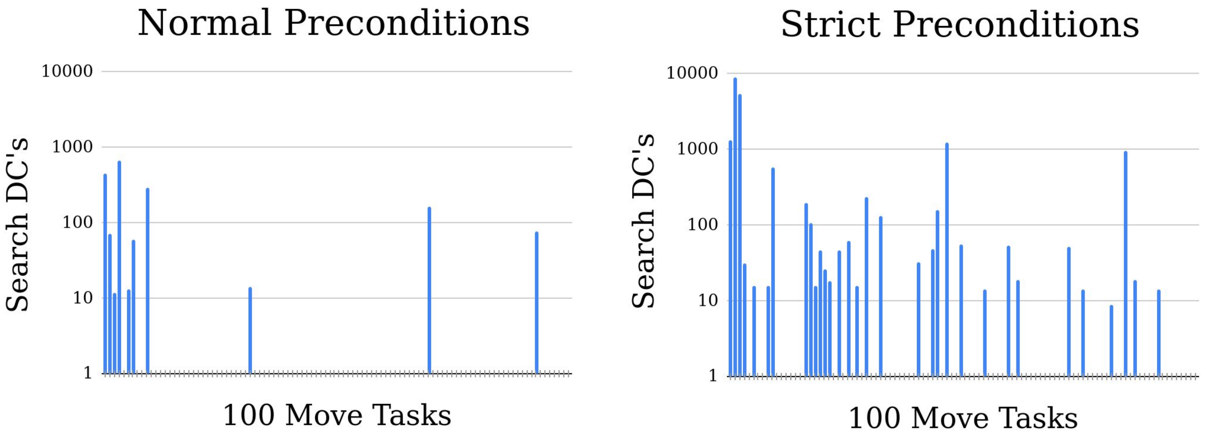


Figure 7.6: The number of decision cycles spent in search during each move task for the agent with normal preconditions (left) and the agent with strict preconditions (right).

the agent with strict preconditions needed to perform search much more often. In addition, the maximum number of decision cycles spent doing search for a given task is an order of magnitude greater.

These results show that the planning problem is much more difficult for the agent with strict preconditions, and that the knowledge it learns is less general. This is due to it having to solve the complete planning problem at a single level. The normal agent could instead perform high-level planning at one level and lower-level planning inside the innate tasks. For example, given the task `move(mug, in(sink))`, at the top level the normal agent would break it into two steps: `pick-up(mug)` and `put(mug, in(sink))`. The strict agent would break it into 6 steps: `view(mug)`, `approach(mug)`, `pick-up(mug)`, `view(sink)`, `approach(sink)`, and `put(mug, in(sink))`. This latter plan is more specific and will transfer to fewer situations. It also means the strict agent needs to search at a greater plan depth, which explains the greater search times. This is a problem, given that we want the agent to be reactive and responsive.

This experiment demonstrates the benefits of our approach of using more abstract preconditions for innate tasks and allowing the agent to perform additional search within them. It makes the planning problem easier and the learned policy knowledge more general. Another advantage of our approach is increased transfer to new tasks. For the strict agent, all the learned policy knowledge is tied to the move task. If we were to teach it a different task, it would not be able to re-use that policy knowledge. In contrast, since the normal agent learns some of that policy knowledge within the innate tasks, that knowledge is available whenever that innate task is used.

Dialog	Learning	Execution
Serve Mary.	initial TCN	
What is the next goal or subtask of serve?		
The goal is that Mary is holding the desired drink.	goal$_1$	
What do I do next for serve?		
Recall the preferred drink of Mary.	subtask recall$_1$	$\texttt{recall}_1(\texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Mary}})$
What do I do next for serve?		
If there is a preferred drink then ask Mary the preferred drink.	subtask ask$_1$	
Blue soda?		$\texttt{ask}_1(\texttt{e}_{\texttt{Mary}}, \texttt{e}_{\texttt{soda}})$
Yes.		
What do I do next for serve?		
If the answer is yes then remember the preferred drink as the desired drink.	subtask remember$_1$	$\texttt{rem}_1(\texttt{e}_{\texttt{soda}}, \texttt{e}_{\texttt{des}})$
	subtask pick-up$_1$	$\texttt{pick-up}_1(\texttt{e}_{\texttt{soda}})$
	subtask give$_1$	$\texttt{give}_1(\texttt{e}_{\texttt{soda}}, \texttt{e}_{\texttt{Mary}})$
What is the next goal or subtask of serve?		
Permanently remember the desired drink as the preferred drink of Mary.	subtask remember$_2$ goal$_2$	$\texttt{rem}_2(\texttt{e}_{\texttt{soda}}, \texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Mary}})$
What is the next goal or subtask of serve?		

You are done.	policy {remember$_2$ give$_1$ pick-up$_1$ remember$_1$ ask$_1$ recall$_1$}
	action-model {ask$_1$ recall$_1$}	

Figure 7.7: The steps of learning the serve task for the first time. In this version, the agent knows Mary's drink preference and she confirms it.

7.6 Learning Demonstration - Serve

To conclude this chapter, we present a demonstration of Rosie learning the task of serving someone a drink. Rosie will first check if it already knows their drink preference and confirm it is what the person wants. Otherwise it will ask the person what drink they want. After giving them the drink, it will store their preference in semantic memory. This task was chosen because it includes all five types of subtasks and requires all of the novel task learning extensions that were discussed in this chapter. It also highlights the use of mental and communicative subtasks and the ability to use them during planning and transfer to later variations in the task without needing additional instruction.

7.6.1 Serve with a Confirmed Preference

The first training interaction is shown in Figure 7.7. It shows the dialog, learned knowledge about serve, and the subtasks performed. Further learning and execution happens within those subtasks, but are omitted for brevity. In this interaction, Rosie already knows Mary's drink preference (soda) and confirms that is what she wants. Note the distinction between the preferred drink (the preference recorded in semantic memory) and the desired drink (the actual drink that the person desires in this instance). First, the agent is given the goal description (*"The goal is that Mary is holding the desired drink."*) and it learns both declarative and procedural knowledge of the goal. However, the word *'desired'* has no special meaning and the agent fails to find a plan to achieve the goal. The instructor tells it to *"Recall the preferred drink of Mary."* and it successfully recalls a soda and adds it to the state with the modifier *'preferred'*. The agent is then instructed to confirm that drink with Mary, and then *"If the answer is yes, then remember the preferred drink as the desired drink."* This last instruction will cause the agent to merge the two entities and result in one having the predicates `{desired, preferred, drink, soda}`.

This is enough information to allow the agent to plan out the remainder of the task on its own and pick up the soda and give it to Mary. The goal is achieved, and the agent asks if there is another step in the serve task. The final instruction is to *"Permanently remember the desired drink as the preferred drink of Mary,"* thus storing it for future use. Note that this subtask is not done in the context of a goal, so the agent creates a second procedural goal of performing the remember subtask. This process of learning procedural goals is covered in chapter 8. Here it signifies that after the agent achieves the primary goal, it should do this extra step of remembering the preference.

Once the task is finished, the agent does the retrospective policy learning. During this process, it learns action models for *recall* and *ask*. For the former, it learns that recalling the preferred drink will result in retrieving a soda, and for the latter it learns that asking to confirm the preferred drink will result in a yes. These models allow it to generate a complete explanation for each subtask and results in a policy rule, though only within the serve task.

7.6.2 Serve without a Known Preference

In the second teaching interaction (Figure 7.8) the agent is told to serve a different person whose drink preference it does not know. Furthermore, both the person being served and the drink are in different locations and Rosie will have to adapt. The agent cannot immediately perform the *recall* subtask since the policy rule it learned included the condition that the person was confirmed. However, because the agent has perceptual innate tasks it reasons that if it finds Alice then it can continue. Once it finds her, it tries to recall her drink preference but this step fails. The instructor then says *"If nothing was recalled then ask Alice 'What drink would you like?'"* Alice responds with *"water"* and it is added to the state along with the predicate `answered`. Through another *remember* instruction, the agent merges the answer with the desired drink and carries out the remainder of the task, including additional find subtasks. At the end it stores water as Alice's preferred drink in semantic memory.

Dialog	Learning	Execution
Serve Alice.		
	subtask find$_1$	$\texttt{find}_1(\texttt{e}_{\texttt{Alice}})$
		$\texttt{recall}_1(\texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Alice}})$
What do I do next for serve?		
If nothing was recalled then ask Alice "What drink would you like?".	subtask ask$_2$	
What drink would you like?		$\texttt{ask}_2(\texttt{e}_{\texttt{Alice}}, \texttt{e}_{\texttt{sent}})$
A water.		
What do I do next for serve?		
Remember the answered drink as the desired drink.	subtask remember$_3$	$\texttt{rem}_3(\texttt{e}_{\texttt{ans}}, \texttt{e}_{\texttt{des}})$
	subtask find$_2$	$\texttt{find}_2(\texttt{e}_{\texttt{water}})$
		$\texttt{pick-up}_1(\texttt{e}_{\texttt{water}})$
	subtask find$_3$	$\texttt{find}_3(\texttt{e}_{\texttt{Alice}})$
		$\texttt{give}_1(\texttt{e}_{\texttt{water}}, \texttt{e}_{\texttt{Alice}})$
		$\texttt{rem}_2(\texttt{e}_{\texttt{water}}, \texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Mary}})$
	policy {find$_3$ pick-up$_3$ find$_2$ remember$_3$ ask$_2$ find$_1$} action-model ask$_2$	

Figure 7.8: The steps of learning the serve task for the second time. In this version, the agent does not know Alice's drink preference.

7.6.3 Serve with an Alternate Request

In the third teaching interaction (Figure 7.9), the agent successfully recalls Bob's drink preference but he says no when it asks him. The instructor then says *"If the answer is no, then ask Bob 'What drink would you like?'"* This is the only additional instruction needed, the rest of the task is completed using previously learned task knowledge.

7.6.4 Discussion

These three interactions show the impressive ability of the agent to generalize learned task knowledge to novel circumstances and extend it when necessary. In tasks 2 and 3, the people being served, the drinks, their locations, and several subtasks are different and yet the agent seamlessly transfers the applicable of the task knowledge. The only help it needs is when the task fundamentally differs from previous examples. Even then, when the previous policy knowledge becomes applicable it continues on with the task. This is a major advantage of using explanation-based learning with goal-based tasks, as the agent knows why each subtask is helpful and when to use it. If instead the agent did not have a goal and was simply learning a procedure, it would not know what to do after encountering a novel branching path. It would also lack the flexibility to come up with alternative plans.

Dialog	Learning	Execution
Serve Bob.		
		$\texttt{find}_1(\texttt{e}_{\texttt{Bob}})$
		$\texttt{recall}_1(\texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Bob}})$
Water?		$\texttt{ask}_2(\texttt{e}_{\texttt{Bob}}, \texttt{e}_{\texttt{water}})$
No.		
What do I do next for serve?		
If the answer is no then ask Bob "What drink would you like?".	subtask ask_3	
What drink would you like?		$\texttt{ask}_3(\texttt{e}_{\texttt{Bob}}, \texttt{e}_{\texttt{sent}})$
Orange juice.		
		$\texttt{rem}_3(\texttt{e}_{\texttt{ans}}, \texttt{e}_{\texttt{des}})$
		$\texttt{find}_2(\texttt{e}_{\texttt{juice}})$
		$\texttt{pick-up}_1(\texttt{e}_{\texttt{juice}})$
		$\texttt{find}_3(\texttt{e}_{\texttt{Bob}})$
		$\texttt{give}_1(\texttt{e}_{\texttt{juice}}, \texttt{e}_{\texttt{Bob}})$
		$\texttt{rem}_2(\texttt{e}_{\texttt{juice}}, \texttt{e}_{\texttt{pref}}, \texttt{e}_{\texttt{Bob}})$
	policy $\{\text{ask}_3\}$	
	action-model ask_3	

Figure 7.9: The steps of learning the serve task for the third time. In this version, the agent knows Bob's drink preference but he wants something else.

This task also highlights the ability of our agent to use mental and communicative subtasks during task learning, which distinguishes this work from other related task learning agents, especially considering their integration with reasoning and planning. This usage is only possible because of the extensions made to learning proposal rules and action models for these innate tasks. Even though the agent ends up learning overly specific action models, they are sufficient for the policy learning.

Following the three teaching interactions, we verified its performance by testing it on three additional serve tasks with different people, drinks, and locations. Rosie successfully performed these tasks without needing further instruction. Furthermore, the agent did not need to perform additional search or policy learning for serve, as the rules that had been learned were sufficient. This is mostly due to the higher level of abstraction provided by the innate tasks. The agent ended up learning several variations of the *pick-up*, *find*, and *give* subtasks that dealt with the environmental differences between tasks.

There are some notable areas of further improvement and future work. An obvious one is the tight restrictions on how this task can be taught. Although the instructions are in natural language, they are very specific and sometimes esoteric, especially those involving mental actions. An untrained instructor is very unlikely to use the specific phrasings required. There are also times where the agent asks for help and the instructor needs to have a detailed model of the agent to know what to say. For example, in task 2 the agent finds Alice and then the *recall* action fails. The instructor gives the correct instruction even though the agent did not directly communicate the problem. Further research on human-robot interaction is needed to examine how people might naturally want to give these types of instructions and how the robot can communicate the appropriate information when the instructor lacks a detailed mental model of the agent's knowledge.

Even with our improvements to the retrospective policy learning, it is still somewhat brittle. The agent requires a lot of domain knowledge to generate a sufficient explanation trace. If it lacks an action model, its ability to learn a policy is limited. We show a way to fall back on learning a procedure if this occurs (8.4.2), but it is not a complete fix. An interesting area of future work is adding the ability to learn action models from instruction.

CHAPTER 8

Diverse Task Formulations

Another challenge that arises when developing an agent that can learn a truly diverse space of tasks is that no single representation, learning mechanism, or method of execution is likely to be optimal for all tasks. Therefore a truly general agent should be able to formulate a task in different ways, depending on the characteristics of the task and how it was taught. A task *formulation* includes which learning methods are used, what knowledge is learned and how it is structured, and how the agent uses this knowledge to perform the task in the future. Tasks can be formulated in different ways, for example, achieving a goal, following a procedure, or maximizing an objective function. These different formulations each have their own trade-offs in generality, ease of instruction, and ease of learning. In addition, a task can combine elements of these different formulations, such as driving to a destination (a goal) as quickly as possible (maximization).

Often the knowledge provided by the instructor suggests an appropriate formulation. For one task the instructor may prefer to describe the goal of the task and have the agent figure out how to achieve it. For another task the instructor may find it difficult to describe the goal and so instead provide a specific procedure for the agent to follow. Having the option of using different formulations for a task gives the instructor greater flexibility in teaching method. In addition, the capabilities and knowledge of the agent can also have an impact on which formulation the agent chooses to use to represent a task. For example, if the instructor did describe the goal of a task, but the agent lacked sufficient knowledge and planning capabilities to figure out how to achieve the goal, it could instead decide to learn the task as a procedure. Or the agent could be given a procedure and try to infer the goal by analyzing how the actions affected the state.

Most approaches to interactive task learning only represent tasks using a single type of formulation. The nine contemporary approaches identified in Chapter 3 either represent tasks as achieving a goal, or following a procedure, but never both. Only two, the previous work (Mohan et al., 2012) and She & Chai (2017) use goal formulations. The latter uses a hypothesis space over possible goals, so it is more flexible than our approach. Both use planning to enable the agent to adapt to variations in the task. The other approaches represent a task using some sort of procedure. Frasca et al. (2018) is limited to tasks with a linear sequence of steps. Mohseni-Kabir et al. (2019) use a hierarchical task network. We consider HTN's to be procedural task formulations, as each task consists of executing a set of possibly ordered subtasks. Suddrey et al. (2016) also use HTN's, but include the ability to plan to when subtask preconditions are unmet. We demonstrate a similar capability in Section 8.4.1. Others represent tasks through a structured process that can be executed (Allen et al., 2007; Li et al., 2020). Both Meriçli et al. (2013) and Gemignani et al. (2015) learn tasks as procedural graph structures, which are very similar to our goal graph. However, we treat each node as a problem space, which allows for much greater flexibility.

A major contribution of this dissertation is developing a goal graph representation that can represent procedural tasks, while utilizing the problem-space formulation for each individual subgoal. This hybrid approach supports learning both goal-based and procedural tasks, and tasks that blend the two, thus significantly expanding the space of tasks that Rosie can learn. Furthermore, since we use a unified representation for both types of formulations, the agent can easily compose tasks of different types within the same hierarchy. Another benefit of our unified approach is that there are ways that the agent can learn tasks that blend the two formulations. Goal-based tasks and procedural tasks have complimentary strengths and weaknesses, so blending the two leads to greater flexibility in certain cases.

In this chapter, we analyze characteristics of goal-based and procedural formulations (Sections 8.1 and 8.2) and describe how they are represented and learned. We also discuss optimization formulations and possible extensions that could support them (Section 8.3). Finally, we present and analyze three demonstrations that show the agent learning tasks that compose and blend different formulations (Section 8.4) and end with a short summary (Section 8.5).

8.1 Goal-Based Formulations

In a goal-based task formulation, the agent has a declarative representation of the task's goal and the task is complete once the goal is achieved. The goal could be explicitly taught by the instructor, or learned by the agent through experience. Like the innate tasks in the previous chapter, goals should not be limited to achieving some physical state in the world, but should include communicative, informational, and perceptual objectives. Once the agent has learned the goal, it can accomplish the task by performing actions to try to achieve that goal. The agent could use methods such as forward search, means end analysis, or reinforcement learning to determine the next action.

A goal-based formulation has a number of benefits. It can be efficient to teach if the agent has the ability to derive and learn the policy on its own, as the instructor only has to specify the goal and not how to achieve it. It also makes the agent more robust to variations in the task as it can find different ways of achieving the goal given different initial conditions, improving generality and transfer. Generally a state-based policy is more flexible than a fixed procedure when unexpected cases arise. However, this flexibility comes at the cost of requiring more domain knowledge or training time. An agent such as Rosie that relies on explanation-based generalization (EBG) requires action models that are sufficient to model how its actions change the state of the world and ultimately achieve the goal (the same action models that are required for planning). Although EBG is a powerful technique for learning a policy, it fails to learn anything if certain action models are missing. For some tasks, searching for a plan to achieve the goal might be prohibitively difficult. For example, baking a cake has a clear goal, but coming up with a sequence of actions to produce one through planning would be extremely hard. Other tasks may not be easily described as achieving a goal apart from performing some specific actions, such as patrolling a building. It may be easier to teach them using a procedure instead.

We represent a goal-based task as a single subgoal node within the goal graph. This node contains the declarative representation of the goal. The task, current state, and goal description define a proper problem-space. The agent executes a sequence of subtasks to reach a terminal state, aided by planning. In the following sections, we identify three variations on a goal-based formulation and describe the extent to which Rosie supports them.

8.1.1 Satisfaction Goals

This is the formulation that was originally implemented by previous work (Mohan & Laird, 2014), where the agent has a representation of the goal and the task is considered finished when the goal is satisfied. The process of learning and executing a standard goal-based task has been covered previously in Chapter 6, so we will only give a brief summary. For a goal-based task, the instructor provides a natural language description of the goal (such as *'The goal is that the cup is in the garbage.'*) which is transformed into a set of desired state predicates: `{in(`e_{cup}`, `$e_{garbage}$`)}`. The agent stores a parameterized declarative representation in semantic memory, and learns a procedural rule that elaborates the goal onto the state. The agent uses planning to search for a next subtask that will help achieve the goal. If that fails, it requests additional instructions from the instructor. Once finished with the task, it learns a state-based policy through EBG by simulating the subtasks and generating an explanation for how they help achieve the goal. For this to work, the agent must have sufficient action models that can simulate the effects of subtasks and model how the subtasks achieved the goal.

In the previous approach, the goal comprised the entire task. However, in this work, a satisfaction goal is just a single node in the goal graph. This means there are many possibilities for interesting task combinations. One could teach a task as achieving three distinct goals in a row, or combine both goal-based and procedural nodes within the same graph (as shown in the *serve* task in Section 7.6).

Currently, a satisfaction goal can only be a conjunction of one or more predicates that are a desired property of an entity (such as `closed` or `visible`) or a desired relation between two entities (such as `in`). This does offer some significant limitations. For example, a goal cannot be a disjunction of predicates. The agent cannot have a goal of learning some new knowledge or achieving some low-level motor state.

Task 1 Keep the milk in the fridge. The goal is that the milk is always in the fridge. ... You are done.
Task 2 Monitor the door until the meeting is over. The goal is that the door is always closed. ... The meeting is over.
Task 3 Observe Bob for three minutes. The goal is that Bob is always visible.

Table 8.1: Three examples of tasks with maintenance goals that Rosie can learn.

8.1.2 Maintenance Goals

A maintenance goal relaxes the assumption that the task is complete when the goal is achieved. Instead, the agent will try to maintain the goal over time. For example, following a person could be defined as keeping a fixed distance from that person. The agent can decide to act only if the goal is not satisfied, or could take preemptive actions to avoid that happening in the first place. Since the goal no longer provides a termination condition, the agent needs some other way of ending the task.

We indicate a maintenance goal by adding an optional `maintain` flag. In order to teach the agent a maintenance goal, the instructor must use the keyword *'always'* in the goal description, as in *"The goal is that the door is always closed."* If the agent sees that keyword, it adds the flag `maintain` to the goal. This tells it to wait when the goal is satisfied instead of selecting the next goal. In our implementation, the agent will only take steps to satisfy the goal and not proactively take steps to keep the goal satisfied. If the goal later becomes unsatisfied, Rosie can do planning and perform actions to achieve it again. Adding support for maintenance goals required very little modification to the agent. The only real change was having it avoid selecting the next goal when a maintenance goal is satisfied. No change is needed for the retrospective policy learning either, as our new method learns the policy for each subtask through separate learning episodes instead of trying to learn the entire task at once (Section 7.4.3).

The instructor has multiple ways of telling Rosie when the task is finished. She can explicitly tell it when to stop by saying *"You are done."* Or she can give an explicit termination condition, such as an until clause (*'until the meeting is over'*) or a temporal clause (*'for three minutes'*). Table 8.1 shows three examples of teaching tasks with maintenance goals.

8.1.3 Negative Goals

A negative goal is satisfied when none of its predicates hold. For example, one could give the goal of clearing the table as *"The goal is that no objects are on the table."* This type of goal ends up causing some major difficulties for our approach. It is easy enough to represent by adding a `negated` flag to the goal, and the agent can do planning to find actions that will achieve this. However, it is not generally compatible with our approach to explanation-based generalization. It is very difficult to explain why some predicate is *not* present in the world, which is required for creating causal connections from the initial state to the goal state. Soar does not backtrace through local negations, so even if a rule tests that a predicate does not exist, it will not be connected to a previous rule that removed the predicate from the state. Thus Rosie could learn such a task, but it cannot learn a policy for it and so must perform planning every time it executes the task. It may be possible to extend the learning to keep track of predicates that were removed as the result of actions and use them to generate an explanation, but that by itself is not guaranteed to learn a correct policy and is left to future work.

8.2 Procedural Task Formulation

In a procedural task formulation the agent has a declarative representation of the specific actions the agent needs to take, and the control flow among them. It could be as simple as a linear sequence of actions, or it could include complex control flow such as loops, conditions, and enumerations. Once learned, the agent can simply follow the procedure to execute the task. Often this procedure is explicitly described by the instructor, and can be easy to teach as the instructor just has to provide a sequence of commands. This is also easier for the agent because it can rely on the procedure instead of needing to learn a policy or perform complex planning. For example, one does not need to be an expert in the science of cooking in order to follow a recipe. However, if the procedure does not include how to deal with problems that arise during execution, it can be hard to recover from errors or unforeseen issues. The procedure only encodes *what* to do, not *why* each step is necessary. It can also be difficult to transfer a learned procedure to similar tasks.

The addition of the goal graph allows Rosie to learn and represent procedural tasks in addition to goal-based tasks. When the agent is learning a task and is given a new subtask command without having an active goal, it creates a new procedural goal node and adds an edge from the previous node to the new one. This procedural node has the goal of executing the subtask given in the command. It then actually executes the task in the world to satisfy the goal, and this process repeats. When the instructor says *"You are done,"* the agent appends a terminal goal node and completes the task. To execute the task, the agent simply has to follow the sequence of procedural nodes and execute each subtask. Since procedural tasks and goal-based tasks share a common representation, the agent can compose and blend them together in interesting ways. We demonstrate this through three case studies in Section 8.4.

Prepare the main office.
Complete the following tasks.
Open the door.
Turn on the lightswitch.
Move a water onto the desk.
End of list.
You are done.

Table 8.2: An example of a composite task.

8.2.1 Composite Goals

A variation of a procedural task is one where the agent is supposed to execute a set of tasks, but where the ordering does not matter, like a to-do list. An example of such a task is shown in Table 8.2. We use the key phrase *"Complete the following tasks"* as a trigger to start learning a composite task goal. The agent creates a new procedural goal node and then waits for the next subtask instruction. After each one, it will add it a predicate to the current goal (the goal of executing that subtask) and actually execute it in the world. This repeats until the instructor says the phrase *"End of list"*. The end result is a single subgoal node that is associated with multiple subtasks. It is considered satisfied when all the subtasks are executed.

8.3 Optimization Formulation

In an optimization formulation, the agent tries to maximize (or minimize) a particular metric. It is similar to a goal-based formulation, except that instead of trying to achieve a desired state, the agent has a way of evaluating states and is trying to find one state that maximizes the value. The metric can be explicitly provided by the instructor, where the agent learns a representation of the metric and can evaluate a state with it. Then the agent try to optimize it through methods such as planning or reinforcement learning (RL). Alternatively, the metric can be implicit if the instructor only provides a reward signal. Unlike a goal-based task, here there is no clear way to know when the agent has achieved the optimal state unless it can somehow detect that it has reached the global maximum. This formulation is also subject to many of the pitfalls that optimization problems can have with large/infinite, non-convex state spaces.

One common version is interactive reinforcement learning (Arzate Cruz & Igarashi, 2020), for example, Thomaz & Breazeal (2008). There the agent is trying to maximize a metric (specifically maximizing reward), but the metric is not explicitly provided. Instead, the agent gets a reward signal from the instructor. This precludes internally simulated exploration, since the agent cannot calculate the reward on its own. But through acting in the world, the agent can learn a policy through reinforcement learning. This may be a particularly difficult way to learn, since exploring in the real world is slow, instructor time is limited, and realistic state spaces are likely to be huge.

This is why having either an innate or learned representation of the reward metric that the agent can evaluate on its own is very useful. There is also the issue of the *credit assignment problem* (Sutton, 1985), where the agent has to assign the delayed reward to the appropriate action.

We believe that the goal graph representation is sufficient to represent and learn optimization tasks, although an implementation is left to future work. Such an implementation would involve adding an optimization goal node that describes the objective. One way would be to learn a declarative representation of a metric and then perform planning to search for a state that maximizes its value. Soar does include reinforcement learning, so in theory it would be possible for Rosie to learn task policies using RL, although the credit assignment problem would be particularly difficult when there are long time scales between when a decision is made and the reward is received. Rosie might have to do some sort of retrospective reinforcement learning where it can replay the task execution internally and then update its policy.

8.4 Blended Formulations

A significant claim of this work is that using a unified representation that accommodates both goal-based and procedural task formulations improves the overall robustness of the agent and expands the space of tasks it can learn. Since these formulations have strengths and weaknesses that are complimentary to each other, having both enables learning tasks that blend their features and combines their strengths. A goal-based formulation offers high flexibility and adaptation at the cost of requiring specialized knowledge (action models) and an explicit representable goal. A procedural formulation is less flexible, often requires more instruction, but has fewer knowledge requirements. Therefore, having the ability to represent both should enable learning a wider range of tasks compared to only using one formulation.

To evaluate this claim, we present several examples where utilizing both types of formulations improves the task learning ability of the agent. In the first example, we show that utilizing the problem-space formulation within procedural tasks improves the robustness. In the second example, we show how procedural learning can overcome policy learning failures for goal-based tasks. The third example demonstrates how subtasks with different formulations can be composed within a parent task.

8.4.1 Planning Within Procedures

The first example of blending involves the agent using the planning capabilities intended for goal-based tasks in order to execute a procedural task. Since each individual subgoal in the graph is treated as a proper problem-space, each procedural step still involves achieving a goal, only the goal is to perform a specific subtask. Therefore, if the desired subtask is not immediately available, e.g. if some precondition is unmet, then the agent can perform planning to make it possible. This makes the agent more flexible and robust to minor task variations.

Dialog	**Learning**	**Execution**
Move the mug into the pantry.	initial TCN, S: start	
What is the next goal or subtask of move?		
Pick up the mug.	G_1: exec(pick-up$_1$)	`pick-up`$_1$(e$_{mug}$)
What do I do next for move?		
Put the mug into the pantry.	G_2: exec(put-down$_2$)	`put-down`$_2$(e$_{mug}$,e$_{pantry}$)
What do I do next for move?		
You are done.	T: terminal	
	policy{put-down$_2$ pick-up$_1$}	
Move the fork into the drawer.		`pick-up`$_1$(e$_{fork}$) G_1
	subtask put-down$_3$	`put-down`$_3$(e$_{fork}$) G_2
	subtask open$_4$	`open`$_4$(e$_{drawer}$)
		`pick-up`$_1$(e$_{fork}$)
		`put-down`$_2$(e$_{fork}$,e$_{drawer}$)
	policy{open$_4$ put-down$_3$ }	

Figure 8.1: A training and testing interaction for teaching move procedurally. In the testing interaction, Rosie uses planning to open the drawer to satisfy the preconditions of put-down.

As an example, we demonstrate the agent learning the *move* task as a procedure. Our goal is to show how planning capabilities can be applied while executing procedural tasks, thus making instruction easier and the learned task more robust. The script can be seen in Figure 8.1. The upper group of sentences shows the training interaction where the instructor does not give a goal but instead gives two steps to complete the task. When the agent is told to *"Pick up the mug,"* it has no active goal so it creates a new procedural goal node in the graph and connects the start node to it. Rosie executes the subtask and then asks what to do next. It repeats this process for the second command *"Put the mug into the pantry."* Finally, it is told *"You are done"* and adds a terminal node to the goal graph. The learned task concept network is shown in Figure 8.2. The three squares in the center are slots that connect the arguments in the task to those in the subtasks. From left to right these correspond to the items *mug*, *in*, and *pantry* in the training interaction.

To test our claim, we had Rosie perform a task where the previously learned procedure does not perfectly apply. That is, if it just relied on the procedure, it would fail. We tell the agent to *"Move the fork into the drawer,"* where the drawer is closed. The first subtask of *pick-up* is completed successfully, but then the agent moves to the goal of performing `put-down(fork, in(drawer)` and cannot immediately execute this subtask as the drawer is not open. However, since Rosie has planning capabilities, it generates a plan that satisfies the preconditions of the *put-down* subtask, which it then performs. These additional steps end up being `put-down(fork), open(drawer), pick-up(fork)`. Note that it does not add them to the procedure but instead learns policy rules for them. For example, it learns a rule summarized as:

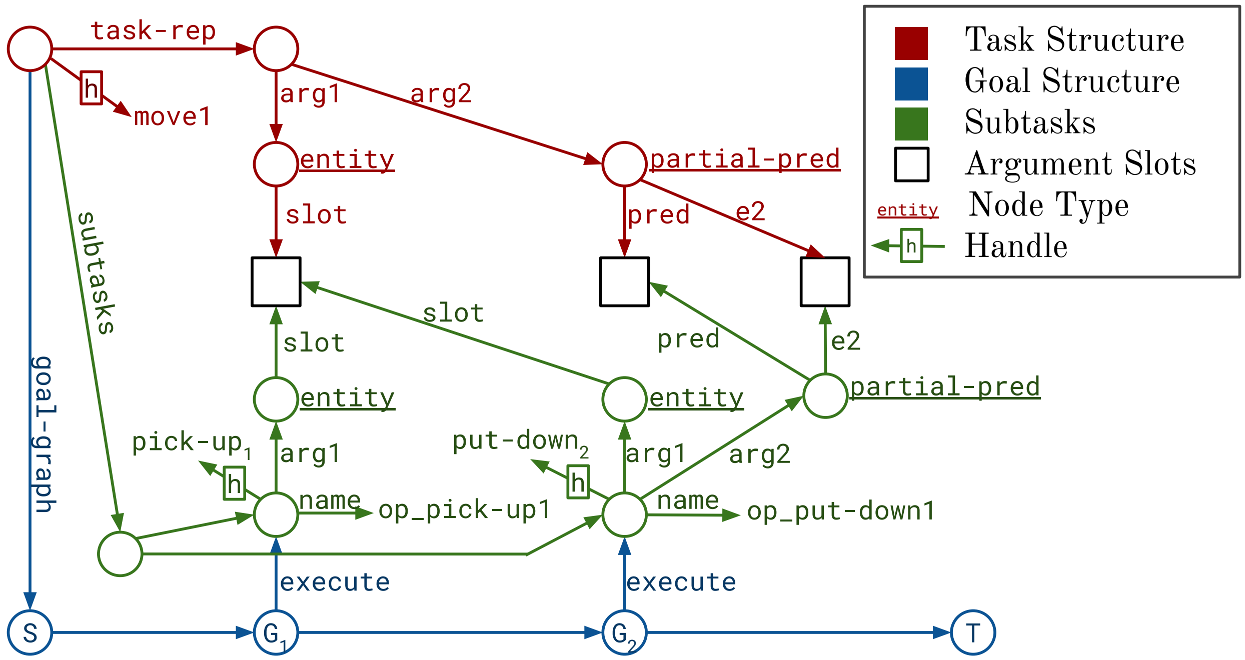


Figure 8.2: The task concept network from learning move as a procedural task. Note that each goal represents executing a specific subtask.

```
IF:
   current-task=move($e_i$, $pred_j$($e_k$)) ∧ goal=perform(put-down$_2$)
   ∧ not-grabbed($e_i$) ∧ closed($e_k$)
THEN:
   prefer open($e_k$)
```

The overall plan is non-optimal, but it shows how the agent uses its action models and planning capabilities to successfully adjust to minor variations in a procedural task. Furthermore, the procedure is made more adaptable because the specific steps are at a higher level of abstraction, as described previously in Section 7.5. Since we utilize learning and planning within innate tasks, some of the adaptability is contained within them. Consider if instead the procedure for move had to be described at a lower level of abstraction, including actions such as *approach* and *view*. Not only would that be harder for the human to instruct (as the instructor would have to have a more detailed model of the agent's actions and perceptual state) but the resulting procedure would be much more specific to the particular environment state when the task was taught.

However, the flexibility for procedural tasks is still fairly limited. Note that in the second move task, the resultant steps were non-optimal. The agent had to pick up the fork as the first procedural goal, even though it immediately put it down again in order to open the drawer. Furthermore, the move task as taught is actually only valid for moving an object *into* another one, and would not work for the task *"Move the apple onto the table."* This is because the subtask put-down(e_i, in(e_{rec})) is considered different from put-down(e_i, on(e_{surf})). And of course, if the agent needed

Dialog	Learning	Execution
Fill the mug with water.	initial TCN, S: start	
What is the next goal or subtask of fill?		
The only goal is that the mug contains water.	G_1: contains(e_1, e_{water}) T: terminal	
What do I do next for fill?		
Pick up the mug.	subtask pick-up$_1$	$\texttt{pick-up}_1(\texttt{e}_{\texttt{mug}})$
What do I do next for fill?		
Put the mug into the watercooler.	subtask put-down$_2$	$\texttt{put-down}_2(\texttt{e}_{\texttt{mug}},\texttt{e}_{\texttt{clr}})$
What do I do next for fill?		
Press the blue button on the watercooler.	subtask press$_3$	$\texttt{press}_3(\texttt{e}_{\texttt{btn}})$
(Goal is achieved)	G_2: exec(press$_3$) G_3: exec(put-down$_2$) policy [pick-up$_1$]	

Figure 8.3: A training interaction for teaching the task *"Fill the mug with water."* Rosie lacks a model of what the blue button on the water cooler does, so instead learns the three subtasks as procedural subgoals.

a completely different set of actions to execute the task, the procedure would be useless. However, we have developed some ways to specify conditional actions which can overcome this limitation in some cases (Section 9.4).

8.4.2 Overcoming Explanation Failures

The second example of blending involves the agent learning procedural steps within a goal-based task to overcome a failure during explanation-based policy learning. Learning a policy for a goal-based task requires generating an explanation of why each subtask led to the goal using planning knowledge. This requires highly-specialized causal knowledge about changes to the world and action models. If the agent lacks some of that knowledge, then the explanation process will fail, and the agent will not learn a policy and likely not be able to execute the task in the future without help. To help overcome this limitation, we present a method to allow the agent to learn procedural subgoals as a fallback for subtasks that it can not explain. The agent inserts a procedural subgoal into the graph for each subtask that it could not explain. The learned task is not as flexible as a full policy, but can still be executed in similar circumstances without needing further assistance.

8.4.2.1 The Fill Task

To demonstrate this capability, we had the agent learn a task where it lacked certain domain knowledge: filling a mug with water from a water cooler in the Magicbot simulator (Figure 8.3).

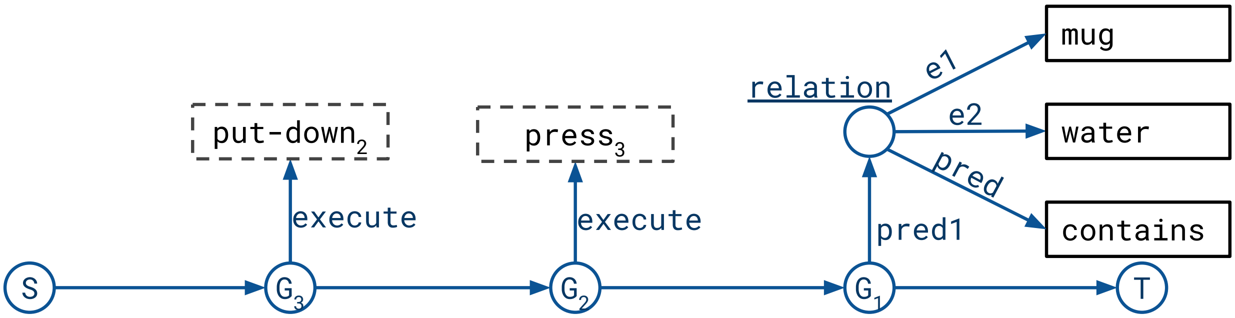


Figure 8.4: The goal graph learned for the fill task without an action model for the water cooler. Each step is inserted as a procedural subgoal.

Rosie can represent and detect the goal (*"The goal is that the mug contains water."*), but has no knowledge of how to use the water cooler. Thus the agent cannot generate a plan and needs to ask for help for each step. It must place the mug into the cooler and press a blue button to dispense the water. Once the button is pressed, the cup fills with water and Rosie detects that the goal has been achieved.

During the retrospective policy learning, the agent tries to explain why the *press* subtask was necessary, but this fails because it lacks the proper action model. It therefore creates a procedural subgoal G_2 with the goal of executing $\texttt{press}_3$ and inserts it into the goal graph immediately before the current goal G_1. This process also happens for the $\texttt{put-down}_2$ subtask. It does successfully explain the $\texttt{pick-up}_1$ subtask because it satisfies the preconditions of $\texttt{put-down}_2$, so it does not create a fourth subgoal.

8.4.2.2 Testing Agent Variations

To further analyze this capability, we tested three different versions of the agent with four variations on the fill task above. The first agent, **Agent 1**, was the full version of Rosie plus an action model of the button on the water cooler. This action model is sufficient for planning and explaining the steps of the task. This agent represented the best possible performance when the agent has specialized domain knowledge. The second agent, **Agent 2**, was the full version of Rosie but without this action model. This represented our approach when the agent lacks a certain action model. It needed to learn the task as a procedure as described in the previous paragraph. The third agent, **Agent 3**, lacked the ability to learn procedural subgoals. This represented the prior work, where if it failed to explain a particular subtask, the entire policy learning phase failed.

We used the Magicbot simulator for the test, with four task variations of increasing difficulty (Figure 8.5). These tasks were designed to require more and more generalization from the training task. The first task is identical to the training task, and each subsequent task has greater variation. The last one involves filling a cup with milk instead, where the agent has to pour the milk carton into the cup instead of using the water cooler. Rosie does have a model for the pour action, so it

Test 1: Fill the green mug with water.
Identical to the training task.

Test 2: Fill the green mug with water.
Green mug starts out grabbed.

Test 3: Fill the blue mug with water.
Blue mug is hidden in the cupboard.

Test 4: Fill the cup with milk.
Agent must pour the milk carton instead.

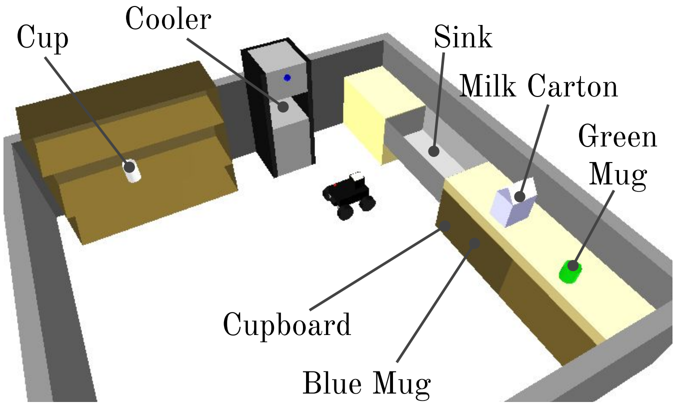


Figure 8.5: The four test variations of the fill task and the testing environment.

can effectively create a plan for that variation.

For each agent, we taught the original fill task as shown above, then told it to perform each of the four testing tasks. If the agent asked for help, we gave it the next step in the task. Therefore, the agents always correctly performed the task, even if no learning occurred. We were interested in overall learning behavior, so we measure the number of times the agent asked for help and whether it learned a policy rule or procedural subgoal for each new subtask. This will give an indication of how well the agent learned and how well it transferred knowledge across task variations. The expectation was that Agent 1 would learn the entire task on-policy and not need additional instructions after the goal, Agent 2 would handle the similar tasks but fail when required to perform very different steps (task 4), and Agent 3 would never successfully learn to fill the cup with water.

The results are shown in Figures 8.6 and 8.7, with Figure 8.6 showing the number of subtask instructions needed to teach each task, and Figure 8.7 showing the subtask decomposition used to execute each task and what was learned. Agent 1, which had a special button action model, could generate a plan for each task variation and did not require additional instructions. The only exception was in the training task, where it did ask for the first subtask due to exceeding its maximum search depth limit.

Note that for task 4, since the agent has a goal representation, it finds an alternative plan, without instruction, to fill the cup with milk instead of water. Agent 2 was almost as effective despite not having the action model. After learning the task as a procedure, it can handle three of the tasks without needing further instruction by using the planning capabilities for procedural subgoals described in the previous Section 8.4.1. However, the procedure was unable to adapt to task 4, which requires completely different steps. The agent first filled the cup with water ($goal_2$ and $goal_3$) before asking for help. The instructor told it to empty the cup into the sink and then the agent could pour the milk to satisfy the goal. Thus Agent 2 did successfully execute the task, but with the unnecessary steps of filling the cup with water first. Agent 3, which could not learn

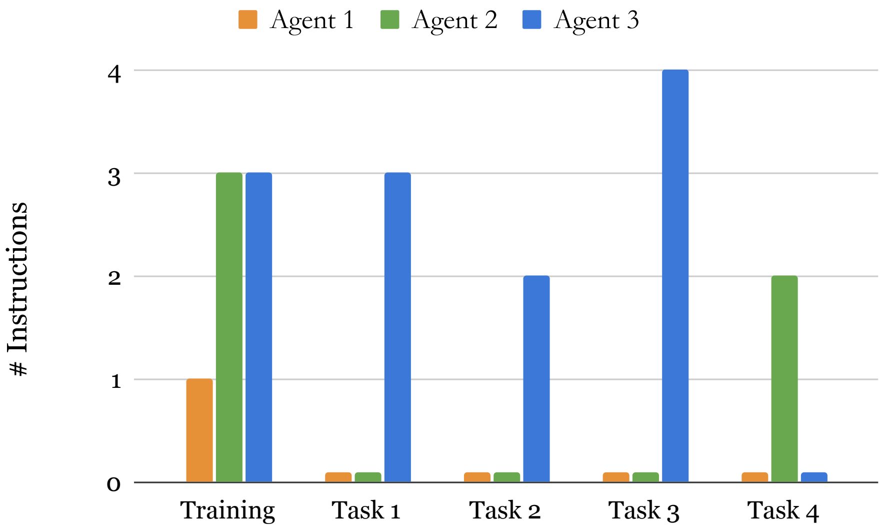


Figure 8.6: The number of subtask instructions given for each agent and task. Zero instructions indicates that the agent used planning or previous knowledge to perform the task.

	Agent 1	Agent 2	Agent 3

Training	`pick-up(mug)`
	`put-down(mug, wc)`
	`press(button)`
		`pick-up(mug)`
		`put-down(mug, wc)`
		`press(button)`
			`pick-up(mug)`
			`put-down(mug, wc)`
			`press(button)`

Task 1	`pick-up(mug)`
	`put-down(mug, wc)`
	`press(button)`
		`pick-up(mug)`
		`put-down(mug, wc)`
		`press(button)`
			`pick-up(mug)`
			`put-down(mug, wc)`
			`press(button)`

Task 2	`put-down(mug, wc)`
	`press(button)`
		`put-down(mug, wc)`
		`press(button)`
			`put-down(mug, wc)`
			`press(button)`

Task 3	`find(mug)`
	`pick-up(mug)`
	`put-down(mug, wc)`
	`press(button)`
		`find(mug)`
		`pick-up(mug)`
		`put-down(mug, wc)`
		`press(button)`
			`find(mug)`
			`pick-up(mug)`
			`put-down(mug, wc)`
			`press(button)`

Task 4	`pick-up(milk)`
	`pour(milk, cup)`
		`pick-up(cup)`
		`put-down(cup, wc)`
		`press(button)`
		`pick-up(cup)`
		`pour(cup, sink)`
		`put-down(cup)`
		`pick-up(milk)`
		`pour(milk, cup)`
			`pick-up(milk)`
			`pour(milk, cup)`

🟦 Learned Policy
🟥 Learned Procedure

Figure 8.7: The subtasks performed to execute each task. A blue background indicates that a policy rule was learned for that subtask, and a red background indicates that a procedural subgoal was added for that subtask.

procedural subgoals, completely failed to learn anything in the training and first three testing tasks (Figure 8.7). It lacked the necessary action model, so the explanation failed and the agent had to continually ask for help. It did correctly perform task 4, as it did know about pouring milk.

Overall these results demonstrate one benefit of having a unified representation that supports both goal-based and procedural tasks. Allowing procedural subgoals within a goal-based task allows the agent to be more robust to explanation failures. The procedure it learns is not as general as a pure policy (Agent 1), but it is better than nothing (Agent 3). And although such a procedure will lead the agent astray for versions of the task that require completely different steps (task 4), there are some simple ways that both versions could be taught using conditional actions (Section 9.4).

Another solution would be for the agent to learn a model of certain actions. This could happen through instruction (*"If an empty cup is in the watercooler and you press the blue button on the watercooler then the cup contains water."*) or learning from experience. This is outside the scope of this thesis but is one interesting area of future work. One issue with learning action models within our current approach is that once a procedural subgoal is learned, the agent will never try to learn a policy rule for that subtask. Rosie would need a way to detect that the procedural subgoal is no longer required and delete it.

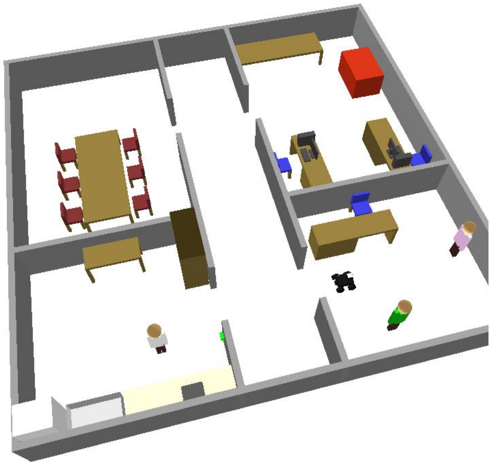
8.4.3 Composing Task Formulations

The third type of blending is being able to compose tasks with different formulations both horizontally (different formulations used within the same task) and vertically (a task of one formulation can have a subtask of another). This is crucial for supporting large-scale tasks that use many different types of task formulations in an extensive hierarchy. Fortunately, our unified task representations make this fairly simple. Since all tasks share a common operator representation, one task can use another without needing to know how it is implemented. There still are some restrictions on how effectively tasks can be composed, but in general any task can use any other.

To demonstrate this task compositionality, we created a guide task that involves giving someone a tour of the lab. The instructions for this task are shown in Figure 8.8, along with the environment. Guide is learned as a procedure, where Rosie is taught to first serve the person a drink, then lead them to the lab and tell them a few things about it. Then Rosie lets the person look around while it follows them, and once they are finished it leads them back to the main office. The complete goal hierarchy that Rosie learns is shown in Figure 8.9. To verify that Rosie had learned the task correctly, we had it repeat the task with a different person and confirmed that it completed all the steps without asking for any help.

We designed this task to demonstrate different types of task composition that Rosie can learn. The first type of composition is a task of one formulation using a subtask with a different one. This is shown by the guide task, which is formulated as a procedure, having a goal-based subtask (serve) and a maintenance subtask (follow). This is a simpler type of composition since the agent does not need to know anything about a subtask to use it in a procedure, other than its representation. It

Guide Alice.



Serve Alice.
The goal is that Alice is holding the desired drink.
Ask Alice "What drink would you like?"
A water.
Remember the answer as the desired drink.
(*Rosie gives Alice a water from the kitchen.*)
Permanently remember the desired drink
as the preferred drink of Alice.
You are done.

Say "Follow me to the lab."
Go to the lab.

Complete the following tasks.
Say "To the right are the computer workstations."
Say "On the left is a workbench."
Say "The red cabinet contains various tools."
End of list.

Say "Feel free to look around and let me know when you are finished."
Follow Alice.
The only goal is that Alice is always visible.
Finished

Say "Follow me back to the main office."
Go to the main office.
You are done.

Figure 8.8: The instructions used to teach the guide task and the testing environment. This task includes predicate, maintenance, procedural, and composite subgoals.

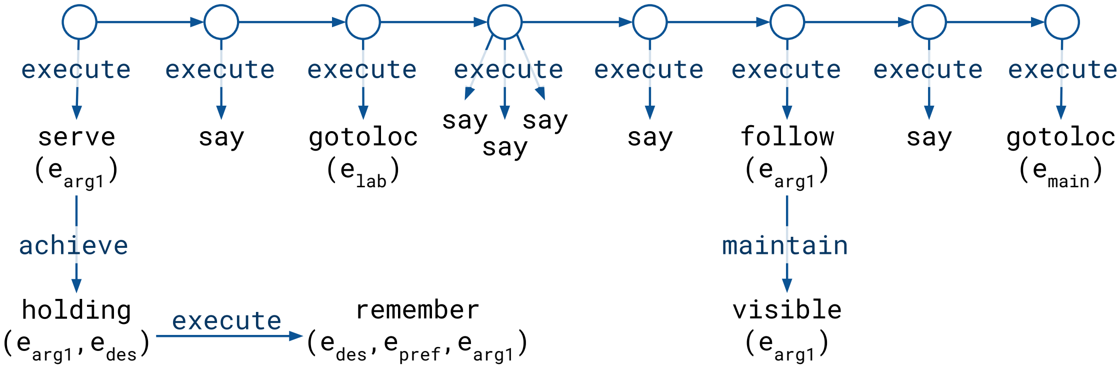


Figure 8.9: A simplified depiction of the complete goal hierarchy learned for the guide task.

is more difficult to have a procedural subtask within a goal-based task. Since Rosie does not learn action models for procedural tasks, it cannot learn a policy for them. Its only option is to represent a procedural subtask using explicit subgoals (as in Section 8.4.2),

The second type of composition demonstrated in the guide task is having a task with different types of subgoals. For example, the guide task has both singular procedural subgoals and a composite subgoal. In addition, the serve task has a predicate subgoal (holding) and procedural subgoal (remember). Rosie can easily chain together multiple types of subgoals when learning a task. This is due to using a unified representation, and that the agent does not attempt retrospective policy learning over multiple subgoals. Currently, we have not tried to learn action models or proposal rules for tasks with more than one subgoal. Another area of future work could be extending the proposal and action model learning to handle more complex tasks.

8.5 Discussion

In this chapter, we have described our novel goal graph representation that can represent both goal-based and procedural task formulations. Since each type has its own strengths and is suitable for certain types of tasks, including both types significantly extends the space of tasks that our agent can learn when compared to other approaches that only learn a single formulation. Our hybrid approach represents high-level task control flow within the graph structure while still using the problem-space definition for individual subgoals. Using a unified representation for both formulations leads to useful ways of blending the two, which we demonstrated through several evaluations. We also showed that subtasks of different formulations can be composed within the same task hierarchy.

There are still many avenues for further improvement. Another major type of task formulation is optimization tasks, which the goal graph should be able to accommodate but would require significant additions to policy learning. There are also many types of goals that cannot be represented (e.g., predicate disjunctions, kinestetic movements).

CHAPTER 9

Diverse Modifiers

A highly desired characteristic of tasks learned by an agent is that they can be applied to a wide range of situations. If the instructor can only use the singular canonical version of a task, it greatly limits that task's usefulness. Often, the instructor will want to add certain phrases that modify how the task should be performed. If a task learning agent can handle these additional constraints and modifiers, it greatly increases the ways in which a single task can be used. For example, it would be extremely limiting if the only valid way to give a *move* command was to use the exact form 'Move a cup onto the table', especially when it is used within other more complex tasks. There are many different types of phrases that can modify this command to increase the ways in which it can be used. For example, there are temporal clauses ('After dinner is finished, move a cup onto the table.'), conditional clauses ('If the table is clear, move a cup onto the table'), and frequency arguments ('Move a cup onto the table twice'). A truly general task learning agent must be able to adapt its knowledge to accommodate these modifiers.

In this work, we extend our representations and learning mechanisms to handle three novel types of modifiers during task learning and execution. It is important that this be done in an integrated way, so that our agent learns just as effectively when these modifiers are used and can transfer from one variant of the task to another. Therefore, the extensions were designed to share as much of the existing task representations and learning mechanisms as possible.

To more formally categorize different types of modifiers, we take inspiration from the different categories of semantic roles (Palmer et al., 2010), which indicate what role each entity within a sentence plays in the situation, such as the agent, destination, or instrument. In the following Section (9.1), we describe one such categorization, VerbNet (Kipper et al., 2008), and discuss how each type of semantic role could be used to modify a task. Section 9.2 analyses the related work within the context of learning various modifiers. We then describe our extensions to support temporal, conditional, and repetitious modifiers (Sections 9.3, 9.4, 9.5). Throughout these sections are task learning examples and the corresponding task knowledge. In Section 9.6, we perform an evaluation where the agent is taught a canonical version of a task, and then verify that it transfers that knowledge across the various types of modifiers. Finally, given these extensions, we analyze the types of task structures that Rosie can learn and discuss its limitations (Section 9.7).

actor	agent	asset	attribute
beneficiary	co-theme	destination	duration
experiencer	extent	final_time	frequency
goal	initial_location	instrument	location
material	participant	patient	pivot
place	product	recipient	result
source	stimulus	time	theme
trajectory	topic	undergoer	value

Table 9.1: The 32 semantic roles defined within VerbNet.

9.1 Categories of Modifiers

A significant research question is identifying what are the different types of modifiers than can be used within a task command. One potential source comes from the computational linguistics community and the notion of a semantic role. Semantic roles describe the role of each participant in a sentence, answering questions such as who, what where, when, and how. The idea is that these roles deal with meaning, or semantics, and are mostly independent from syntax. There have been several different frameworks for categorizing semantic roles, including FrameNet (Baker et al., 1998), VerbNet (Kipper et al., 2008), and PropBank (Palmer et al., 2005). We will refer to those in VerbNet, as it is focused specifically on verbs and their arguments, which is clearly very appropriate for describing tasks.

Some semantic roles are necessary for a given verb, i.e., the action or event described by the verb would not be fully understood unless the role was defined (although it can be implicit). For example, a 'deliver' action at a minimum requires an agent (who/what is performing the delivery), a theme (the thing being delivered), and either a destination or recipient. Other semantic roles might be optional for a given verb and apply to many different verbs. VerbNet identifies a total of 32 semantic roles and arranges them in a hierarchy. We list them in Table 9.1 (Kipper et al., 2008).

We will not exhaustively go through the list, but highlight some that are particularly relevant to ITL and our work. For our tasks, the *agent* (the actor who carries out the task) is always the robot itself and is implicit in the command. Some approaches, such as Frasca et al. (2018), explicitly represent the agent and can learn from watching others perform the task.

One class of semantic roles, *undergoer*, describe participants in an event that are not the instigator. In Rosie, these tend to be represented as entity arguments within the task. For example, a *patient* is a participant that experiences a change of state or location from the action (as with the door in *"Open the door."*), whereas a *theme* is similar but does not go through a change (as with the table in *"Find the table."*). These roles can help point out current limitations and future work. For example, there is the role of *instrument*, which is something is used to help accomplish the task, such as a tool. For the most part, we have not learned any tasks that involve these, although the water cooler could be an example. Another role is *product*, which is a subtype of patient. To date, we have not looked at tasks that create something new.

Another major class of roles is *place*. Many of our tasks involve a *destination*, or a place that is a goal of the action. These are often represented as partial predicates. Another subtype of place is *extent*, which is a measurable change in a scalar value. This is used in commands such as *"Drive forward for three meters."* Finally, we also have some limited handling of the *initial_location* role, as the instructor can further constrain a task with a *from* clause (such as *'from the table.'*).

Several of the semantic roles are relevant to this chapter. VerbNet identifies a class of semantic roles underneath *time*: *initial_time*, *final_time*, *duration*, and *frequency*. The first three correspond to the three types of temporal modifiers that we describe later in Section 9.3. Frequency, or the number of occurrences of an event, is similar to our notion of a repetitious modifier (Section 9.5).

Using a taxonomy of semantic roles like the one from VerbNet can be helpful in identifying different categories of task arguments and modifiers and can help direct future work. It is not a comprehensive list, for example, *manner* is not included (a modifier that describes how a task should be performed, such as quickly or carefully), but that would likely be useful for an agent to learn. It also does not list conditions, which are a main part of this chapter, although they are somewhat related to the *cause* role.

9.2 Related Work

The core research goal of this chapter is having a task learning agent transfer knowledge of a particular task to many different variations, specifically those with temporal, conditional, and repetitious modifiers. The key is that the agent should not only be able to understand and execute commands that have those modifiers, but learn them as part of tasks. Many of the contemporary approaches to learning tasks through instruction are only concerned with transferring to different object or location arguments in the task. This is true for the previous approach (Mohan & Laird, 2014), although it did have support for an until clause, as well as for She & Chai (2017) and Suddrey et al. (2016). As mentioned, Frasca et al. (2018) can represent the different agent roles in a task, but none of the modifiers mentioned here.

Several of the approaches learn some form of conditional or looping structure. SUGILITE (Li et al., 2020) can learn conditions within a task, such as ordering a hot or iced coffee depending on the temperature. The HTN learned by Mohseni-Kabir et al. (2019) can encode conditional subtasks and will infer enumerations during learning (it asks if the same steps should be applied to other objects of the same type). However, it does not support true loops of multiple steps, as with our work and others. PLOW (Allen et al., 2007) could learn iterations that enumerated over each item in a set or a fixed number of items. It did not handle conditional actions however. The instruction graph from Meriçli et al. (2013) supports conditions and until clauses that specify the end of a continuous movement. However, it demonstrates very little generalization and cannot compose tasks hierarchically. The executable graph learned by Gemignani et al. (2015) could represent more

	start-clause	**end-clause**

duration	`after <duration>`
	after thirty seconds
		`for <duration>`
		for three minutes

time	`at <time>`
	at 3:15
		`until <time>`
		until 15:45

predicate	`when <predicate>` *when the light is off*	`until <predicate>` *until the door is closed*

Table 9.2: The different types of temporal clauses that Rosie can learn and the corresponding prepositional phrases.

complex structures such as a conditional branch, counting loop, and do-until loop. The loops are essentially identical to those that we demonstrate in this chapter, and they support single-step else branches. However, each node in their graph is a primitive action, they cannot compose tasks hierarchically.

What distinguishes our work is the ability to use temporal, conditional, and repetitious modifiers with both innate and learned tasks, and use them as part of learning even larger scale tasks. We also show that the temporal and conditional modifiers can be used within both goal-based and procedural task formulations, although the loops can only be learned within a procedural task.

9.3 Temporal Modifiers

Temporal modifiers are clauses that specify *when* as task should be performed. They can broadly be grouped into those determining the start, duration, or end of a task. In English, these are often indicated through prepositional phrases, with the preposition giving the temporal semantics. In this work, we have added support for various temporal clauses and mapped a specific prepositional phrase to each one. This mapping is shown in Table 9.2. During the natural language comprehension and interpretation phases, the temporal clause is categorized according to Table 9.2 and added to the task representation as either a `start-clause` or `end-clause`. We have chosen one common usage for each cell, but there are other phrases that could also apply. Further research into the agent's natural language comprehension could develop more general ways of mapping temporal clauses to semantic representations. This table is not an exhaustive list, as there are other types of temporal clauses that we are not considering, such as longer time scales (on Monday, this September, next year) or repeated time periods (every hour, every Tuesday). In the following sections, we describe how temporal clauses affect task execution and learning.

9.3.1 Executing Tasks with Temporal Clauses

In Rosie, temporal clauses are divided into those specifying when a task should start and those specifying when a task should end. If the task has a start clause, the agent waits for it to be satisfied before selecting the next subgoal in the goal graph. For a duration clause, the agent marks the start time and then waits for the given time period. For a time clause, the agent waits until the current time exceeds the given time. And for a predicate clause, the agent waits until it perceives that it is satisfied. One potential problem is if an object referenced in the start clause is not visible. For example, suppose the agent was told *"When the microwave is off, open it."* If the microwave is not currently visible, the agent has no way of knowing when this occurs. Therefore, we manually have the agent execute the `view` subtask for such an object. Since a start clause only delays normal execution until some condition is met, they can be used in conjunction with any task, whether innate or learned.

End clauses have a more limited use, as many tasks already have a well-defined endpoint when a goal or procedure is completed. Using an end clause with such tasks is somewhat nonsensical, e.g. *"Pick up the fork for three minutes."* or *"Move the white mug onto the table until 5:30."* However, they are very useful for tasks that do not have a well-defined endpoint or that can be interrupted. In previous chapters, we have seen examples of end clauses being used with a maintenance task (*"Follow Bob for three minutes."*) and as part of the find task (*"Explore until the soda is visible."*) When an end clause is satisfied, the agent immediately completes the task and returns to the parent task. For certain innate tasks that involve ongoing motor commands, the agent also sends a stop command.

9.3.2 Learning Tasks with Temporal Clauses

Adding support for temporal clauses did not require significant modifications to task learning. From a representational standpoint, temporal clauses are handled just like any other task argument. If Rosie is given a subtask with a temporal clause, it stores it within the subtask TCN and includes it in the learned proposal rule. Thus they can be used within procedural tasks without issue. For goal-based tasks, both duration and clock time clauses are ignored during policy learning as Rosie does not have detailed models of temporal dynamics or have goal representations that involve time. However, end clauses that contain a predicate have an interesting use case. Such clauses indicate that when a subtask is finished, the given predicate will be satisfied. Thus during planning or policy learning, the agent treats the end clause predicate like an action model and adds it to the world. For example, Rosie does not have an innate model that a microwave will turn off after some time. If we told the agent to *"Wait until the microwave is off."* then during task planning Rosie can use that to model the state change.

To show an example in practice, we revisit the task of filling a cup with water from Section 8.4.2 (Instructions in Figure 9.3). In that task, the agent did not have an action model of the water cooler button and could not explain why pressing the button achieved the goal, so it learned the task as a procedure. Another option is to add an until clause to the press action, as in *"Press the blue button on the water cooler until the green mug contains water."* Then when the agent models the press action it will add the predicate `contains(mug, water)` to the world and the goal will be

Fill the green mug with water.
The only goal is that the green mug contains water.
Pick up the green mug.
Put the green mug into the watercooler.
If the drink is water and the green mug is in the watercooler, then press the blue button on the watercooler until the green mug contains water.

Table 9.3: The instructions for teaching the fill task using conditions and an until clause as a substitute for a button action model.

achieved. However, this instruction is under-specified since it does not require that the cup be in the cooler. To correct this, the instructor needs to give the additional condition *"If the green mug is in the water cooler, then press ..."*. However, since Rosie generalizes the type of liquid, it needs to be further constrained to just water (*"If the drink is water ..."*). The full instruction for press is shown in Figure 9.3. This example demonstrates that an until clause can act in place of an innate action model to allow task explanation, but that it can be very particular and would be difficult for a novice to teach.

9.4 Conditional Modifiers

Adding conditions to a task or goal message allows the instructor to more precisely specify the circumstances when it should be pursued. This greatly increases the potential variations that a single task can encompass. Providing multiple goals with different conditions allows different outcomes depending on the type of objects involved, such as learning that silverware are stored in a drawer whereas drinks are stored in the fridge. Adding conditions to a task within a procedure introduces a branch in the control flow and allows multiple execution paths. In a previous chapter, we have seen that adding conditional clauses allows different behavior based on the response to a question or the presence of some knowledge in long-term memory (Serving drinks in Section 7.6).

The extensions required to learn and execute conditional tasks and goals are a major contribution of this thesis and help to address two major limitations of the previous work. First, since a major research goal of Rosie is one-shot learning, it aggressively generalizes a single goal description as broadly as possible. Suppose Rosie is taught that the goal of storing a fork is that the fork is in the drawer. If it then is told to store the ketchup, should it transfer the previous goal? Our approach maximizes transfer, and will also store the ketchup in the drawer. Adding support for conditional clauses allows the instructor to constrain goals to avoid over-generalization. We further discuss this trade-off between generalization and instructional efficiency at the end of this section. Second, given the current limitation that Rosie only associates one task with each verb, learning conditional modifiers goes a long way towards allowing a task to have multiple outcomes. Addressing these limitations greatly increases the intra-task diversity that Rosie can learn.

Type	**Example Phrase**	**Predicates**
unary	*if the door is closed*	closed(e_{door})
relation	*if the cup is on the table*	on(e_{cup}, e_{table})
exists	*if there is a preferred drink*	exists(preferred,drink)
visible	*if you see the fridge*	visible(e_{fridge})
conjunction	*if the you see the fridge* *and the fridge is closed*	visible(e_{fridge}) $\wedge$ closed(e_{fridge})

Table 9.4: The different types of conditional phrases that the agent can handle.

In the following sections, we describe how conditions are represented and then describe two cases of learning conditions as either constraints within policy rules or branches in the procedural graph structure. We then describe an extension made to allow additional conditional steps to be inserted into an existing procedure and finish with a discussion of a trade-off revealed by the results.

9.4.1 Representation

When the instructor gives an *if-then* statement with a task or goal, the agent takes the parsed if clause and turns it into a set of predicates. Table 9.4 shows the kinds of conditions that the agent understands. The set of condition predicates is handled like any other type of task argument in terms of how it is represented and stored. Any arguments are generalized by connecting explicit arguments to slots in the parent task concept network and storing default values for implicit arguments. For example, the goal for storing the fork is taught as *"If the fork is a utensil, then the goal is that the fork is in the drawer."* The condition gets turned into the predicate utensil(e_{fork}) and, when stored, the fork entity is connected to the entity slot in the store task.

9.4.2 Learning Policy Conditions

The first case of handling conditional modifiers is when they are given with a task command while the agent is working towards satisfying an existing goal. In this section we use a simplified version of the *serve* task from Section 7.6, where the goal is to give someone their desired drink. If the person's drink preference is known, the agent will serve that drink, and otherwise ask the person what they want. Instructions for the two variations are shown in Table 9.5. At line 7, the instructor gives a conditional *remember* task. The conditions are added to the subtask representation in the *serve* TCN as shown in Figure 9.1.

When the agent goes to learn the proposal rule for *remember*, it handles the conditions differently. Instead of adding them to the proposed subtask representation, they are instead matched against the current task, goal, and world. The subtask is only proposed if the conditions match. Therefore when the chunk is learned, the preconditions of the rule include the condition tests that were made. As an example, the *remember* subtask proposal rule is learned as:

1 Serve Alice.

2 *What is the goal or first action?*

3 The only goal is that Alice is holding the desired drink.

4 *What do I do next?*

5 Recall the preferred drink of Alice. (retrieves soda)

6 *What do I do next?*

7 If there is a preferred drink, then remember the preferred drink as the desired drink.

8 *(Rosie finishes the task by getting Alice a soda)*

9 Serve Bob.

10 *What do I do next?*

11 If nothing was recalled, then ask Bob "What drink would you like?"

12 *What drink would you like?*

13 A water.

14 *What do I do next?*

15 Remember the answered drink as the desired drink.

16 *(Rosie finishes the task by getting Bob a water)*

Table 9.5: The instructions for teaching the simplified serve task. These include conditional task modifiers received while working towards a goal.

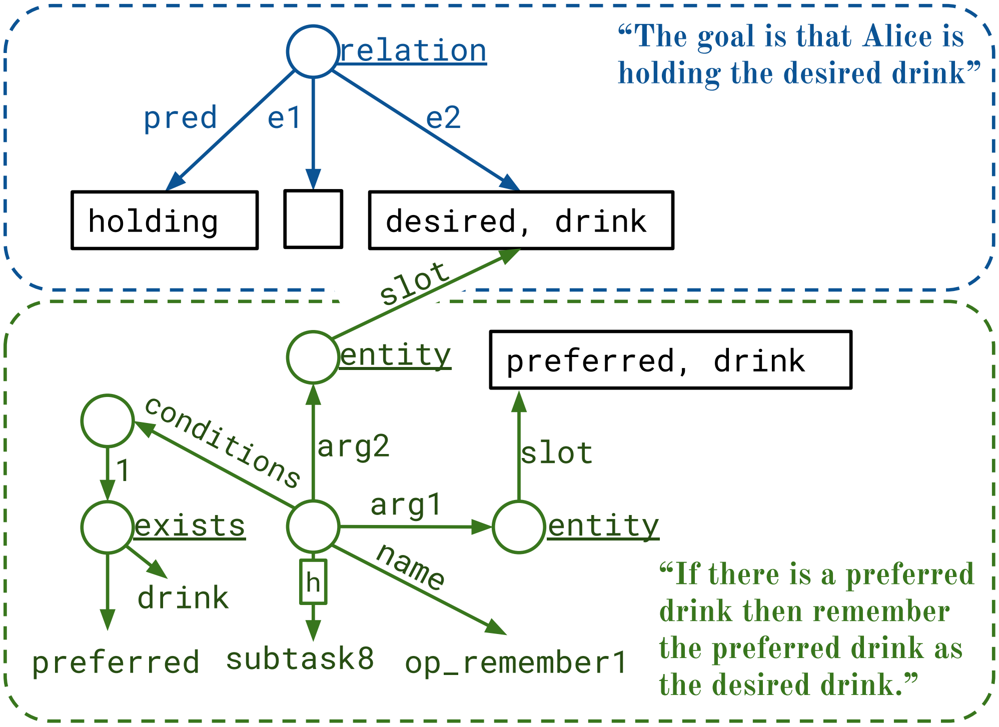
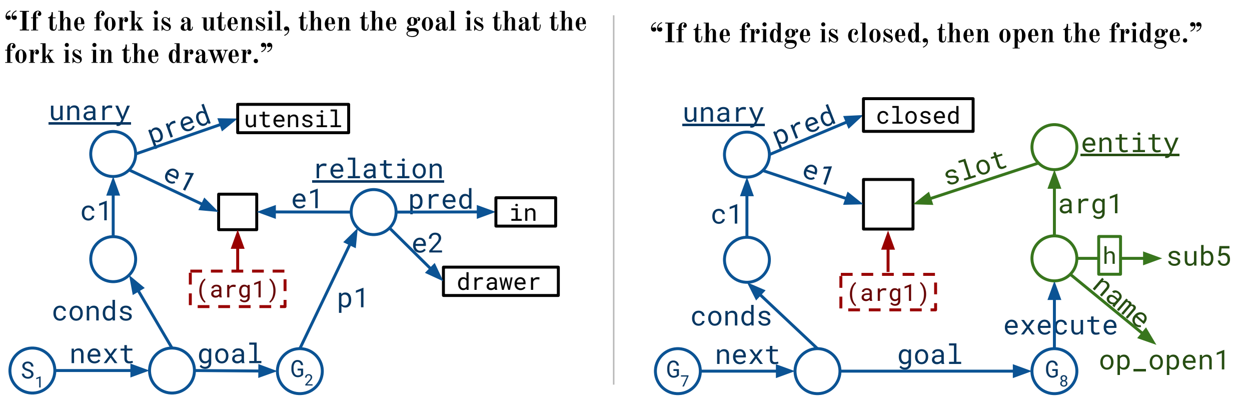

relation
pred
e1
e2
holding
desired, drink
"The goal is that Alice is holding the desired drink"
slot
entity
preferred, drink
conditions
arg2
slot
1
exists
arg1
entity
name
drink
h
preferred
subtask8
op_remember1
"If there is a preferred drink then remember the preferred drink as the desired drink."

Figure 9.1: The goal and remember subtask representations in the TCN for *serve*.



"If the fork is a utensil, then the goal is that the fork is in the drawer."

"If the fridge is closed, then open the fridge."

Figure 9.2: An example of the nodes and edges created in the goal graph when learning a conditional goal (left) and conditional subtask (right).

```
IF:
   current-task=serve($e_1$) $\wedge$ goal=holding($e_1$, $e_2$)
   $\wedge$ preferred($e_3$) $\wedge$ drink($e_3$) $\wedge$ preferred($e_4$) $\wedge$ drink($e_4$)
THEN:
   propose remember($e_4$, $e_2$)
```

The predicates *preferred* and *drink* appear twice, as they do in the TCN. Note that for learning a rule in this way to work, the conditions have to be satisfied when the command is given. If they are not satisfied, then the agent will fail to learn the proposal rule and will have to be given another instruction. If the rule is learned, the subtask will be proposed and the agent will continue on with the task. No other changes to the task learning or reasoning are required, as the conditions have been completely incorporated into the agent's procedural knowledge of that subtask. During retrospective policy learning, these preconditions will be included when generating the explanation and learning policy rules.

9.4.3 Learning Procedural Branches

The other use case for conditional tasks and goals is when the current goal is satisfied and the agent does not have a valid next goal to select. In this case, the agent will ask the instructor what to do next. If the instructor replies with a conditional goal or subtask, it indicates a branching point in the control flow. The new subtask or goal is added to the goal graph as normal, but the conditions are added to the `next` edge pointing to the new node. Figure 9.2 shows an example for both a goal (left) and subtask (right). Each has a conditional predicate on the next pointer that links to the rest of the TCN. Having branching paths greatly increases the range of tasks that can be represented, as it allows the agent to execute different behavior depending on certain conditions being met.

When a conditional node is created, there is a decision to make about what to do if the conditions are not met (i.e., the *else* branch). The question is whether a branch in the graph should default to

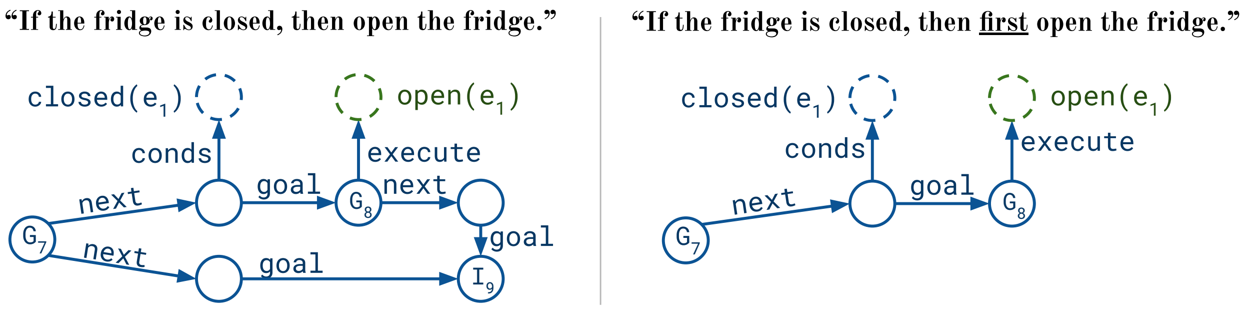

"If the fridge is closed, then open the fridge."
closed(e1)
open(e1)
conds
execute
next
goal
next
next
goal
goal
G7
G8
I9
"If the fridge is closed, then first open the fridge."
closed(e1)
open(e1)
conds
execute
next
goal
G7
G8

Figure 9.3: The difference between a temporary branch in the graph (left), which is the default, and a permanent split (right), indicated by the keyword *first*.

a permanent or temporary split. We chose to make the default decision that the conditional node is a temporary branch and that it immediately merges back with an else branch. This is in line with our assumption that task variations tend to be similar, and it saves repeat instructions if the rest of the steps happen regardless of the branch. We do allow the instructor to explicitly chose the other option by using the keyword *first*, as in *"If the fridge is open, then first open the fridge."* In this case, the agent will create a permanent split in the graph. The two options are shown in Figure 9.3. By default, when the agent creates a conditional node it also creates an else branch pointing to an intermediate node and links the conditional node to it as well (left). If the instructor says the keyword *first*, no such else branch is created (right). It would be interesting for future HRI research to explore how people teach a conditional step in a task and what assumptions they make. This implementation cannot represent every type of if/else structure, we more fully describe the limits of which goal graph structures and possible in Section 9.7.

9.4.3.1 Conditional Store Example

As a motivating example, we use a version of the store task with conditional goals (Table 9.6). In the store task, we give two specific goals that are conditioned on the type of object (e.g., a utensil goes in drawer) and a default goal that involves the agent asking where to store the object. When it receives the conditional goal, it creates the new goal node and adds the conditions to the incoming edge. Here we show the use of the keyword only (*'the only goal'*). When the agent sees that word, it will connect the goal to a terminal node. This is a convenience for the instructor, it saves them having to tell the agent the task is over when the goal is achieved. It also overrides the default behavior of creating an else branch. The full goal graph learned from the three store tasks is shown in Figure 9.4.

For each edge in the graph, the agent learns a `select-next-goal` proposal rule. If the edge has conditions, those get tested when the rule is learned and are included in the preconditions. Figure 9.5 shows this for the three goals of store. The left and middle include the conditions from the graph edge, and the right one acts as a default option. Note that when the agent goes to select

Store the fork. If the fork is a utensil, then the only goal is that the fork is in the drawer.
Store the soda. If the soda is a drink, then the only goal is that the soda is in the fridge and the fridge is closed.
Store the plate. The only goal is that the plate is in the storage location. Ask “Where should I store it?” Remember the answer as the storage location

Table 9.6: The instructions for teaching the store task using conditional goals.

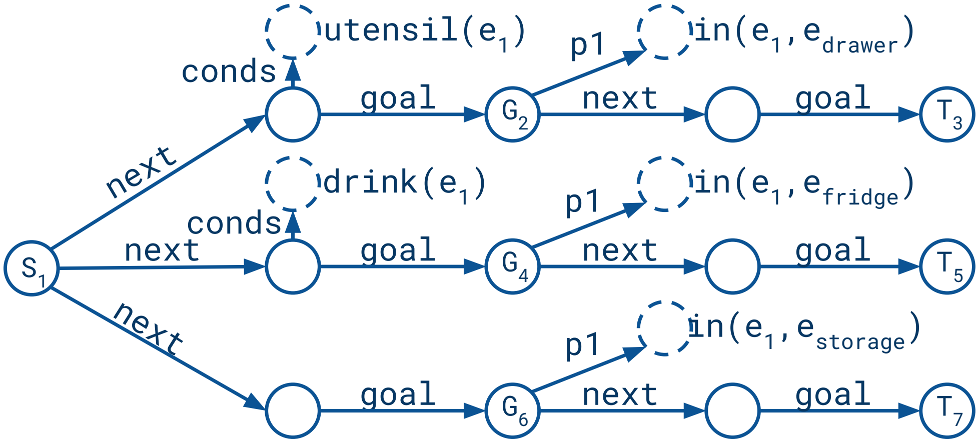


Figure 9.4: The complete goal graph learned for the store task with three branches.

IF:
task=store(e_1)
∧ satisfied(S_1)
∧ utensil(e_1)
THEN:
propose operator
select-next-goal(G_2)
	IF:
	task=store(e_1)
	∧ satisfied(S_1)
	∧ drink(e_1)
	THEN:
	propose operator
	select-next-goal(G_4)
		IF:
		task=store(e_1)
		∧ satisfied(S_1)
		THEN:
		propose operator
		select-next-goal(G_6)

Figure 9.5: The `select-next-goal` proposal rules learned for the first three nodes in the goal graph for store.

the next subgoal, there might be multiple valid choices. Currently, the agent selects randomly, but prefers edges with conditions over those without. This heuristic is based on the assumption that an edge with conditions is more specific than one without.

9.4.4 Learning Opportunistic Tasks

A special type of conditional task is what we call an *opportunistic task* where the agent should execute the task whenever certain conditions are met, regardless of its current goal. The instructor can indicate such a task by adding a *whenever* clause, such as *"Whenever you see trash on the ground then discard it."* Learning such a task follows a similar process to other conditional tasks: the message is turned into a task representation with conditional predicates and a general semantic task structure is created and stored. However, the task structure is not added as a subtask but a stand-alone task.

After the agent learns the representation, it now must have a way of recognizing when the conditions are met and performing the appropriate actions. This requires a mechanism that serves as a prospective memory – the ability to remember now to perform some action in the future (Brandimonte et al., 2014). Soar has no built-in prospective memory, but there are some options for implementation (Li & Laird, 2014). One possibility is for the agent to periodically query semantic memory to see if there are any opportunistic tasks with matching conditions. This requires the agent to be proactive and it is inefficient as most of the time there will not be any matches. Another option is to maintain a representation of the conditions for all opportunistic tasks in working memory and check it against the world. This would be more efficient from the processing side, but has higher memory requirements and raises issues when many opportunistic tasks have been learned.

Our solution is to learn a proposal rule for each opportunistic task that includes the task conditions as preconditions. Learning such a rule using chunking requires that the agent internally imagine and experience the situation when the task should be performed. Thus, when the agent goes to learn an opportunistic task, it creates an internal copy of the world and adds any objects or predicates required by the conditions. So for the example above, it would add a trash object if one did not exist. Additional structures are added to the state to mimic one where the agent is deciding what task to execute next. Assuming the agent has no existing tasks relevant to this imagined situation, the agent will not have any operators proposed, and it will encounter a state no-change impasse. In that substate, it can retrieve the representation of the opportunistic task from semantic memory. It then instantiates that representation by grounding objects and predicates using the world model and checks that the conditions are satisfied. It then proposes the task operator on the superstate, which causes a proposal rule to be learned for this imagined situation, and will fire if the agent is at a task decision point and the conditions are satisfied. An opportunistic task is preferred above other normal subtasks but is ignored during retrospective policy learning since it has nothing to do with achieving the parent task's goal. The benefit of this approach is that procedural rule matching is very efficient (sublinear in the number of rules) and it does not require any additional

Check the kitchen	**Check reception**	**Check the lobby**
Turn on the lights	Turn on the lights	Turn on the lights
Move plates into the sink	Fill the water pitcher	Store any books
Close the fridge		
Turn off the lights	Turn off the lights	Turn off the lights

Table 9.7: The desired subtasks to be performed in each room for the *check* task.

working memory structures. The main disadvantage is that there are no easy ways to remove the learned proposal rules if the interrupting task is no longer desired. It would be possible to learn a rule that rejects the proposed task, and avoid it in specific situations in the future, but the agent cannot excise the rule itself. An example of the agent learning and executing an opportunistic task will be shown in Chapter 10.

9.4.5 Inserting Additional Subtasks

At this point, we have only discussed learning a procedure in a single teaching interaction. This is usually sufficient if the task is a linear sequence of actions. However, this restriction becomes more burdensome if the task has many conditional steps. Without a way to insert additional subtasks into the goal graph during a later task, the instructor is forced to give the agent all possible instructions at once. As a motivating example, consider teaching Rosie a simple patrol task, where it should travel to a few different rooms, check each one, and perform some simple upkeep tasks. Table 9.7 shows the desired steps of the *check room* task for three different rooms. Notice that they have some subtasks in common (involving the lights) and others specific to the room. Without a way to add additional steps later, the instructor would have to give instructions for all three rooms at once.

To ease the instructor's burden, we have implemented a way for the instructor to interrupt the execution of a procedural task to give further subtasks. If Rosie is told to *"Interrupt [task]"*, the next time Rosie gets to a decision point for that task (when it has to select the next goal), it stops and asks for another instruction. Any new goal or subtask is inserted into the goal graph directly after the current node. This can happen multiple times until Rosie is told to *"Continue [task]."* Figure 9.6 shows the full set of instructions used to teach the three versions of the check task. There, the instructor interrupts later tasks to add additional steps for different rooms. Without this ability, the instructor would have to give all the steps when teaching the kitchen version.

9.4.6 Discussion

The previous sections show how adding conditional tasks and goal increases the range of tasks that can be learned and amount of variation each task can include. However, they can be somewhat cumbersome to teach, as they require precise wording. Better natural language comprehension would help, but learning conditional steps from experience is also an attractive option. This highlights a

S Check the kitchen.

1 Go to the kitchen.

2 Turn on the lightswitch.

3 If you are in the kitchen and a plate is on the table, then move the plate into the sink.

4 If you are in the kitchen, then close the fridge.

5 If the current room is empty, then turn off the lights.

T You are done

Check reception.

(After Rosie turns on the lights)

Interrupt check.

6 If you are in reception and the pitcher is empty, then fill the pitcher with water.

7 If you are in reception, then move the pitcher onto the desk.

Continue check.

Check the lobby.

(After Rosie turns on the lights)

Interrupt check.

8 If you are in the lobby and a book is on the table, then move the book into the shelf.

Continue check.

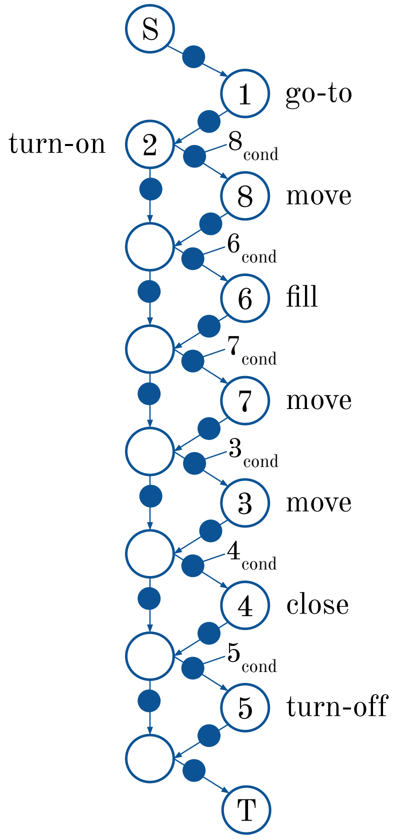


Figure 9.6: The complete set of instructions for teaching the check room task across three different interactions, and the resultant goal graph that is learned. In the second and third tasks, the instructor uses interrupt/continue to insert additional subtasks into the graph.

trade-off for interactive task learning agent research. Our work has the goal of trying to maximize the learning and generalization from a simple example. The trade-off for this priority is that the agent will sometimes make assumptions that are overly broad. For example, if Rosie is taught to store forks in a drawer, it will assume every object is stored in the drawer. Or if Rosie is taught how to check the kitchen, it will assume that checking every room will involve the same steps. An agent that does not make as many initial assumptions may not over-generalize but will require more examples. For example, the agent could ask for the goal of *store* every time and learn from experience where certain objects go. This will make the learning slower, but less prone to over-generalization. To avoid over-generalization, the aggressive learner must rely on the instructor to explicitly provide various cases. This is the approach that we have taken in this work, and the addition of conditional modifiers allows the instructor to explicitly define different task variations. This ability comes at the cost of requiring the instructor to have high expertise in understanding the agent and how it learns. Another option is to have the agent try to maximize its learning, but allow the instructor to correct it when it over-generalizes. Adding the ability to interrupt the task and insert additional subtasks is a step in this direction, but research into a more comprehensive approach to correcting and extending learned task knowledge is an important area of future work.

9.5 Repetitious Modifiers

The final type of modifier that we will discuss is one that specifies *how many times* a task should be performed. For example, suppose that the instructor wanted the agent to put four sodas on the table. She could indicate this by giving a repetition count (*'Move a soda onto the table four times.'*), a do-until loop (*'Move sodas onto the table until there are four sodas on the table.'*), or by adding a count to the referring expression (*'Move four sodas onto the table.'*). Sometimes repetitions may be implied through the use of quantifiers, such as *'Store all objects on the table.'*

In this work, we extend our goal graph representation to enable learning certain types of looping structures within a procedural task, namely those involving a fixed number of iterations, a while clause, or an until clause. In the following sections, we describe how each of these are implemented, and discuss other types of repetitious modifiers and how Rosie might be extended to learn them.

9.5.1 Learning Numeric Repetitions

The first type of repetitious modifier is where the instructor gives a fixed number of repetitions to perform by using the phrase *'for three times.'* This is interpreted as a special type of end clause, where the task is not finished until the agent has executed the task the given number of times. The agent will execute the task as normal but maintain a loop counter. When it reaches a terminal node, it will increment the loop counter and reset the current goal back to the start goal node. Once the loop counter reaches the desired number, the task will end. This approach requires no special representations or learning mechanisms, but has a significant limitation. The agent will execute a

S	Squaredance.
1	Repeat the following tasks for four times.
2	Drive forward one meter.
3	Turn right.
	Repeat.
T	You are done.

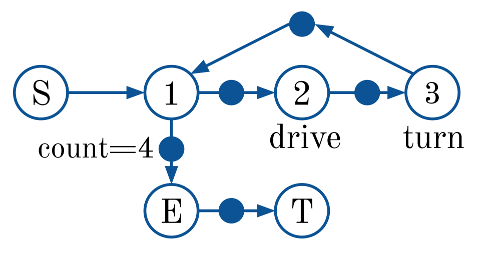


Figure 9.7: The instructions used to teach a task of driving in a square using a loop, and the resultant goal graph.

single fixed task multiple times. Any objects involved are fixed at the start of execution, so that an instruction such as *"Move a cup from the counter to the table for three times."* will cause the agent to try and move the same cup three times.

We have addressed this limitation by implementing looping structures in the goal graph. Such a loop is indicated through a repeat command such as *"Repeat the following tasks for three times."* This message is parsed and interpreted as a normal subtask, but it is handled as a special case when adding to the parent goal graph. If the subtask is *repeat*, then the agent adds two nodes to the graph, one for the repeat node and one for the loop exit. It then adds an edge between them with the loop exit conditions included. In this case, the conditions are that the loop counter equals the given number. Whenever the agent encounters a repeat node, it exits the loop if the exit conditions are satisfied, otherwise it moves to the first task in the loop and increments the counter.

Subsequent subtask instructions following the repeat command are then learned normally, until the instructor says *'Repeat'* and the agent creates an edge from the last subtask back to the repeat node. Figure 9.7 shows an example of teaching the agent to 'squaredance,' which involves driving in a square by repeating drive and turn steps four times. Using a looping structure helps overcome the fixed argument problem above, since each iteration can involve a different instantiation of a subtask. Thus the instructor could say *"Repeat the following task three times."* and *"Move a cup from the counter onto the table."* to have the agent move three cups onto the table.

9.5.2 Learning Conditional Repetitions

Another type of repetitious modifier indicates that a task should be repeated while or until some conditions are met. For example, one possible way to tell the agent to clear the table would be to say *"Store an object on the table until the table is clear."* However, this instruction presents the same problem as described the previous section, in that the reference *'an object'* would be fixed to a single object. To teach such a task to Rosie, the instructor must use a *repeat* command with an until clause, such as *"Repeat the following task until the table is clear."* This type of loop is learned in the same manner as above, except that instead of the exit conditions being a fixed count, it is the set of predicates from the until clause. The demonstration in Chapter 10 gives an example of

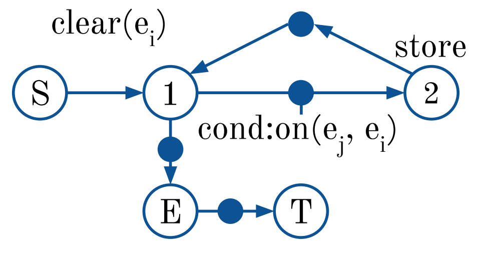
S	Clear the table.	 clear(e_i) store S 1 2 cond:on(e_j, e_i) E T
1	Repeat the following tasks while an object is on the table.	
2	Store an object on the table.	
	Repeat.	
T	You are done.	

Figure 9.8: The instructions and goal graph for teaching *'clear the table'* using a while loop.

learning this type of loop. The agent can also learn a *while* loop in the same way, except that the conditions are placed on the edge going to the first subtask and the default edge goes to the exit node. Figure 9.8 shows an example of teaching the task *"Clear the table."* using a while loop.

9.5.3 Discussion

In the previous sections, we have shown extensions to the goal graph representation to support repetitious control structures involving loops with a fixed number of iterations or a until/while clause. This helps address one of the major limitations of our work – that the agent cannot easily handle collections or sets of objects. So while we cannot use a command such as *"Store all the objects on the table."* we can teach a similar concept using a loop. Adding proper enumeration loops, such as *"Do the following tasks for each object on the table."* is beyond the scope of this work.

Parallel work within Rosie by Kirk & Laird (2019) has demonstrated the ability to learn concepts involving sets of objects for use in games, such as learning that the goal of tic-tac-toe is having three captured locations in a line, as well as other hierarchical concepts. It would be useful in the future to integrate these abilities with our task learning approach.

9.6 Evaluation

A major claim of this work is that once Rosie has learned the canonical version of a task, it can combine that task knowledge with all the modifiers listed above and execute all those variations without needing additional instruction. To demonstrate this ability, we performed an evaluation of the agent where we taught it a move task and then taught it nine other tasks that combined move with a different modifiers. The purpose of this evaluation is to showcase the many different ways a learned task can be used, and to demonstrate how our approach can generalize from a single example to all these variations.

For this evaluation, we used the Magicbot simulator environment with a simulated kitchen (Figure 9.9), similar to others used in previous chapters. One small change was made to the water cooler, where it would automatically fill a cup that was placed on it instead of requiring a button press. We ran the agent through a test script, that consisted of teaching `move` through the

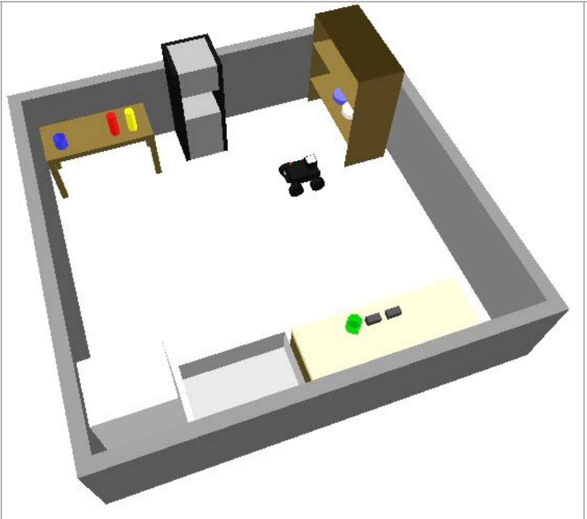
Furniture		**Objects**
- table - watercooler - pantry - counter - drawer - sink - fridge		- blue cup - green mug - blue plate - white plate - fork - spoon - ketchup - mustard

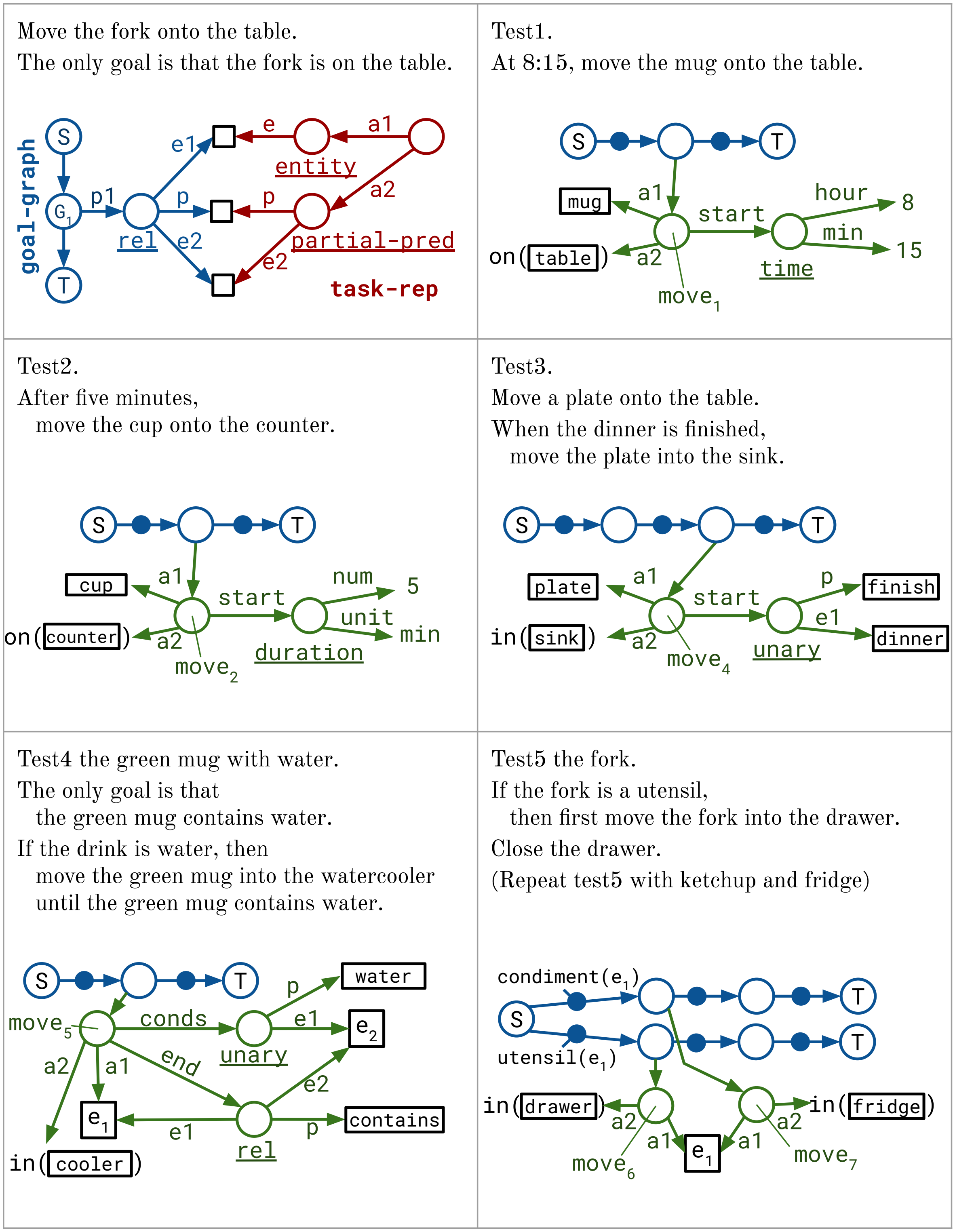
Figure 9.9: The simulated environment and the objects used in the move variations evaluation.

instructions *"Move the fork onto the table."* and *"The only goal is that the fork is on the table."* Then we taught the agent nine different tasks. For each task, we did a training interaction and then a follow-up test of the same task to verify it learned and executed it correctly. Each task was named using a generic verb (e.g. *"Test1."* or *"Test4 the fork."*). For each task, we recorded the teaching instruction, the move subtask representation, and the goal graph. These are shown in Figures 9.10 and 9.11.

Tests 1-3 combine *move* with a temporal modifier that indicates when it should begin: a time, duration, and condition predicate respectively. These three tasks only involve a linear sequence of procedural goal nodes. For Test 3, the agent waits between subtasks one and two until a person tells it that dinner is finished.

Test 4 combines move with a conditional and an end clause. The parent task has the goal of filling a cup with water, but the agent does not have a model of the water cooler. Here the agent only needs to move the cup into the cooler to have it fill with water automatically. As with the example in Table 9.3, the agent uses the end clause (*'until the green mug contains water'*) as an additional model of the move subtask. This model helps it generate a valid explanation and learn a policy for move. Test 5 is like a procedural version of the store task, where it moves the object to different places depending on the category (utensils are moved into the drawer, condiments are moved into the fridge). The conditions are embedded in the graph structure, not the subtask itself. Since the instructor uses the keyword *first*, the agent creates a permanent split in the graph.

Tests 6-9 involve repetitious modifiers. Test 6 has the agent move a mug into the pantry three times. Since the repetition clause is combined with the move instruction, the agent actually tries to move the same mug into the pantry three times. It goes through the move goal graph three times, but since the goal is already satisfied during the second and third iterations so it does not do



Move the fork onto the table.
The only goal is that the fork is on the table.

Test1.
At 8:15, move the mug onto the table.

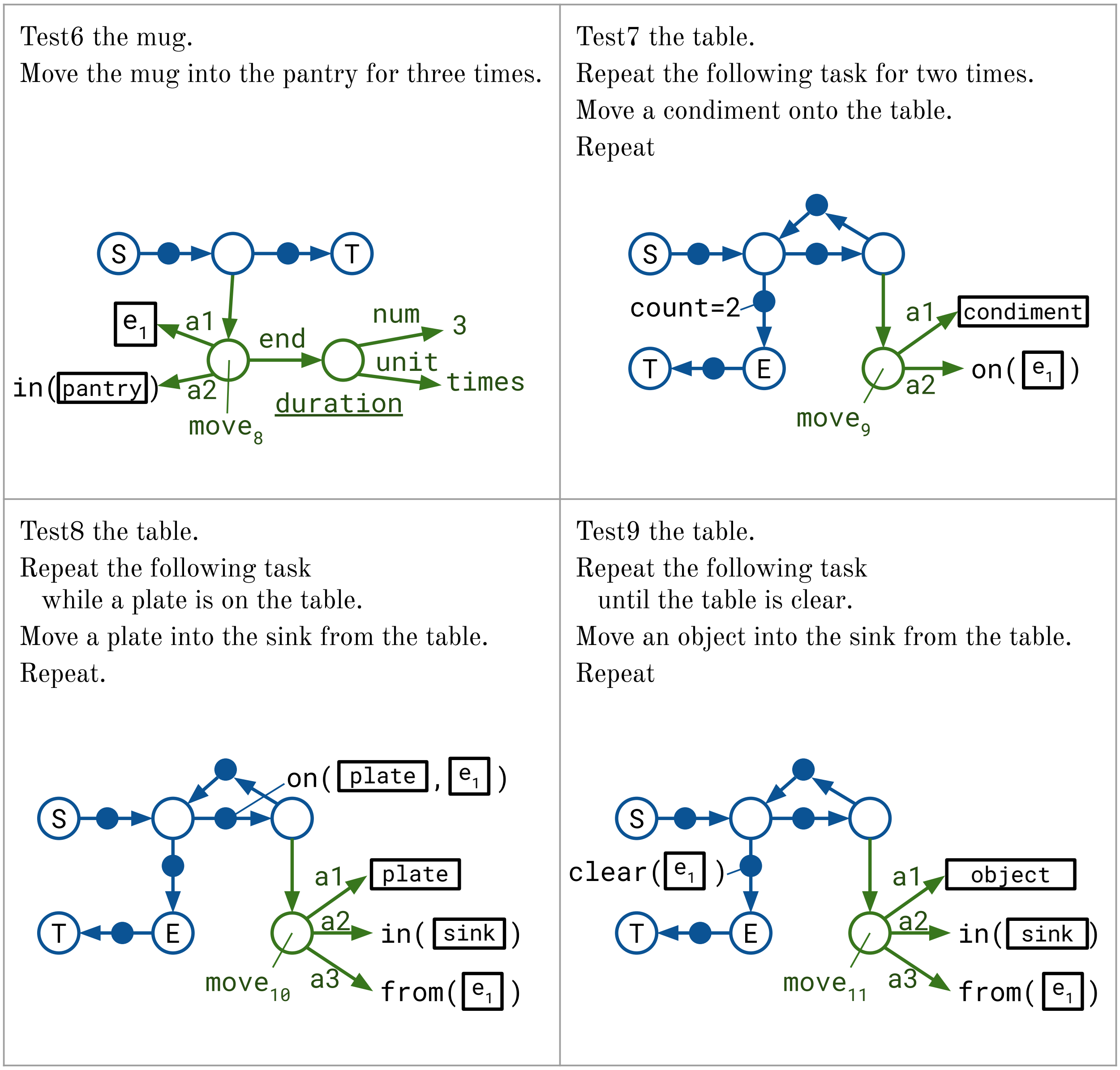
Test2.
After five minutes,
move the cup onto the counter.

Test3.
Move a plate onto the table.
When the dinner is finished,
move the plate into the sink.

Test4 the green mug with water.
The only goal is that
the green mug contains water.
If the drink is water, then
move the green mug into the watercooler
until the green mug contains water.

Test5 the fork.
If the fork is a utensil,
then first move the fork into the drawer.
Close the drawer.
(Repeat test5 with ketchup and fridge)

Figure 9.10: Results from the task modifiers evaluation for the training task and tests 1-5. Each is shown with the instructions, learned goal graph (blue), and the learned subtask structures for each move (green).



Test6 the mug.
Move the mug into the pantry for three times.

Test7 the table.
Repeat the following task for two times.
Move a condiment onto the table.
Repeat

Test8 the table.
Repeat the following task
while a plate is on the table.
Move a plate into the sink from the table.
Repeat.

Test9 the table.
Repeat the following task
until the table is clear.
Move an object into the sink from the table.
Repeat

Figure 9.11: Results from the task modifiers evaluation tests 6-9. Each is shown with the instructions, learned goal graph (blue), and the learned subtask structures for each move (green).

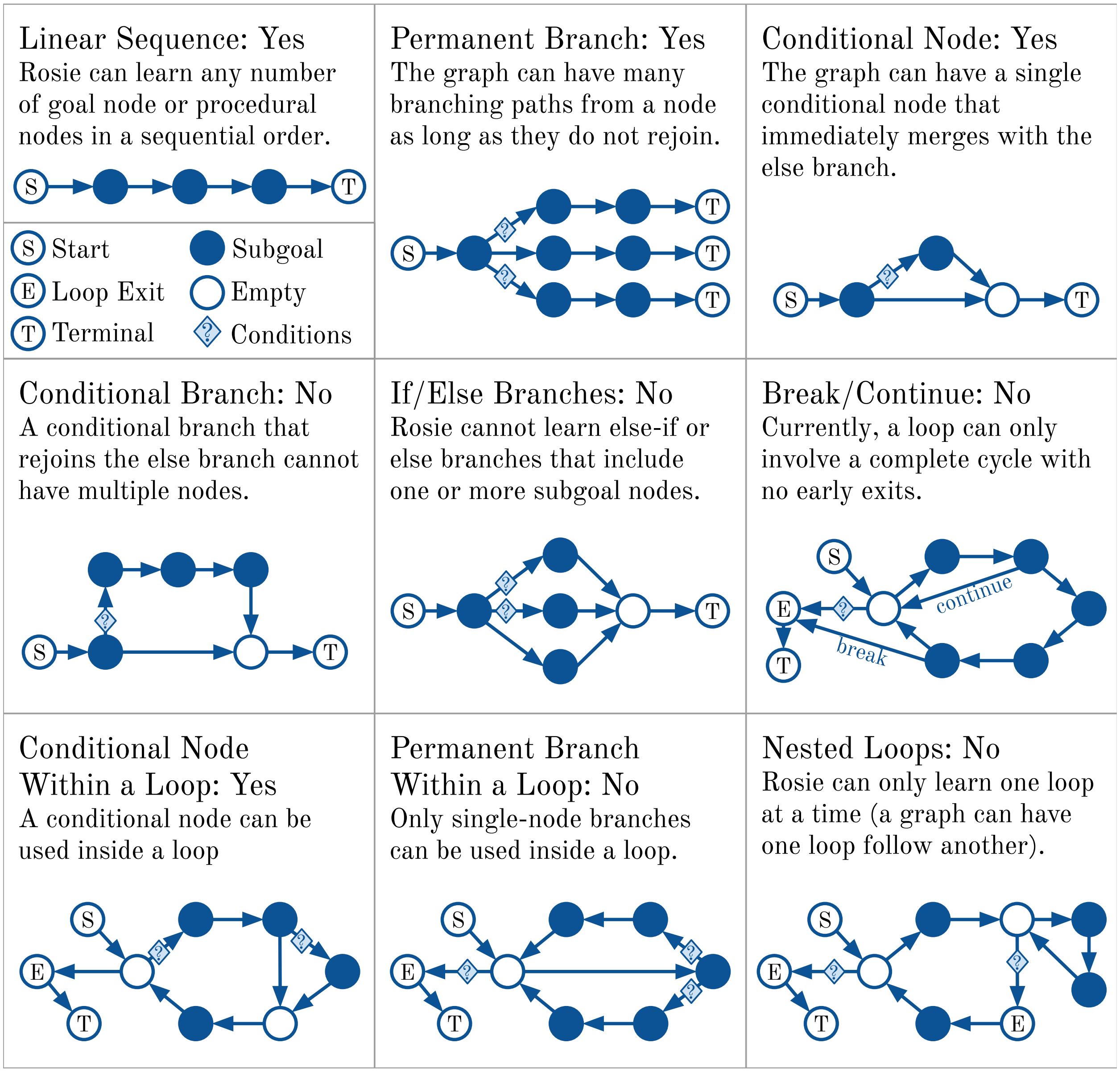
anything. Test 7 has the agent move two condiments onto the table. Since the move instruction is embedded within a proper repeat block, it actually moves two different condiments onto the table. Test 8 teaches the agent to clear all the plates off the table using a while loop, and Test 9 has the agent clear all objects off the table using an until loop. Note that these add a *from* clause to the move subtask to ensure that only objects on the table are moved.

To summarize, these results show that after a single teaching interaction the agent was able to use the learned move task as part of other tasks and combine it with nine different modifiers. It never required any additional instructions to help it perform move, and it successfully performed each of the nine tasks again when we tested each one. The fact that the agent can handle all of these different use cases from a single training example is a testament to the generalization capabilities of our approach. And the key design characteristic that enables this outcome is that the addition of new capabilities, such as handling new types of modifiers, was done in a way that is consistent with existing mechanisms and representations. This ensures that the learned task knowledge is compositional.

9.7 Analysis of Graph Structures

We conclude this chapter with an analysis of the types of goal graph structures that Rosie is currently able to learn. A selected set of examples is shown in Figure 9.12. In principle, the goal graph representation is general enough to encode the structure of almost any procedure (only considering the control flow between nodes, not what the nodes themselves can possibly represent). However, there are some significant limitations on the types of structures Rosie can learn through instruction. First, the only way for two branching paths to rejoin is if the conditional branch has only one node (top-right corner of Figure 9.12). Rosie cannot learn tasks where the conditional branch has multiple steps (center-left), or with multiple if/else branches (center). There are no limitations on branches that permanently split, so it is possible to teach an equivalent graph structure by giving the subsequent instructions multiple times. This would quickly become too cumbersome for large scale tasks. Overcoming this limitation would require giving the instructor more precise control over specifying control structures. However, these types of instructions would start looking more like programming and less like natural language instruction.

The second major limitation is that Rosie can only learn looping structures that are a single cycle with a single exit point. A loop in Rosie can include a single conditional node (bottom-right), but cannot contain major branches (bottom-middle) or nested loops (bottom-right). There are also no mechanisms to learn additional edges that would act like break or continue statements (center-right). Again, this is less a fundamental limitation of the graph representation, as the diagrams show possible ways of representing these most complex structures. The restrictions come from the current set of instructions that Rosie can understand and properly interpret. Future work could further expand the subset of graph structures that could be learned.



Linear Sequence: Yes
Rosie can learn any number of goal node or procedural nodes in a sequential order.

Permanent Branch: Yes
The graph can have many branching paths from a node as long as they do not rejoin.

Conditional Node: Yes
The graph can have a single conditional node that immediately merges with the else branch.

Conditional Branch: No
A conditional branch that rejoins the else branch cannot have multiple nodes.

If/Else Branches: No
Rosie cannot learn else-if or else branches that include one or more subgoal nodes.

Break/Continue: No
Currently, a loop can only involve a complete cycle with no early exits.

Conditional Node Within a Loop: Yes
A conditional node can be used inside a loop

Permanent Branch Within a Loop: No
Only single-node branches can be used inside a loop.

Nested Loops: No
Rosie can only learn one loop at a time (a graph can have one loop follow another).

Figure 9.12: Different types of procedural graph structures that are possible to represent and which ones Rosie can actually learn through instruction.

9.8 Discussion

In this chapter, we discussed extensions to the task representations and learning mechanisms that support learning subtasks with temporal, conditional, and repetitious modifiers. There are several capabilities that distinguish this work. The agent can apply these modifiers to a learned task without needing further instruction. It can combine these modifiers with both innate tasks and learned tasks while learning hierarchical tasks. And it can learn subtasks with temporal and conditional modifiers within both goal-based and procedural task formulations.

One issue that stands out is that the language required to teach some of these more complex control structures, such as conditions and loops, is precise and complex. This is a problem in general with learning tasks through instruction. Language is abstract and ambiguous and often does not explicitly contain all the precise control information. The more precise the instructions become, the less natural it is. The problem of how to generate code specifications from dialog was addressed by Chaurasia & Mooney (2017), and their approach heavily utilized the language to refine the learned procedure by asking questions and describing the current procedure to confirm it is what the instructor wanted. In Rosie, we could address this by making it more explainable and correctable to avoid problems of making the wrong assumption when learning control structures. Fortunately, the declarative representations are very amenable to this. We could also have Rosie initiative clarification questions, for example, if it receives a conditional it could ask something like *"What should I do otherwise?"*

CHAPTER 10

Task Learning Demonstration

A crucial requirement of our work is that all of the capabilities for learning tasks with diverse types of actions, formulations, and modifiers are integrated together and can be combined to learn complex, high-level tasks. To show all of these pieces working together, we present a demonstration of training Rosie to perform an interior guard of a barracks environment. The agent learns to perform a patrol of the building and inspect certain rooms, checking to make sure that certain people are where they are supposed to be and performing various tidying or stocking tasks. The agent is also taught how to report a fire if it should see one. This evaluation is extensive, involving fifty instructions and takes over seventeen minutes just to teach.

In the following sections, we identify that the knowledge the agent starts with and two tasks the agent was previously trained on (Section 10.1). We then go through the process of training the full guard task, reporting on what instructions are used and what knowledge is learned (Section 10.2). In the final section, we describe a follow-up test involving some varied situations to verify the agent's learning and conclude with some analysis (Section 10.3).

10.1 Prior Knowledge

The agent starts with complete knowledge of the physical layout of the barracks environment. This includes having a map of the building and knowing the names of rooms within it, knowing the names of people and being able to recognize them, and being able to recognize all of the objects/furniture involved in the task. An image of the full barracks environment is shown in Figure 10.1. Rosie also starts with knowledge of the innate tasks described in Chapter 7. We also taught Rosie two additional tasks through instruction that are used in the demonstration: *fetch* and *ensure*.

Figure 10.2 shows the instructions for teaching the task *"Fetch a radio from the eastern sentry post."* The purpose is for the agent to go find the indicated object and bring it back to wherever it started. We use the phrase *starting location* in the goal description to make the task more general, if we had referred to a specific location then it would be fixed to that particular place in the goal. However, the agent does not know any semantics of the word *starting*, and cannot ground the referring expression to any known location in the environment. We tell it to *"First, remember the*

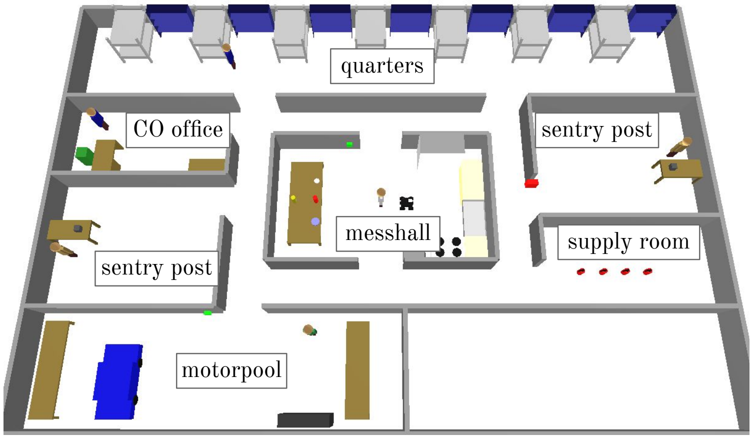


Figure 10.1: A map of the barracks environment used in the guard demo.

current location as the starting location." This is the first step the agent does during each fetch task and ensures that it correctly brings the object back to where it started. This task demonstrates the utility of having innate tasks that involve mental actions.

The second pre-taught task is *ensure*. In this task, the agent must verify that a specific person or thing is present in the room. If not, the agent will report it to the commanding officer (CO). The instructions and learning process is shown in Figure 10.3. It is taught as a straightforward conditional task with two cases. If the entity is present (during training, a sentry was in the room) then the agent immediately finishes the task. Otherwise (during training, a dispatcher was not in the room) the agent finds the CO and reports the missing entity and location. These two tasks are the only extra ones that the agent starts with (the knowledge is pre-loaded before running the evaluation).

10.2 Training

Once the agent has all of the domain and task knowledge listed above, it is ready for the main training interaction for the guard task. The following interactions all occur in one ongoing teaching episode, there are no breaks or restarts. We first teach it two tasks: how to inspect a sentry post and how to report a fire. We then move on to the main guard task, where the agent first learns the name of the relieving officer, then cycles between inspecting three locations: the messhall, the eastern sentry post, and the motorpool. We interrupt the inspect task to teach it additional steps specific to the messhall and the motorpool. The agent executes two complete patrol cycles and then finishes the guard task when it sees the relieving officer.

Dialog	**Learning**	**Execution**
Fetch a radio from the eastern sentry-post.	G_1: start	$\mathtt{fetch}(e_{radio},\mathtt{from}(e_{ESP}))$
What is the next goal or subtask of fetch?		
The only goal is that the radio is in the starting location.	G_2: in(e_{arg1}, e_{start}) G_3: terminal	
What do I do next for fetch?		
First, remember the current location as the starting location.	subtask rem_1	$\mathtt{rem}_1(e_{cur},\ e_{start})$
(Rosie finishes the task using planning)	subtask go-to$_2$	$\mathtt{go\text{-}to}_2(e_{ESP})$
	subtask pick-up$_3$	$\mathtt{pick\text{-}up}_3(e_{radio})$
	subtask go-to$_4$	$\mathtt{go\text{-}to}_4(e_{start})$
	subtask put-down$_5$	$\mathtt{put\text{-}down}_5(e_{radio})$

Figure 10.2: The instructions, learning events, and executed actions that occurred while teaching the fetch task.

Dialog	**Learning**	**Execution**
Ensure a sentry is present.	G_4: start	$\mathtt{ensure}(e_{sen},p_{pres})$
What is the next goal or subtask of ensure?		
If a sentry is present, then finish the task.	G_5: terminal	
Ensure a dispatcher is present. *		$\mathtt{ensure}(e_{disp},p_{pres})$
Remember the current location as the starting location. *	G_6: subtask rem_6	$\mathtt{rem}_6(e_{cur},\ e_{start})$
Find the CO. *	G_7: subtask $find_7$	$\mathtt{find}_7(e_{CO})$
Describe the dispatcher.	G_8: subtask $desc_8$	$\mathtt{desc}_8(e_{disp})$
The dispatcher *		
Say “is not present”.	G_9: subtask say_9	$\mathtt{say}_9(e_{sent1})$
is not present *		
Describe the starting location.	G_{10}: subtask $desc_{10}$	$\mathtt{desc}_{10}(e_{start})$
the motorpool *		
Go to the starting location. *	G_{11}: subtask go-to$_{11}$	$\mathtt{goto}_{11}(e_{start})$
You are done.	G_{12}: terminal	
*Omitted *What is the next goal or subtask of ensure?*		

Figure 10.3: The instructions, learning events, and executed actions that occurred while teaching the ensure task. Asterisks in the left column indicate places where Rosie asked for the next step in the task.

Dialog	Learning	Execution
Inspect the eastern sentry-post.	G_{13}: start	`inspect`(e_{ESP})
What is the next goal or subtask of inspect?		
Go to the eastern sentry post. *	G_{14}: subtask go-to$_{12}$	`go-to`$_{12}$(e_{ESP})
If the lightswitch is off,	G_{15}: subtask turn-on$_{13}$	
then turn the lightswitch on. *	G_{16}: none	
If you are in a sentry-post and an	G_{17}: subtask fetch$_{14}$	`fetch`$_{14}$(e_{ext}, e_{SR})
extinguisher is not present, then fetch an	G_{18}: none	
extinguisher from the supply room. *		
If you are in a sentry-post,	G_{19}: subtask ensure$_{15}$	`ensure`$_{15}$(e_{sen}, p_{pres})
then ensure a sentry is present. *	G_{20}: none	
If the current location is empty,	G_{21}: subtask turn-off$_{16}$	
then turn off the lightswitch. *	G_{22}: none	
You are done.	G_{23}: terminal	
*Omitted *What is the next goal or subtask of raise?*		

Figure 10.4: The instructions, learning events, and executed actions that occurred while teaching the initial inspect task. Note that the lights were already on and a person was present, so the turn-on/turn-off subtasks were not executed.

10.2.1 Inspect

The first training interaction teaches the agent the *inspect* task by telling it to *"Inspect the eastern sentry post."* (Figure 10.4). The inspect task involves different actions depending on the room. It is taught as a linear procedural task, with conditional steps along the way that test the particular room. The first steps are always to go to the room and turn on the lights, and the last step is always to turn off the lights if the room is empty (no people are present). In this training interaction, the agent also learns two actions specific to a sentry post: fetch a fire extinguisher if one is not in the room and ensure that there is a sentry present. For each conditional subtask, the agent creates a conditional goal node for that subtask and an empty default goal node to serve as the else branch. Note that during this training, the lights were already on, a sentry was present, but there was no fire extinguisher. These conditions determine which of the subtasks were executed during this first run-through.

10.2.2 Raise a Fire Alarm

The second interaction teaches the agent how to report a fire. The task is taught as a linear sequence of steps (Figure 10.5), where the agent goes to the eastern hallway, turns on the fire alarm, and then tells the CO the location of the fire. The agent does not know any specific semantics for the

Dialog	Learning	Execution
Raise a fire-alarm.	G_{24}: start	raise(e_{fa})
What is the next goal or subtask of raise?		
Remember the current location as the emergency location. *	G_{25}: subtask rem_{17}	rem_{17}(e_{cur}, e_{emer})
Go to the eastern hallway. *	G_{26}: subtask go-to_{18}	go-to_{18}(e_{EH})
Turn on the alarm. *	G_{27}: subtask turn-on_{19}	turn-on_{19}(e_{alarm})
Say “There is a fire.” to the CO.	G_{28}: subtask say_{20}	say_{20}(e_{sent2}, e_{CO})
Can you help me find the CO?		
The CO is in the CO’s office.		
There is a fire. *		
Describe the emergency location.	G_{29}: subtask desc_{21}	desc_{21}(e_{emer})
The eastern sentry-post. *		
Go to the emergency location. *	G_{30}: goto_{22}	goto_{22}(e_{emer})
You are done.	G_{31}: terminal	
*Omitted *What is the next goal or subtask of raise?*		

Figure 10.5: The instructions, learning events, and executed actions that occurred while teaching the raise alarm task.

phrase *emergency location*, instead, as before, the instructor uses the remember task to connect the term *emergency* to a particular location. This interaction also shows an example of the agent using planning within an innate subtask. The task *“Say ‘There is a fire.’ to the CO.”* is not immediately achievable since the CO is not in the room. The agent performs a search and decides to propose the subtask find(e_{CO}). The agent has no prior knowledge of where the CO might be, so it asks the instructor. It then drives to the CO’s office and completes the rest of the task. Note that this all happens within the *say* subtask, so the details are abstracted away from the perspective of the *raise* task.

After the task is taught, we tell the agent that *“Whenever you see a fire, then raise a fire-alarm.”* This is learned as an opportunistic task as described in Section 9.4.4. Rosie creates an imagined world state where a fire is visible, and then learns a proposal rule to initiate raise($e_{fire-alarm}$). Later on, we will see an example of it interrupting the current task to report a fire (Section 10.3).

10.2.3 Guard

Finally, the agent is ready to learn the guard task (Figure 10.6). The main purpose is to repeatedly inspect three locations until ‘relieved.’ Of course, Rosie has no concept of what it means to be relieved, so we teach that by having Rosie ask who the relieving officer is and remembering it for

Dialog	**Learning**	**Execution**
Guard the barracks.	G_{32}: start	$\mathtt{guard(e_{barr})}$
What is the next goal or subtask of guard?		
Ask “Who is my relieving officer?”	G_{33}: subtask ask$_{23}$	$\mathtt{ask_{23}(e_{sent3})}$
Who is my relieving officer?		
Officer Mercer. *		
Remember the answer as the relieving officer. *	G_{34}: subtask rem$_{24}$	$\mathtt{rem_{24}(e_{ans},\ e_{rel})}$
Repeat the following tasks	G_{35}: repeat	
until the relieving officer is present. *	G_{36}: loop exit	
Inspect the messhall. *	G_{37}: subtask inspect$_{25}$	$\mathtt{inspect_{25}(e_{MH})}$
(Interrupts task to add steps - Section 2.4)		
Inspect the eastern sentry-post. *	G_{48}: subtask inspect$_{34}$	$\mathtt{inspect_{34}(e_{ESP})}$
Inspect the motorpool. *	G_{49}: subtask inspect$_{35}$	$\mathtt{inspect_{35}(e_{MP})}$
(Interrupt task to add steps - Section 2.5)		
Go to the CO’s office. *	G_{52}: subtask go-to$_{39}$	$\mathtt{go\text{-}to_{39}(e_{CO\text{-}OF})}$
Repeat.		
(Relieving officer is present, loop ends)		
What is the next goal or subtask of guard?		
You are done.		
*Omitted *What is the next goal or subtask of guard?*	G_{53}: terminal	

Figure 10.6: The instructions, learning events, and executed actions that occurred while teaching the raise alarm task.

Dialog	Learning	Execution
Interrupt inspect.		
What is the next goal or subtask of inspect?		
If you are in the messhall and a plate is on the table, then move the plate into the sink.	G_{38}: subtask move_{26} G_{39}: none	$\texttt{move}_{26}(\texttt{e}_{\texttt{pl}},\texttt{in}(\texttt{e}_{\texttt{sk}}))$
What is the next goal or subtask of move?		
The only goal is that the plate is in the sink.	**move** G_{40}- G_{42} subtasks 27,28	$\texttt{pick-up}_{27}(\texttt{e}_{\texttt{pl}})$ $\texttt{put}_{28}(\texttt{e}_{\texttt{pl}},\ \texttt{e}_{\texttt{sk}})$
What is the next goal or subtask of inspect?		
If you are in the messhall and a condiment is on the table, then store the condiment.	G_{43}: subtask store_{29} G_{44}: none	$\texttt{store}_{29}(\texttt{e}_{\texttt{cond}})$
What is the next goal or subtask of store?		
If the object is a condiment, then the only goal is that the condiment is in the fridge and the fridge is closed.	**store** G_{45}- G_{47} subtasks 30-33	$\texttt{open}_{30}(\texttt{e}_{\texttt{fr}})$ $\texttt{pick-up}_{31}(\texttt{e}_{\texttt{ket}})$ $\texttt{put}_{32}(\texttt{e}_{\texttt{ket}},\ \texttt{e}_{\texttt{fr}})$ $\texttt{close}_{33}(\texttt{e}_{\texttt{fr}})$
What is the next goal or subtask of inspect?		
Continue inspect.		

Figure 10.7: The instructions, learning events, and executed actions that occurred when interrupting the inspect task to teach steps specific to the messhall: move and store.

later. Then, the repeat statement includes the until clause *'until the relieving officer is present.'* The last step before the repeat has the agent drive to the CO's office, so this check will occur there (if the agent sees the officer somewhere else it will not stop the task).

The first repeated step is to inspect the messhall, and while the agent is executing that task the instructor interrupts the agent and gives it additional steps to perform in the messhall (Section 10.2.4). The second repeated step it to inspect the eastern sentry-post, which it already learned how to do. The third step is to inspect the motorpool, where again the instructor will interrupt and teach additional steps (Section 10.2.5). The fourth and final step has the agent drive back to the CO's office, after which the task will repeat if the relieving officer is not present. During the training, at this point we move the location of Officer Mercer in the simulator to the CO's office so the task will end.

10.2.4 Inspect the Messhall

A major requirement for the inspect task is that the agent performs different actions depending on the room being inspected. Without a way to modify the task later, the instructor would have to give all possible steps during the first teaching interaction. However, as described in Section 9.4.5, we have implemented a mechanism to interrupt a procedural task and insert additional steps. When the agent first inspects the messhall, as it is turning on the light the instructor gives the command *'Interrupt inspect.'* The agent will finish the current subtask, and then stop and ask for the next instruction (Figure 10.7).

The instructor gives two additional commands specific to the messhall, moving a plate from the table into the sink and storing a condiment on the table. In this version, the agent would only perform this once per inspect task, even if there were multiple plates or condiments. Note that neither of these tasks had been taught before, so the agent asks for their goals. This is an example of how the instruction is mixed-initiative. Even though most of the time Rosie just asks for the next instruction, if it is missing some knowledge (such as the goal of move) it can initiate a new interaction to acquire it. This removes some of the burden on the instructor to model exactly what the agent knows and does not know. Once the two subtasks have been completed, the instructor tells the agent to continue the inspect task.

10.2.5 Inspect the Motorpool

A similar process repeats for teaching how to inspect the motorpool (Figure 10.8). Again, the instructor interrupts the task as the agent goes to turn on the light. The only novel point to make in this interaction is that Rosie learns a composite goal that is triggered by the phrase *'complete the following tasks.'* This represents the goal of completing three subtasks in any order: ensuring a sentry and dispatcher are present and locking the vehicle. These three are all added to a single node in the goal graph. After executing all three, it finishes the inspect task and continues learning guard. This is the final instruction given to the agent. In the next section we demonstrate what the agent learned through a testing run-through.

10.3 Testing

Once the agent was fully trained through the above interactions, we ran a verification test to ensure that it could execute the learned task correctly. We had it repeat the guard task and modified the simulation environment throughout the test to have it encounter novel scenarios. The top-level execution and environmental changes are shown in Figure 10.9. These changes tested whether the agent could handle situations that did not appear during training, including novel objects (Officer Williams instead of Mercer, mustard instead of ketchup), missing people (removed a sentry and dispatcher), relocated people (moved the CO to a different location), and added a fire in the motorpool. The agent successfully completed all the tasks, including interrupting the guard task to report the fire, without needing any additional instruction.

The main purpose of this demonstration is to show how the agent can learn a high-level, long-running task composed of an extensive hierarchy and dependent on subtasks that themselves are learned. This single guard task took 27.8 minutes of real time and involved executing 389 different subtasks. Most of the time was spent waiting while performing driving commands in the simulator.

Dialog	**Learning**	**Execution**
Interrupt inspect.		
What is the next goal or subtask of inspect?		
If you are in the motorpool,	G_{50}: composite	
then complete the following tasks.	G_{51}: none	
Give me another task or say 'You are done'.		
Ensure a sentry is present. *	G_{50}: +subtask ensure_{36}	$\texttt{ensure}_{36}(\texttt{e}_{\texttt{sentry}})$
Ensure a dispatcher is present. *	G_{50}: +subtask ensure_{37}	$\texttt{ensure}_{37}(\texttt{e}_{\texttt{disp}})$
Lock the vehicle. *	G_{50}: +subtask lock_{38}	$\texttt{lock}_{38}(\texttt{e}_{\texttt{veh}})$
You are done.		
What is the next goal or subtask of inspect?		
Continue inspect.		
*Omitted *Give me another task or say 'You are done'*		

Figure 10.8: The instructions, learning events, and executed actions that occurred when interrupting the inspect task to teach steps specific to the motorpool.

	Training	**Testing**
Time	17.1 minutes	27.8 minutes
Executed Tasks	224	389
Learned Chunks	676	23

Table 10.1: Broad statistics for the training and testing interactions.

The agent achieved all of this without asking for additional help, showing that it could at least apply the learned task knowledge and even generalize it to account for some variations. Table 10.1 shows that it did learn a few more rules during the testing task, but that most of the learned chunks from the training were sufficient. The final learned chunks included subtask proposal rules (54), policy rules (40), action models (4), and goal elaboration (120) and selection (213) rules.

Figure 10.10 shows the full task decomposition for the guard and raise alarm tasks. This hierarchy was fully learned during the training session, except for the portion in the bottom-left box, which includes innate task knowledge. This guard task was chosen so as to demonstrate now most of the capabilities from this thesis can be utilized in a single task. All five types of actions are used throughout, including different uses of communicative (say, ask) and mental (remember) actions. The agent learned both goal-based tasks (fetch, store, move) and procedural tasks (guard, raise, inspect). It included repetitious and conditional clauses, and involved an opportunistic task (raise fire alarm). It also showed the instructor interrupting the inspect task to add additional actions. Taken together, these represent significant advances in the scope and breadth of tasks that the agent can learn.

Dialog and Execution	**Changes to Environment**
Guard the barracks.	
Who is my relieving officer?	
Officer Williams.	
[Inspects messhall]	placed plate on table
[Inspects eastern sentry post]	removed sentry
[Inspects motorpool]	moved CO to messhall, added fire to motorpool
[Interrupts to raise fire alarm]	
[Goes to CO's office]	placed mustard on table
[Inspects messhall]	removed fire extinguisher, added sentry
[Inspects eastern sentry post]	removed dispatcher, unlocked vehicle
[Inspects motorpool]	moved Williams to CO's office
[Goes to CO's office]	
I am ready for a new task.	

Figure 10.9: The top-level subtasks executed during the guard test and the changes made to the simulator before each step.

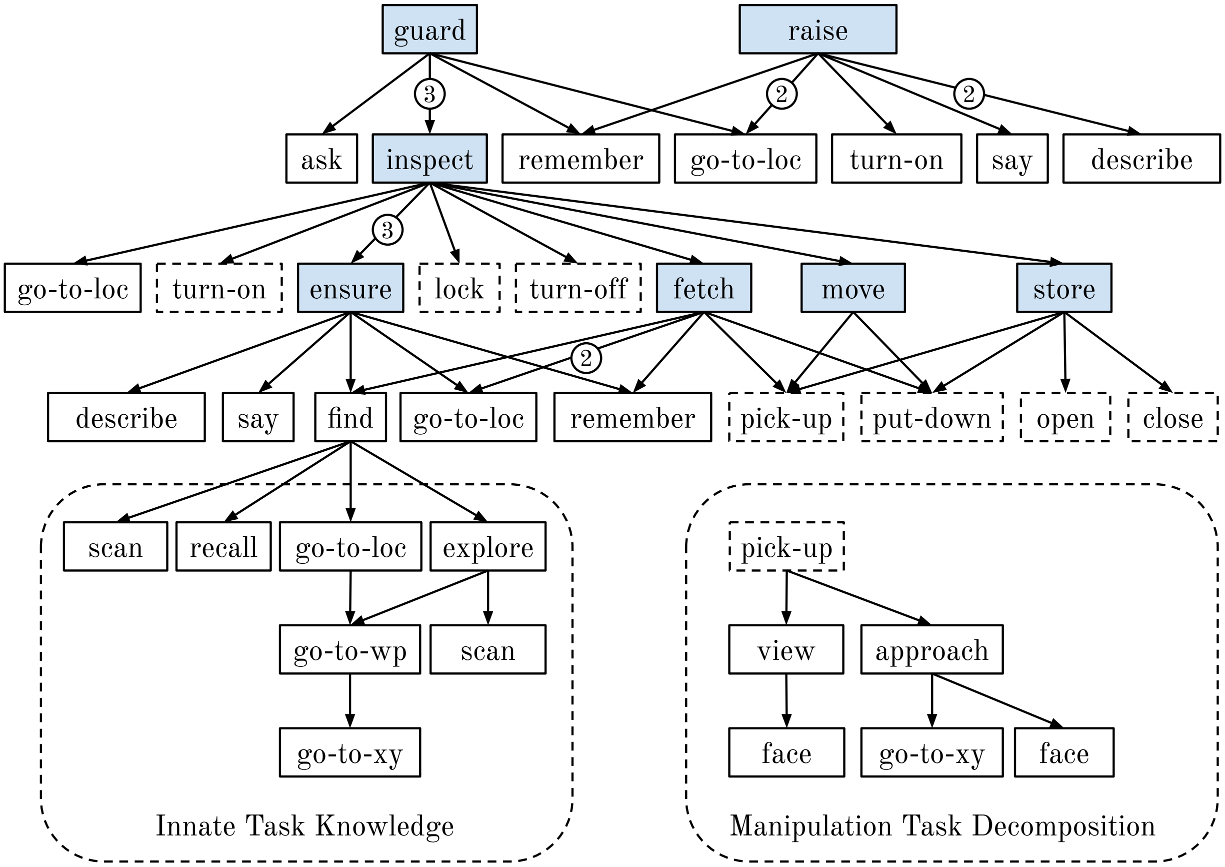


Figure 10.10: The full task decomposition learned for the guard and raise alarm tasks. Blue rectangles indicate learned tasks. All the dashed rectangles indicate innate manipulation tasks that all share the same task decomposition (that is learned from experience). Some tasks have multiple versions of a subtask, indicated by a circled number on the edge.

CHAPTER 11

Conclusion

In this work, we greatly expanded the space of tasks learnable by an agent through one-shot situated interactive instruction. This was achieved by identifying three dimensions of variation – diverse types of actions, diverse formulations, and diverse task modifiers – and extending the capabilities of the task learning agent Rosie along those dimensions. These extensions were done in a unified manner to remain consistent with the core research goal of one-shot learning from mixed-initiative human-agent dialog.

Through this work, we made contributions to the field by providing answers to the following research questions:

1. *What is a useful characterization of the space of learnable tasks for ITL?*

   The notion of a task is a very broad one and covers an enormous range of activity. The boundary of exactly what is and is not a task is fuzzy, but some primary characteristics were proposed by Wray III et al. (2019): that they achieve, maintain, or perform some type of goal; that they involve multiple steps (non-atomic); that they operate on the order of minutes to hours; and that they are instructable. These characteristics encompass a vast range of possible tasks. We identified several dimensions of task learning diversity that can be used to measure the scope of tasks within this space that an ITL agent can learn: diverse types of actions that can be used (Chapter 7), diverse ways that a task can be formulated (Chapter 8), and diverse ways that a single task can be modified (Chapter 9).

   This characterization is useful in two main ways. First, it can be used as a guide to identify areas of future work and ways to expand the capabilities of an agent. Second, it can serve as a method for comparing different ITL agents, as we did in Chapters 7, 8, 9. Once challenge in the research area of ITL is the lack of common data sets, environments, or benchmarks. Since each approach likely uses different embodiments, teaching methods, and settings, it is unlikely that such tools will be adopted by a broad group. We propose that these three dimensions are a useful way to compare the space of tasks that are learnable by different ITL approaches.

2. *How can diverse types of primitive actions be integrated into an ITL agent?*

   Since tasks are ultimately composed from individual skills or actions that the agent can perform, one major factor that determines the space of tasks an ITL agent can ultimately learn is the diversity of available primitive actions. Much of the ITL research is concerned with agents that learn physical manipulation and navigation tasks, sometimes with perceptual actions either implicitly or explicitly represented. In Chapter 7, we identified five types of actions: manipulation, movement, perceptual, communicative, and mental, and implemented multiple primitive actions of each type within our agent. One particular contribution was implementing mental and communicative actions in a way that was consistent with our approach. This meant not only developing ways of representing and executing these actions, but developing a novel approach to learning proposal rules and action models that allowed mental and communicative actions to be used during planning and explanation-based policy learning.

   Another contribution was unifying the representations for innate and learned tasks so that the existing planning and learning mechanisms could be used within innate tasks themselves. This significantly improved the flexibility and adaptability of the agent within complex, partially observable environments without requiring a priori knowledge or additional instruction. This also allowed the preconditions for innate tasks to be less specific, which simplified the planning problem for higher-level tasks.

3. *How can an ITL agent learn tasks with diverse formulations?*

   The incredible breadth of possible tasks suggests that no single representation, learning mechanism, and method of execution is likely to be sufficient for all tasks. A truly general agent should be able to formulate a task in different ways, depending on the characteristics of the task and how it was taught. Different formulations will have various trade-offs and some will be more suitable than others for a given task. In Chapter 8, we identified three major categories of formulations: achieving some goal state, following a procedure, and maximizing some objective. Most related work focuses on learning only one of these formulations. We demonstrated how our approach to ITL could be extended to learn both goal-based tasks and procedural tasks, including maintenance and composite tasks, using a novel goal graph structure that can represent both formulations. The key contribution was doing this in a unified way that allowed for blending between the two formulations and increased the overall flexibility of the agent. Two examples were leveraging planning within procedural tasks (Section 8.4.1) and learning procedures within goal-based tasks when explanation failed (Section 8.4.2).

4. *How can an ITL agent generalize task knowledge across diverse variations?*

   The previous question addressed variations between different types of tasks. However, there can also be significant variation in the use of a single task. This includes different goals or procedures based on the arguments used, or additional constraints given by modifying clauses on the task instruction. Ideally, a task learning agent should be able to transfer knowledge between different task variations while requiring minimal additional instruction. In Chapter 9, we categorized different types of modifiers that can be used with tasks and described extensions to the task and goal graph representations that enabled the agent to learn certain temporal, conditional, and repetitious modifiers. Other work has included these modifiers (Meriçli et al., 2013; Suddrey et al., 2016), but these used purely procedural task graph structures. Our contribution is showing temporal and conditional structures being used within both goal-based and procedural task formulations. In addition, we demonstrate significant generalization from a single task instruction to nine different variations (Section 9.6).

5. *What organization of knowledge and processing supports one-shot ITL?*

   In Chapter 2, we characterized the learning problem and identified desiderata for an agent that learns new tasks through one-shot instruction – that the learning should be online, real-time, incremental, compositional, efficient, and general. These requirements are difficult to satisfy, made even harder by the desire to support a wide range of diverse types of tasks. Our approach uses relational representations, domain knowledge, and explanation-based generalization to perform rapid task learning. Processing is organized into two stages. First, knowledge from instructions is used to construct declarative structures. Second, those structures are interpreted within the current situated context which compiles it into procedural rules. The novel aspect of our approach is that the intermediate structure is completely embedded in the rules, and no longer needs to be present in working memory.

We addressed these questions through a complete, integrated agent implemented in the Soar cognitive architecture that demonstrated end-to-end behavior in real-world and simulated environments. This included integrating the task learning capabilities with natural language comprehension and interaction management (Chapter 4) and perceptual understanding (Chapter 5). By expanding the task learning capabilities of Rosie along the three dimensions, we have produced an agent able to learn a wider range of high-level tasks from one-shot situated interactive instruction than any other robotic agent that we know. However, this rapid learning and generalization does have a number of trade-offs that we discuss in the following section.

11.1 Limitations and Future Work

The focus of this work has been on learning high-level tasks from single examples. Our use of symbolic representations and explanation-based mechanisms were chosen for their support of rapid learning and extensive generalization. This comes at the cost of being less robust to uncertainty and noise. Although we have done some work with handling perceptual errors (Chapter 5), for the most part the agent assumes accurate perception. A major area of research is how to combine high-level reasoning and representations with low-level perception, and Rosie could well benefit from advances in that area.

Handling uncertainty, both in perception but also task knowledge, is a significant challenge that will need to be addressed. Soar does not natively represent uncertainty, but perceptual representations could be extended to include confidence values. In previous work (Mohan et al., 2012), multiple possible values with associated confidences were provided for perceptual features, such as color and shape. Rosie then used hand-coded thresholds to determine if it could be confident in a specific value for that feature, and if not, would ask for additional training. This same approach could be used for categories, where a classifier could output multiple category labels with confidences. Although it would likely be impractical to implement full Bayesean reasoning in Soar, it could use the weights to break ties when choosing an object from a set. For example, if it needed to pick up a mug, it could choose the one with a higher likelihood.

For task knowledge, Rosie assumes very little uncertainty in order to perform aggressive generalization. Once it learns a particular goal or procedure, it will immediately apply that to new objects. We currently rely on the instructor to explicitly teach task variations. One way to relax this requirement is to add the ability to correct the agent when it overgeneralizes. Another way to actually address uncertainty would be to allow a task to have different variations depending on the types of arguments. In this case, there might be multiple task concept networks corresponding to a verb. Then, instead of entity slots being completely blank placeholders, they could contain information about the types of entities. For example, consider the *store* task, which was taught with multiple conditional goals (*"If the fork is a utensil, then the goal is that the fork is in the drawer."*). An alternative approach would be to have multiple TCN's for store, each for a different object category (utensils, drinks, etc.) Then when a new command is given, the agent could select the best match given a measure of entity similarity (e.g., using Semantic Relatedness, such as Budanitsky & Hirst (2006)) or decide to learn a new one.

Another crucial issue that we do not address is scalability. Rosie can handle ten or twenty objects in the world model, but in real-world cluttered environments there might be hundreds. Since the forward search is exponential with the number of objects that can be interacted with, it does not scale to these numbers. Therefore, improvements are needed for determining which objects are important to the current task and restricting attention to those.

As mentioned in Chapter 7, the agent can only learn tasks that are hierarchical compositions of known primitive actions; it cannot learn new ones. Currently, Rosie sends high-level commands to a robot controller that executes them. One possible extension would be adding the ability to train new behaviors using demonstrations. The learning would most likely occur outside the Soar agent but still be directed by it.

Another major area of needed improvement is the quality of the instruction. Although Rosie does learn from natural language instruction, it is a long way from allowing a novice, untrained user to instruct the agent. The instruction could be improved in a number of ways. First, better natural language comprehension could allow for more sentence variation. Essentially this would allow many similar sentences to map to the same internal representation, so that no modifications to task learning would be required. Second, there are not many existing mechanisms in Rosie that support its ability to explain its behavior to an instructor or correct its behavior through instruction or experience. The former would help an instructor obtain a better understanding of what the agent knows, and the latter would help the agent to recover if the instructor made a mistake. Since Rosie uses symbolic, declarative representations, it would be relatively easy to add these features compared to more opaque machine learning approaches such as neutral networks. The agent also assumes that each sentence is complete and can stand alone, whereas natural instruction is filled with sentence fragments that help maintain common ground and the state of the interaction. This work also sometimes relies on very precise ways of phrasing certain commands. Again, this would be helped by better natural language comprehension, but more common sense reasoning or robust message interpretation would also be useful. It would be productive to have the results of HRI studies that examine how people naturally try to instruct an agent such as Rosie. For example, Ramaraj et al. (2019) looked at making Rosie more explainable to make instruction easier.

One broader outcome of this work is the development of an approach for learning and structuring procedural task knowledge within a cognitive architecture. It is common for agents in a cognitive architecture such as Soar to represent all the task execution knowledge using only procedural rules. However, we use a hybrid approach where task knowledge given by the instructor is first stored as a general declarative representation (the task concept network), then compiled into rules by interpreting the knowledge within the current context. This has a number of benefits. First, the agent can actually analyze the declarative task knowledge to do things such as generalize arguments that are shared between the task and subtasks or turn goals into action models. It also makes it easier to modify the existing task knowledge. Both of these are very difficult with procedural rules, since they cannot be tested or modified.

This form of declarative task knowledge could be a way of allowing agent designers to program behavior at a higher-level than rules. One could imagine designing special interfaces to allow users to view, modify, and create TCN's and load the changes into an agent. Thus it is possible for the approach to agent design described in this thesis to be more generally applicable than just interactive task learning.

11.2 Summary

Interactive task learning provides a vision of giving people the power to direct, customize, and extend the behavior of artificial agents using natural methods of interaction. Supporting this ability across a range of environments and breadth of tasks is an extremely challenging problem, requiring the integration of a number of difficult AI research areas. This thesis takes a step in increasing the scope and diversity of tasks that can be learned through situated interactive instruction. We demonstrate an integrated, unified approach that displays some novel capabilities including learning goal-based tasks mental and communicative actions, learning tasks that blend goal and procedural formulations, and combining a procedural goal graph and problem-space formulations. We hope that this dissertation inspires others to continue to push the bounds of learnable task diversity and shows the power of utilizing instruction for learning.

APPENDIX A

Soar Representations

Throughout the examples of task knowledge in this dissertation, we have relied on abstractions over the actual Soar representations and implementation details. In this appendix chapter, we provide examples of the actual Soar structures that are used in Rosie.

A.1 World

Chapter 5 details how the agent constructs the world representation used during task learning and execution. It consists of entities, unary predicates (properties), and binary predicates (relations). Below, in Figure A.1, we show a subset of the predicates from Figure 5.1 and the corresponding Soar structures in working memory. Note that due to legacy reasons, the representations use the term ‘object’ instead of the more general ‘entity’ used in this thesis. We often append the number 1 to a symbol to differentiate it from the English word.

Predicates	Soar Representations

	`(<top-state> ^world <world>)`
	`   (<world> ^objects <objs>  ^predicates <preds>  ^robot <robot>)`

sink(e_1) furniture(e_1) drain(e_1) receptacle(e_1) not-visible(e_1)	`(<objs> ^object <e1>)`
	`(<e1> ^handle e1  ^item-type object  ^predicates <e1-preds>)`
	`(<e1-preds> ^category sink1 furniture1`
	`            ^affordance1 drain1  ^affordance1 receptacle1`
	`            ^is-visible1 not-visible1)`

person(e_4) mary(e_4) reachable(e_4)	`(<objs> ^object <e4>)`
	`(<e4> ^handle e4  ^item-type object  ^predicates <e4-preds>)`
	`(<e4-preds> ^category person`
	`            ^name mary1  ^is-reachable1 reachable1)`

kitchen(e_5) room(e_5) location(e_5) current(e_5)	`(<objs> ^object <e5>)`
	`(<e5> ^handle e5  ^item-type object  ^predicates <e5-preds>)`
	`(<e5-preds> ^category kitchen room location`
	`            ^modifier1 current1)`

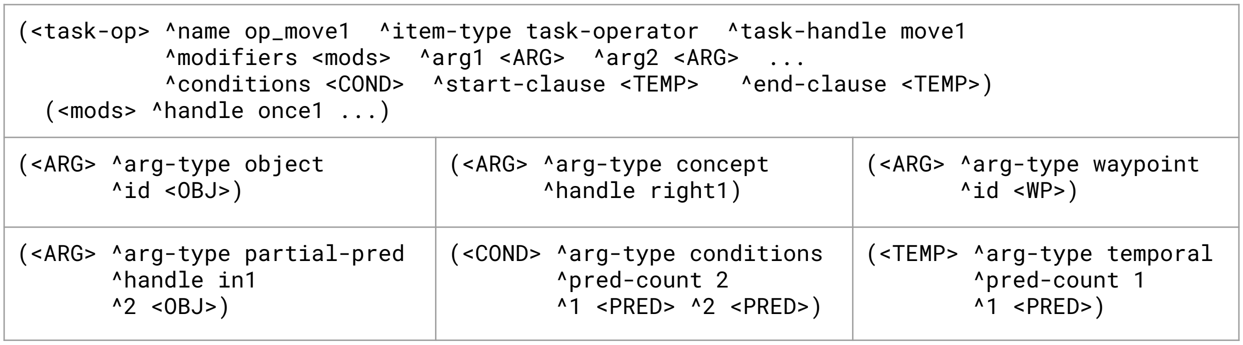
in(e_1, e_5) in(e_4, e_5)	`(<preds> ^predicate <in-pred>)`
	`   (<in-pred> ^handle in1  ^instance <i1> <i2>)`
	`      (<i1> ^1 <e1> ^2 <e5>)`
	`      (<i2> ^1 <e4> ^2 <e5>)`

stopped(robot) current-waypoint(wp01) current-location(e5)	`(<robot> ^handle rosie  ^moving-status stopped  ^arm <arm>`
	`   ^current-waypoint <wp01>  ^current-location <e5>)`
	`  (<arm> ^moving-status stopped  ^holding-object false)`
	`  (<wp01> ^handle wp01  ^x 3.00  ^y 3.00)`

Figure A.1: Examples of world predicates and the corresponding soar representations in working memory.

A.2 Tasks and Goals

Below is the specification for a task operator represented in working memory. The various arguments and clauses are all optional and appear in different combinations depending on the task. Objects are grounded to the world, and the predicates in the conditions or temporal clauses are the same as the goal predicates below.



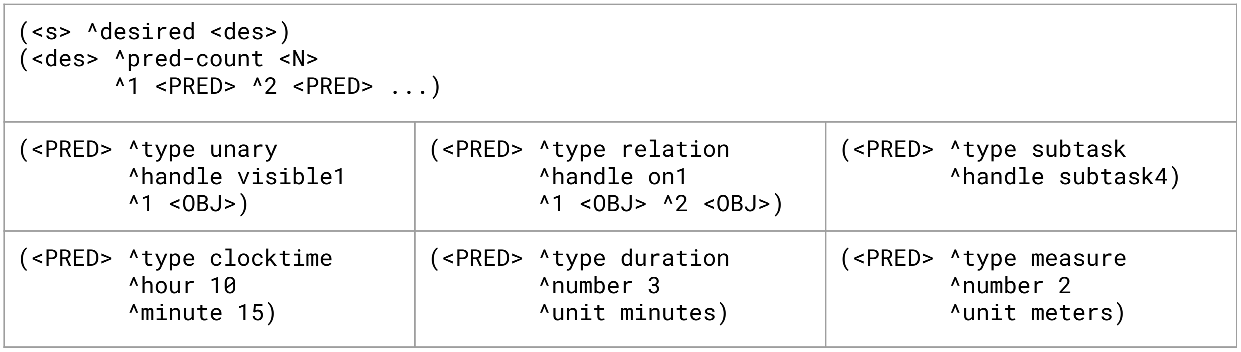
```
(<task-op> ^name op_move1  ^item-type task-operator  ^task-handle move1
           ^modifiers <mods>  ^arg1 <ARG>  ^arg2 <ARG>  ...
           ^conditions <COND>  ^start-clause <TEMP>   ^end-clause <TEMP>)
  (<mods> ^handle once1 ...)
```

`(<ARG> ^arg-type object`
`       ^id <OBJ>)`
	`(<ARG> ^arg-type concept`
	`       ^handle right1)`
		`(<ARG> ^arg-type waypoint`
		`       ^id <WP>)`

`(<ARG> ^arg-type partial-pred`
`       ^handle in1`
`       ^2 <OBJ>)`
	`(<COND> ^arg-type conditions`
	`        ^pred-count 2`
	`        ^1 <PRED> ^2 <PRED>)`
		`(<TEMP> ^arg-type temporal`
		`        ^pred-count 1`
		`        ^1 <PRED>)`

Figure A.2: The WM representation of a task operator, and all of the different types of arguments it can contain. Everything except the name, item-type, and task-handle is optional.

Given the current task and current goal handle, Rosie learns a goal elaboration rule that creates a predicate-based representation on the state.



```
(<s> ^desired <des>)
(<des> ^pred-count <N>
       ^1 <PRED> ^2 <PRED> ...)
```

`(<PRED> ^type unary`
`        ^handle visible1`
`        ^1 <OBJ>)`
	`(<PRED> ^type relation`
	`        ^handle on1`
	`        ^1 <OBJ> ^2 <OBJ>)`
		`(<PRED> ^type subtask`
		`        ^handle subtask4)`

`(<PRED> ^type clocktime`
`        ^hour 10`
`        ^minute 15)`
	`(<PRED> ^type duration`
	`        ^number 3`
	`        ^unit minutes)`
		`(<PRED> ^type measure`
		`        ^number 2`
		`        ^unit meters)`

Figure A.3: The WM goal representation as a set of predicates, and their different types.

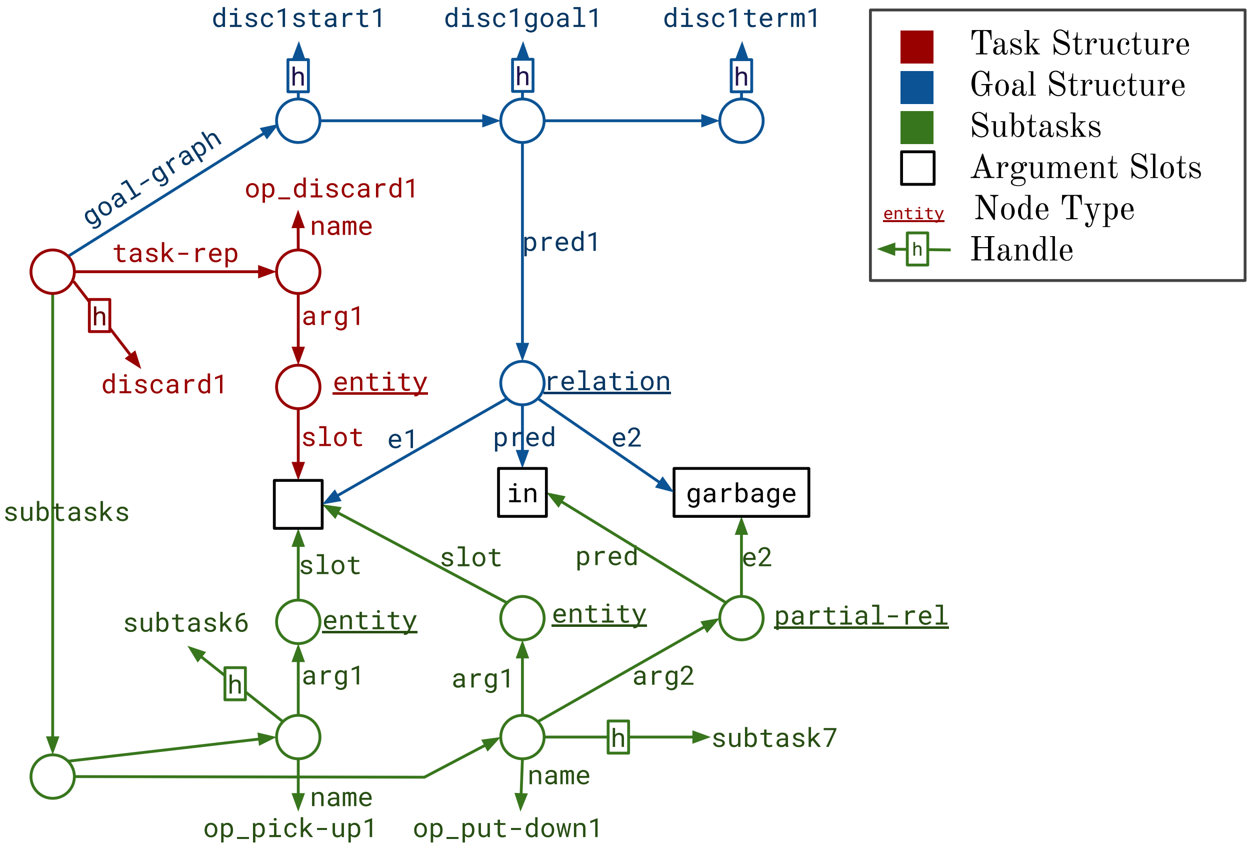


Figure A.4: The Task Concept Network learned for the task *"Discard the white cup."*

A.3 Task Concept Network

For the sake of clarity, the task concept networks shown throughout this dissertation hid certain details of the actual representations. Below, we show the TCN for discard and the actual representation in semantic memory.

```
(<TCN> ^item-type task  ^handle discard1
       ^procedural <task-rep>  ^goal-graph <start>)
   # Task Representation
   (<task-rep> ^op_name op_discard1 ^arg1 <arg1> ^subtasks <subtasks>)
     (<arg1> ^arg-type object ^id <slot2>)

   # Goal Graph
   (<start> ^item-type start-goal   ^handle discard1start1 ^next.goal <goal1>)
   (<goal1> ^item-type task-goal    ^handle discard1goal1  ^next.goal <term>
            ^pred-count 1  ^1 <pred1>)
     (<pred1> ^type relation ^id <pred-slot> ^1 <slot1> ^2 <slot2>)
```

```
(<term> ^item-type terminal-goal ^handle discard1term1)

# Subtasks
(<subtasks> ^subtask <sub6> <sub7>)
  (<sub6> ^handle subtask6  ^op_name op_pick-up1  ^task-handle pick-up1
          ^arg1 <sub6arg1>)
      (<sub6arg1> ^arg-type object ^id <slot1>)
  (<sub7> ^handle subtask7  ^op_name op_put-down1 ^task-handle put-down1
          ^arg1 <sub7arg1> ^arg2 <sub7arg2>)
      (<sub7arg1> ^arg-type object ^id <slot1>)
    (<sub7arg2> ^arg-type partial-pred  ^id <pred-slot>  ^2 <slot2>)

# Slots
(<pred-slot> ^default.predicate-handle in1)
(<slot2> ^default.category garbage1)
```

Every argument and predicate in the task and goal representations from the previous section has a dual within the TCN. Typically the only difference is that links to grounded objects and predicates are replaced with slots.

A.4 Learned Task Rules

In Section 6.4, we gave examples of the agent chunking different types of task knowledge into procedural rules. For clarity, we only showed summaries of the learned chunks. In this section, we show the actual soar rules that were learned. These are identical to the actual productions, except for rearranging and renaming the variables and formatting. Note that the representations use the word ‘object’, whereas in the dissertation we use the broader term ‘entity’.

A.4.1 Goal Elaboration

The following chunk elaborates the goal of discard, that the object is in the garbage.

```
sp {chunk*action*discard*elaborate*desired*discard1goal1
    (state <s> ^problem-space.name action
               ^name <op_discard>
               ^task-operator <cur-task>
               ^current-task-segment <cur-seg>
               ^world <world>)
   (<cur-task> ^task-handle discard1 ^name <op_discard> ^arg1 <arg1>)
      (<arg1> ^arg-type object ^id <obj>)
   (<cur-seg> ^current-goal.handle discard1goal1)

   (<world> ^objects.object <garbage>)
      (<garbage> ^predicates.category garbage1)
-->
   (<s> ^desired <des>)
      (<des> ^handle discard1goal1 ^pred-count 1 ^1 <pred1>)
         (<pred1> ^type relation ^handle in1 ^1 <obj> ^2 <garbage>)
}
```

A.4.2 Subtask Proposal

The following chunk proposes the pick-up subtask within the discard task state for any confirmed object that is grabbable and not-grabbed.

```
sp {chunk*action*discard*propose*subtask*pick-up1
   (state <s> ^problem-space.name action
              ^name <op_discard>
              ^task-operator <cur-task> ^world <world>)
   (<cur-task> ^name <op_discard> ^task-handle discard1 ^arg1 <arg1>)
```

```
      (<arg1> ^arg-type object ^id <obj>)

   (<world> ^robot <robot> ^objects <objects>)
   (<robot> ^arm.holding-object false)
   (<objects> ^object <obj>)
      (<obj> ^predicates <obj-preds>)
         (<obj-preds> ^affordance1 grabbable1 ^is-grabbable1 not-grabbed1
                      ^is-confirmed1 confirmed1)
-->
   (<s> ^operator <o> +)
   (<o> ^name op_pick-up1 ^item-type task-operator ^task-handle pick-up1
        ^subtask-handle subtask6 ^arg1 <st-arg1>)
      (<st-arg1> ^arg-type object ^id <obj>)
}
```

A.4.3 Subtask Policy

The following chunk is a policy rule for the discard task that gives a best preference to a put-down operator if it achieves the goal (desired) predicate.

```
sp {chunk*action*discard*prefer*subtask*put-down
   (state <s> ^problem-space.name action
              ^operator <o> +
              ^desired <des> ^world <world>)

    (<o> ^name op_put-down1 ^task-handle <task-h> ^subtask-handle <subtask-h>
         ^arg1 <arg1> ^arg2 <arg2>)
       (<arg1> ^arg-type object ^id <obj>)
       (<arg2> ^arg-type partial-predicate ^handle <in> ^2 <garbage>)
      -{ (<o> ^{ <name3> <> arg2 <> arg1 }.arg-type <arg3>) } # No third arg

    (<des> ^handle <des-h> -^modifiers <mods> ^pred-count 1 ^1 <goal-pred1>)
      (<goal-pred1> ^type relation ^handle <in> ^1 <obj> ^2 <garbage>)

    (<world> ^objects <objects> ^predicates <predicates>)
    (<objects> ^object <obj> ^object <garbage>)
    (<predicates> ^predicate <in-pred>)
       (<in-pred> ^handle <in>)
-->
```

```
   (<s> ^operator <o> >)
}
```

A.4.4 Action Model

The following chunk represents the action model for the *discard* task, which adds the relation that the object is in the garbage. It only applies if the execution-type is internal (the agent is simulating tasks) and it includes a check that the given relation does not already exist in the world.

```
sp {chunk*action*execution-type*internal*apply*discard
   (state <s> ^problem-space <ps> ^operator <o> ^world <world>)
      (<ps> ^name action ^execution-type internal)

   (<o> ^item-type task-operator ^task-handle discard1 ^arg1 <arg1>)
      (<arg1> ^arg-type object ^id <obj1>)

   (<world> ^objects <objects> ^predicates <preds>)
   (<objects> ^object <obj1> ^object <obj2>)
      (<obj2> ^predicates <obj2-preds>)
         (<obj2-preds> ^category garbage1)

   (<preds> ^predicate <in-pred>)
      (<in-pred> ^handle in1)
     -{ (<in-pred> ^instance <i1>)
       (<i1> ^1 <obj1> ^2 <obj2>) }
-->
   (<in-pred> ^instance <i>)
   (<i> ^1 <obj1> ^2 <obj2>)
}
```

APPENDIX B

Innate Task Implementations

In this chapter, we provide more details for each innate task included in Rosie (Figure 7.2).

B.1 Manipulation Actions

All the manipulation actions map to a motor command that the agent can perform. However, this motor command has additional constraints that were not in the task preconditions. Mostly, these are that the objects involved are `visible` and `reachable`. Table B.1 shows the preconditions for each manipulation task, as well as its goal.

Task	**Preconditions**	**Goal**
`pick-up(`e_i`)`	`not-grabbed(`e_i`)` $\wedge$ `not-holding-object(`arm`)`	`grabbed(`e_i`)`
`put-down(`e_i`)`	`grabbed(`e_i`)`	`not-grabbed(`e_i`)`
`put-down(`e_i`, on(`e_j`))`	`grabbed(`e_i`)` $\wedge$ `surface(`e_j`)`	`on(`e_i`, `e_j`)`
`put-down(`e_i`, in(`e_j`))`	`grabbed(`e_i`)` $\wedge$ `receptacle(`e_j`)` $\wedge$ `!openable(`e_j`)`	`in(`e_i`, `e_j`)`
`put-down(`e_i`, in(`e_j`))`	`grabbed(`e_i`)` $\wedge$ `receptacle(`e_j`)` $\wedge$ `openable(`e_j`)` $\wedge$ `is-open(`e_j`)`	`in(`e_i`, `e_j`)`
`pour(`e_i`, in(`e_j`))`	`dispenser(`e_i`)` $\wedge$ `grabbed(`e_i`)` $\wedge$ `fillable(`e_j`)` $\wedge$ `empty(`e_j`)`	`exec(pour-cmd)`
`pour(`e_i`, in(`e_j`))`	`contains(`e_i`, `e_{liq}`)` $\wedge$ `drain(`e_j`)`	`exec(pour-cmd)`
`give(`e_i`, to(`e_j`))`	`grabbed(`e_i`)` $\wedge$ `person(`e_j`)`	`holding(`e_j`, `e_i`)`
`open(`e_i`)`	`is-closed(`e_i`)`	`is-open(`e_i`)`
`close(`e_i`)`	`is-open(`e_i`)`	`is-closed(`e_i`)`
`turn-on(`e_i`)`	`is-off(`e_i`)`	`is-on(`e_i`)`
`turn-off(`e_i`)`	`is-on(`e_i`)`	`is-off(`e_i`)`

Table B.1: The task structure and goal for each manipulation task, along with some of the preconditions. For a task to be proposed, all objects or people involved must also be `confirmed`.

Most action models simply change the state to include the goal (e.g., the model for *close* adds `not-open(`e_i`)` and removes `open(`e_i`)`. The only non-obvious actions models are that *pick-up* removes any spatial relations in the world involving that object, *put-down* adds that the object is in the current location, and *open* will mark any objects inside as visible.

B.2 Movement Tasks

The movement-based innate tasks typically do not have proposal rules or action models and have the goal of executing some motor command. The exceptions are *approach*, which has the goal of the entity becoming `reachable`, and *go-to-location*, which has the goal of being in that location. Often movement tasks use other movement tasks within them. For example, *approach* uses *face* and *go-to-xy*.

Task	**Description**
`approach(`e_i`)`	Drive up to the entity so that it is `reachable`.
`drive(`*distance*`)`	Drive forward some number of meters.
`drive(through(`e_{door}`))`	Drive through the given doorway.
`face(`e_i`)`	Turn towards the given entity.
`go-to-location(`e_{loc}`)`	Drive to a location entity in the world, such as the kitchen or main office.
`go-to-waypoint(`*wp*`)`	Navigate to the given waypoint using an A* search on the waypoint graph.
`go-to-next-waypoint(`*wp*`)`	Use lower-level navigation primitives to traverse one edge of the waypoint graph to drive to an adjacent waypoint.
`go-to-xy(`*coord*`)`	Face and drive to the given xy coordinate.
`orient(`*dir*`)`	Face the given cardinal direction (e.g. north).
`stop()`	Stop or interrupt the current task.
`turn(`*dir*`)`	Turn the given relative direction (e.g. left).
`turn(`*dir*`, `*deg*`)`	Turn the specified number of degrees.

Table B.2: A list of the innate movement tasks in Rosie.

B.3 Perceptual Tasks

In Rosie we have implemented four innate perceptual tasks, two of which have proposal rules and action models (*view* and *find*). *Scan* is what we use to have the robot look around the room by turning 360 degrees. *View* has the goal of the entity being `visible` when it has a known spatial location. Usually it turns to face the entity, but if an object is inside a closed receptacle, it will go to open it. The latter behavior is not pre-programmed, it is the result of combining planning with the action model for *open*. The *Explore* task will visit and scan every waypoint in the waypoint map. When it has scanned the current waypoint, it selects the closest unvisited waypoint next. Explore is typically paired with an until clause so it terminates early when some condition is met, such as *'explore until a stapler is visible.'* *Find* has the goal of an entity being `confirmed` when it does not know where it is. It is the most complex innate task, as we have programmed a lot of different strategies for it. The subtasks proposed for find are in the following Table B.3.

P1: STM	If `in`(e_i, e_{loc}) then `go-to-location`(e_{loc}) If `on`(e_i, e_{surf}) and `unconfirmed`(e_{surf}) then `find`(e_{surf}) If `in`(e_i, e_{rec}) and `unconfirmed`(e_{rec}) then `find`(e_{rec}) If `in`(e_i, e_{rec}) and `closed`(e_{rec}) then `open`(e_{rec})
P2: Local	`scan(until{visible(`e_i`)})`
P3: LTM	If `object`(e_i) then `recall`(e_{loc}, e_i) If `person`(e_i) then `recall`(e_{off}, e_i) `recall`($e_{cur-loc}$, `when{visible(`e_i`)})`
P4: Interact	`initiate-interaction(get-find-help(`e_i`))`
P5: Global	`explore(until{visible(`e_i`)})`

Table B.3: A list of the pre-programmed strategies (subtask proposals) for finding an object in preference order. Entity e_i is the object being found.

B.4 Communicative Tasks

Of the three communicative tasks, *say* is the only one with a pre-programmed proposal rule:

```
IF:
   message($e_i$) ∧ sentence($e_i$, $s$) ∧ person($e_j$) ∧
   visible($e_j$) ∧ !heard($e_j$, $e_i$)
THEN:
   propose say($e_i$, to($e_j$))
```

Say can also be used with only a sentence and no target person, but only when given by an instructor in a procedural task. Since say involves a message entity with a complete sentence, the whole sentence is simply sent to the instructor. *Describe* is very similar, except that it happens with a non-message entity such as an object. The code that generates natural language sentences from structured agent messages using templates can take the predicates associated with an object and turn them into an English description. *Ask* will also send an outgoing message with the question sentence, but will then wait for a response. After it receives a response, the agent will add it as an entity in the world along with the predicate `answered`(e_i). The following Table B.4 shows examples of the three types of responses that Rosie understands and the entities added to the world.

Response	Added Entity
yes	`message(`e_i`) yes(`e_i`) answered(`e_i`)`

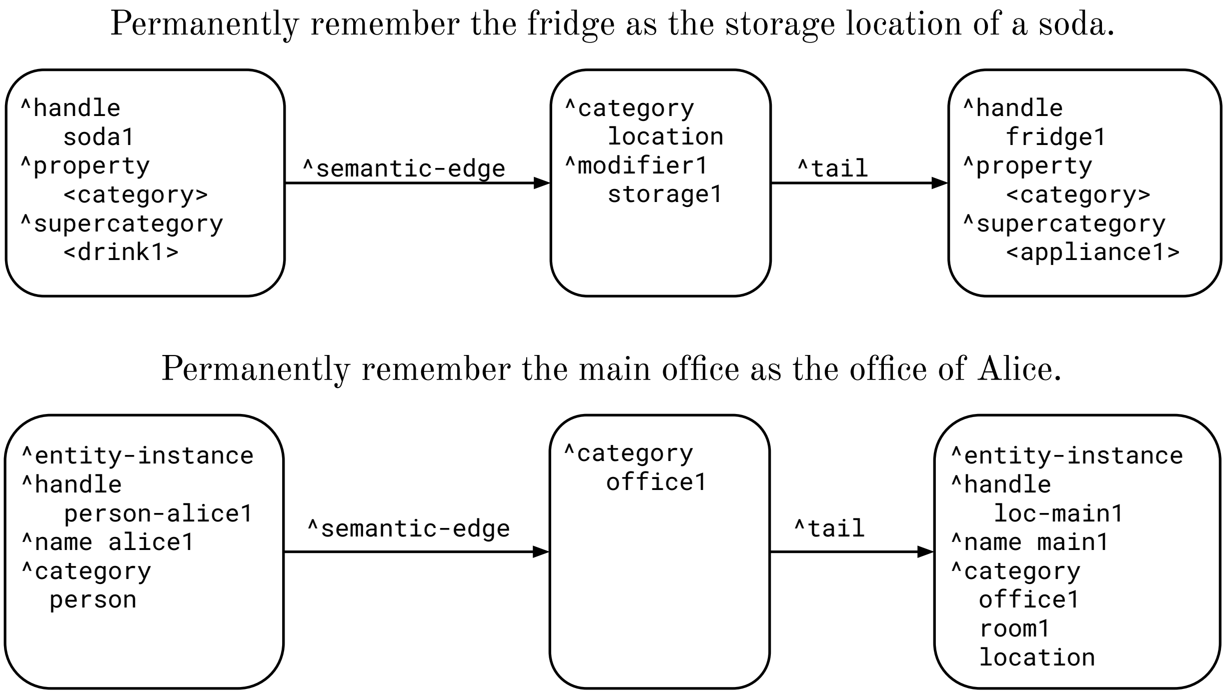
'I will be late.'	`message(`e_i`) answered(`e_i`)`
	`sentence(`e_i`, |I will be late|)`

a soda	`soda(`e_i`) drink(`e_i`) object(`e_i`)`
	`not-confirmed(`e_i`) answered(`e_i`)`

Table B.4: Three possible types of responses to the agent asking a question, and the corresponding entities added to the world as a result.

B.5 Mental Tasks

The two main mental tasks were covered in detail in Section 7.3.5. The third is *wait*, which should be paired with an end clause or duration such as *'for three minutes'*. Here, we will discuss how structures are stored and retrieved in semantic memory. If the instructor says *'Permanently remember the fridge as the storage location of a soda.'*, this defines an edge in semantic memory labelled 'storage location' that goes from the soda to the fridge. The head and tail nodes are expected to already exist in semantic memory. If there is a proper entity instance stored in smem that matches the description, it is used. So if the person said *'Permanently remember the main office as the office of Alice'*, both Alice and the main office exist as long-term instances in semantic memory, so those are used. If the entity description does not match a proper entity in semantic memory (as with soda or fridge), the semantic category is used instead. An edge is then created between them with the edge predicates. An example is shown in Figure B.1. These can be used during *recall* commands (e.g., *'Recall the storage location of a soda.'*, where a similar structure matching process occurs.



Permanently remember the fridge as the storage location of a soda.

Permanently remember the main office as the office of Alice.

Figure B.1: Two examples of semantic edges created through *permanently remember* commands. The first connects two categories; the second connects long-term entity instances.

BIBLIOGRAPHY

Ahmadzadeh, S. R., Kaushik, R., & Chernova, S. (2016). Trajectory learning from demonstration with canal surfaces: A parameter-free approach. *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (pp. 544–549). IEEE.

Akgun, B., Cakmak, M., Jiang, K., & Thomaz, A. L. (2012). Keyframe-based learning from demonstration. *International Journal of Social Robotics*, *4*, 343–355.

Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., & Taysom, W. (2007). Plow: A collaborative task learning agent. *AAAI* (pp. 1514–1519).

Amiri, S., Bajracharya, S., Goktolga, C., Thomason, J., & Zhang, S. (2019). Augmenting knowledge through statistical, goal-oriented human-robot dialog. *arXiv preprint arXiv:1907.03390*.

Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, *57*, 469–483.

Artzi, Y., & Zettlemoyer, L. (2013). Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, *1*, 49–62.

Arumugam, D., Karamcheti, S., Gopalan, N., Wong, L. L., & Tellex, S. (2017). Accurately and efficiently interpreting human-robot instructions of varying granularities. *arXiv preprint arXiv:1704.06616*.

Arzate Cruz, C., & Igarashi, T. (2020). A survey on interactive reinforcement learning: Design principles and open challenges. *Proceedings of the 2020 ACM Designing Interactive Systems Conference* (pp. 1195–1209).

Baker, C. F., Fillmore, C. J., & Lowe, J. B. (1998). The berkeley framenet project. *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1* (pp. 86–90).

Bergen, B., & Chang, N. (2013). Embodied construction grammar. In *The Oxford handbook of construction grammar*. Oxford University Press.

Billard, A., Calinon, S., Dillmann, R., & Schaal, S. (2008). Survey: Robot programming by demonstration. *Handbook of robotics*, *59*.

Blodow, N., Jain, D., Marton, Z.-C., & Beetz, M. (2010). Perception and probabilistic anchoring for dynamic world state logging. *Proceedings of the Tenth IEEE-RAS International Conference on Humanoid Robots* (pp. 160–166). Nashville, TN: IEEE.

Bouguerra, A., Karlsson, L., & Saffiotti, A. (2006). Situation assessment for sensor-based recovery planning. *Frontiers in Artificial Intelligence and Applications*, *141*, 673–677.

Branavan, S., Zettlemoyer, L. S., & Barzilay, R. (2010). Reading between the lines: Learning to map high-level instructions to commands. *48th Annual Meeting of the Association for Computational Linguistics*.

Brandimonte, M. A., Einstein, G. O., & McDaniel, M. A. (2014). *Prospective memory: Theory and applications*. Psychology Press.

Broxvall, M., Coradeschi, S., Karlsson, L., & Saffiotti, A. (2005). Recovery planning for ambiguous cases in perceptual anchoring. *Proceedings of the Twentieth National Conference on Artificial Intelligence* (pp. 1254–1260). Menlo Park, CA: AAAI Press.

Budanitsky, A., & Hirst, G. (2006). Evaluating wordnet-based measures of lexical semantic relatedness. *Computational linguistics*, *32*, 13–47.

Calinon, S., Guenter, F., & Billard, A. (2007). On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *37*, 286–298.

Cantrell, R., Talamadupula, K., Schermerhorn, P., Benton, J., Kambhampati, S., & Scheutz, M. (2012). Tell me when and why to do it! run-time planner model updates via natural language instruction. *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction* (pp. 471–478).

Chao, C., Cakmak, M., & Thomaz, A. L. (2011). Towards grounding concepts for transfer in goal learning from demonstration. *2011 IEEE International Conference on Development and Learning (ICDL)* (pp. 1–6). IEEE.

Chaurasia, S., & Mooney, R. (2017). Dialog for language to code. *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)* (pp. 175–180).

Chen, D. L., & Mooney, R. J. (2011). Learning to interpret natural language navigation instructions from observations. *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Chu, V., Fitzgerald, T., & Thomaz, A. L. (2016). Learning object affordances by leveraging the combination of human-guidance and self-exploration. *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)* (pp. 221–228). IEEE.

Coradeschi, S., & Saffiotti, A. (2003). An introduction to the anchoring problem. *Robotics and Autonomous Systems*, *43*, 85–96.

Crangle, C., & Suppes, P. (1994). *Language and learning for robots*. Stanford, CA: Center for the Study of Language and Information.

Cypher, A., & Halbert, D. C. (1993). *Watch what I do: programming by demonstration*. MIT press.

Davies, M. (2009). The 385+ million word corpus of contemporary american english (1990–2008+): Design, architecture, and linguistic insights. *International journal of corpus linguistics*, *14*, 159–190.

DeJong, G., & Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine learning*, *1*, 145–176.

Dillmann, R. (2004). Teaching and learning of robot tasks via observation of human performance. *Robotics and Autonomous Systems*, *47*, 109–116.

Elfring, J., Van Den Dries, S., Van De Molengraft, M., & Steinbuch, M. (2013). Semantic world modeling using probabilistic multiple hypothesis anchoring. *Robotics and Autonomous Systems*, *61*, 95–105.

Elliott, S., Xu, Z., & Cakmak, M. (2017). Learning generalizable surface cleaning actions from demonstration. *2017 26th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)* (pp. 993–999). IEEE.

Erol, K., Hendler, J., & Nau, D. S. (1994). Htn planning: Complexity and expressivity. *AAAI* (pp. 1123–1128).

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*, 189–208.

Forbes, M., Rao, R. P., Zettlemoyer, L., & Cakmak, M. (2015). Robot programming by demonstration with situated spatial language understanding. *2015 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 2014–2020). IEEE.

Frasca, T., Oosterveld, B., Krause, E., & Scheutz, M. (2018). One-shot interaction learning from natural language instruction and demonstration. *Advances in Cognitive Systems*, *6*, 1–18.

Gemignani, G., Bastianelli, E., & Nardi, D. (2015). Teaching robots parametrized executable plans through spoken interaction. *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems* (pp. 851–859).

Gluck, K. A., & Laird, J. E. (2019). *Interactive Task Learning: Humans, Robots, and Agents Acquiring New Tasks Through Natural Interactions*. MIT Press.

Gong, Z., & Zhang, Y. (2018). Temporal spatial inverse semantics for robots communicating with humans. *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 4451–4458). IEEE.

Grosz, B. J., & Sidner, C. L. (1986). Attention, intentions, and the structure of discourse. *Computational linguistics*, *12*, 175–204.

Gundel, J. K., Hedberg, N., & Zacharski, R. (1993). Cognitive status and the form of referring expressions in discourse. *Language*, (pp. 274–307).

Havoutis, I., & Calinon, S. (2019). Learning from demonstration for semi-autonomous teleoperation. *Autonomous Robots*, *43*, 713–726.

Hayes, B., & Scassellati, B. (2014). Discovering task constraints through observation and active learning. *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 4442–4449). IEEE.

Heyer, T., & Graser, A. (2012). Intelligent object anchoring using relative anchors. *Proceedings of the Thirteenth International Conference on Optimization of Electrical and Electronic Equipment* (pp. 1444–1451). Brasov, Romania: IEEE.

von Hoyningen-Huene, N., & Beetz, M. (2009). Robust real-time multiple target tracking. *Proceedings of the Ninth Asian Conference on Computer Vision* (pp. 247–256). Xi'an, China.

Huffman, S. B., & Laird, J. E. (1995). Flexibly instructable agents. *Journal of Artificial Intelligence Research*, *3*, 271–324.

Khan, Z., Balch, T., & Dellaert, F. (2006). Mcmc data association and sparse factorization updating for real time multitarget tracking with merged and multiple measurements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *28*, 1960–1972.

Kipper, K., Korhonen, A., Ryant, N., & Palmer, M. (2008). A large-scale classification of english verbs. *Language Resources and Evaluation*, *42*, 21–40.

Kirk, J. R., & Laird, J. E. (2016). Learning general and efficient representations of novel games through interactive instruction. *Proceedings of the Fourth Annual Conference on Advances in Cognitive Systems*. Evanston, IL.

Kirk, J. R., & Laird, J. E. (2019). Learning hierarchical symbolic representations to support interactive task learning and knowledge transfer. *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (pp. 6095–6102). AAAI Press.

Kukliński, K., Fischer, K., Marhenke, I., Kirstein, F., Maria, V., Sølvason, D., Krüger, N., & Savarimuthu, T. R. (2014). Teleoperation for learning by demonstration: Data glove versus object manipulation for intuitive robot control. *2014 6th international congress on Ultra modern telecommunications and control systems and workshops (ICUMT)* (pp. 346–351). IEEE.

Labutov, I., Srivastava, S., & Mitchell, T. (2018). Lia: A natural language programmable personal assistant. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (pp. 145–150).

Laird, J. (2012). *The Soar cognitive architecture*. Cambridge, MA: MIT Press.

Lauria, S., Bugmann, G., Kyriacou, T., & Klein, E. (2002). Mobile robot programming using natural language. *Robotics and Autonomous Systems*, *38*, 171–181.

Li, J., & Laird, J. (2014). Spontaneous retrieval for prospective memory: Effects of encoding specificity and retention interval. *Proceedings of the 29th AAAI Conference on Artificial Intelligence. AAAI*.

Li, T. J.-J., Mitchell, T., & Myers, B. (2020). Interactive task learning from gui-grounded natural language instructions and demonstrations. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations* (pp. 215–223).

Lindes, P., Mininger, A., Kirk, J. R., & Laird, J. E. (2017). Grounding language for interactive task learning. *Proceedings of the First Workshop on Language Grounding for Robotics at ACL*.

Liu, Y., Gupta, A., Abbeel, P., & Levine, S. (2018). Imitation from observation: Learning to imitate behaviors from raw video via context translation. *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1118–1125). IEEE.

Loutfi, A., Coradeschi, S., & Saffiotti, A. (2005). Maintaining coherent perceptual information using anchoring. *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (pp. 1477–1482). Edinburgh, Scotland.

Matuszek, C., FitzGerald, N., Zettlemoyer, L., Bo, L., & Fox, D. (2012). A joint model of language and perception for grounded attribute learning. *arXiv preprint arXiv:1206.6423*.

Matuszek, C., Herbst, E., Zettlemoyer, L., & Fox, D. (2013). Learning to parse natural language commands to a robot control system. *Experimental robotics* (pp. 403–415). Springer.

Meriçli, C., Klee, S. D., Paparian, J., & Veloso, M. (2013). An interactive approach for situated task teaching through verbal instructions. *Intelligent Robotic Systems* (pp. 47–52).

Mininger, A., & Laird, J. E. (2016). Interactively learning strategies for handling references to unseen or unknown objects. *Proceedings of the Fourth Annual Conference on Advances in Cognitive Systems*. Evanston, IL.

Mininger, A., & Laird, J. E. (2018). Interactively learning a blend of goal-based and procedural tasks. *Thirty-Second AAAI Conference on Artificial Intelligence* (pp. 1497–1494).

Mininger, A., & Laird, J. E. (2019). Using domain knowledge to correct anchoring errors in a cognitive architecture. *Advances in Cognitive Systems*, *8*, 133–148.

Misra, D., Langford, J., & Artzi, Y. (2017). Mapping instructions and visual observations to actions with reinforcement learning. *arXiv preprint arXiv:1704.08795*.

Misra, D. K., Sung, J., Lee, K., & Saxena, A. (2016). Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research*, *35*, 281–300.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine learning*, *1*, 47–80.

Mohan, S., & Laird, J. E. (2014). Learning goal-oriented hierarchical tasks from situated interactive instruction. *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (pp. 387–394).

Mohan, S., Mininger, A., & Laird, J. E. (2013). Towards an indexical model of situated language comprehension for cognitive agents in physical worlds. *Advances in Cognitive Systems*, *3*, 163–182.

Mohan, S., Mininger, A. H., Kirk, J. R., & Laird, J. E. (2012). Acquiring grounded representations of words with situated interactive instruction. *Advances in Cognitive Systems*, *2*, 113–130.

Mohseni-Kabir, A., Li, C., Wu, V., Miller, D., Hylak, B., Chernova, S., Berenson, D., Sidner, C., & Rich, C. (2019). Simultaneous learning of hierarchy and primitives for complex robot tasks. *Autonomous Robots*, *43*, 859–874.

Mohseni-Kabir, A., Rich, C., Chernova, S., Sidner, C. L., & Miller, D. (2015). Interactive hierarchical task learning from a single demonstration. *Proceedings of the Tenth Annual ACM/IEEE International Conference on Human-Robot Interaction* (pp. 205–212).

Mollard, Y., Munzer, T., Baisero, A., Toussaint, M., & Lopes, M. (2015). Robot programming from demonstration, feedback and transfer. *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)* (pp. 1825–1831). IEEE.

Nehaniv, C. L., & Dautenhahn, K. (2002). The correspondence problem. *Imitation in animals and artifacts*, *41*.

Newell, A. (1993). Reasoning, problem solving, and decision processes: The problem space as a fundamental category. In *The Soar papers (vol. 1) research on integrated intelligence*, (pp. 55–80). MIT Press.

Nicolescu, M. N., & Mataric, M. J. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. *Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (pp. 241–248).

Nitti, D., De Laet, T., & De Raedt, L. (2014). Relational object tracking and learning. *Proceedings of the 2014 IEEE International Conference on Robotics and Automation* (pp. 935–942). Hong Kong, China: IEEE.

Palmer, M., Gildea, D., & Kingsbury, P. (2005). The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, *31*, 71–106.

Palmer, M., Gildea, D., & Xue, N. (2010). Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, *3*, 1–103.

Perera, I., Allen, J., Teng, C. M., & Galescu, L. (2018). A situated dialogue system for learning structural concepts in blocks world. *Proceedings of the 19th Annual SIGdial Meeting on Discourse and Dialogue* (pp. 89–98).

Persson, A., Dos Martires, P. Z., De Raedt, L., & Loutfi, A. (2019). Semantic relational object tracking. *IEEE Transactions on Cognitive and Developmental Systems*, *12*, 84–97.

Phillips, M., Hwang, V., Chitta, S., & Likhachev, M. (2016). Learning to plan for constrained manipulation from demonstrations. *Autonomous Robots*, *40*, 109–124.

Ramaraj, P., Sahay, S., Kumar, S. H., Lasecki, W. S., & Laird, J. E. (2019). Towards using transparency mechanisms to build better mental models. *Seventh Annual Advances in Cognitive Systems Goal Reasoning Workshop*. Cambridge MA.

Ravichandar, H., Polydoros, A. S., Chernova, S., & Billard, A. (2020). Recent advances in robot learning from demonstration. *Annual Review of Control, Robotics, and Autonomous Systems*, *3*.

Rich, C., & Sidner, C. L. (1998). Collagen: A collaboration manager for software interface agents. In *Computational Models of Mixed-Initiative Interaction*, (pp. 149–184). Springer.

Rosen, E., Whitney, D., Phillips, E., Ullman, D., & Tellex, S. (2018). Testing robot teleoperation using a virtual reality interface with ros reality. *Proceedings of the 1st International Workshop on Virtual, Augmented, and Mixed Reality for HRI (VAM-HRI)* (pp. 1–4).

Rosenthal, S., Biswas, J., & Veloso, M. M. (2010). An effective personal mobile robot agent through symbiotic human-robot interaction. *AAMAS* (pp. 915–922).

Rychener, M. D. (1983). The instructible production system: A retrospective analysis. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning, An Artificial Intelligence Approach*, (pp. 429–459). Springer.

Scheutz, M., Williams, T., Krause, E., Oosterveld, B., Sarathy, V., & Frasca, T. (2019). An overview of the distributed integrated cognition affect and reflection diarc architecture. In *Cognitive architectures*, (pp. 165–193). Springer.

She, L., & Chai, J. (2017). Interactive learning of grounded verb semantics towards human-robot communication. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1634–1644).

She, L., Yang, S., Cheng, Y., Jia, Y., Chai, J., & Xi, N. (2014). Back to the blocks world: Learning new actions through situated human-robot dialogue. *Proceedings of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL)* (pp. 89–97).

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2* (pp. 981–988).

Simon, H. A., & Hayes, J. R. (1976). The understanding process: Problem isomorphs. *Cognitive psychology*, *8*, 165–190.

Spranger, J., Buzatoiu, R., Polydoros, A., Nalpantidis, L., & Boukas, E. (2018). Human-machine interface for remote training of robot tasks. *arXiv preprint arXiv:1809.09558*.

Spranger, M., & Steels, L. (2015). Co-acquisition of syntax and semantics-an investigation in spatial language. *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Suddrey, G., Lehnert, C., Eich, M., Maire, F., & Roberts, J. (2016). Teaching robots generalizable hierarchical tasks through natural language instruction. *IEEE Robotics and Automation Letters*, *2*, 201–208.

Sutton, R. S. (1985). *Temporal Credit Assignment in Reinforcement Learning.*. Ph.D. thesis, University of Massachusetts Amherst.

Tellex, S. A., Kollar, T. F., Dickerson, S. R., Walter, M. R., Banerjee, A., Teller, S., & Roy, N. (2011). Understanding natural language commands for robotic navigation and mobile manipulation. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Thomason, J., Padmakumar, A., Sinapov, J., Walker, N., Jiang, Y., Yedidsion, H., Hart, J., Stone, P., & Mooney, R. (2020). Jointly improving parsing and perception for natural language commands through human-robot dialog. *Journal of Artificial Intelligence Research*, *67*, 327–374.

Thomason, J., Sinapov, J., Svetlik, M., Stone, P., & Mooney, R. J. (2016). Learning multi-modal grounded linguistic semantics by playing" i spy". *IJCAI* (pp. 3477–3483).

Thomaz, A. L., & Breazeal, C. (2008). Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence*, *172*, 716–737.

Vo, B.-N., Mallick, M., bar shalom, Y., Coraluppi, S., Osborne III, R., Mahler, R., & Vo, B.-T. (2015). Multitarget tracking. *Wiley encyclopedia of electrical and electronics engineering*, (pp. 1–25).

Vogt, D., Stepputtis, S., Grehl, S., Jung, B., & Amor, H. B. (2017). A system for learning continuous human-robot interactions from human-human demonstrations. *2017 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 2882–2889). IEEE.

Wang, W., Li, R., Chen, Y., Diekel, Z. M., & Jia, Y. (2018). Facilitating human–robot collaborative tasks by teaching-learning-collaboration from human demonstrations. *IEEE Transactions on Automation Science and Engineering*, *16*, 640–653.

Winograd, T. (1972). Understanding natural language. *Cognitive psychology*, *3*, 1–191.

Wray III, R. E., Taatgen, N. A., Lebiere, C., Pastra, K., Pirolli, P., Rosenbloom, P. S., Scheutz, M., Stewart, T. C., & Wiles, J. (2019). *Functional Knowledge Requirements for Interactive Task Learning*, chapter 3, (pp. 19–51). MIT Press.

Zhang, T., McCarthy, Z., Jow, O., Lee, D., Chen, X., Goldberg, K., & Abbeel, P. (2018). Deep imitation learning for complex manipulation tasks from virtual reality teleoperation. *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1–8). IEEE.

Ziparo, V. A., Iocchi, L., Lima, P. U., Nardi, D., & Palamara, P. F. (2011). Petri net plans. *Autonomous Agents and Multi-Agent Systems*, *23*, 344–383.