

Interactive Language-Based Task Library Instruction and Management for Single and Multiple Robots

Steven D. Klee

CMU-CS-15-114
June 2015

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Manuela Veloso, Chair
M. Bernardine Dias

*Submitted in partial fulfillment of the requirements
for the degree of Master of Science.*

Keywords: Robot Task Specification, Human Robot Interaction

Abstract

Robots acquire behaviors to perform tasks, in general by being programmed, or occasionally by being instructed through demonstrations. In this thesis, we address the challenge of providing task behaviors to a robot through language instructions and interactions.

We consider robots equipped with built-in motion and perception primitives with their functionality known to the user. We contribute *Instruction Graphs*, a robot task representation for composing sequences, conditionals, and loops of robot primitives. Instruction Graphs are only robot-primitive dependent, therefore applying to any robot hardware platform. We present the process by which Instruction Graphs are incrementally created from language instructions, as well as the method to execute Instruction Graphs on robots. We present examples and demonstrations of acquired Instruction Graphs with our CoBot mobile service robot and our Baxter manipulator robot.

We extend Instruction Graphs from single-robot tasks to multi-robot tasks with multi-robot sparse interaction coordination. We introduce Instruction Graph constructs that enable a robot to query the state of another robot, similar to a sensing primitive. We show examples where multi-robot Instruction Graphs are used to coordinate the two arms of Baxter, and to coordinate CoBot and Baxter.

Motivated by the fact that the robots can accumulate a large number of Instruction Graphs, we finally address the problem of managing a task library. We first devise an approach for correcting single steps of an Instruction Graph from a task library during its test execution. We then contribute an algorithm for learning *Generalized Instruction Graphs*, which represent parameterized tasks, from a task library of instantiated Instruction Graphs. We further contribute an algorithm that, given the initial steps of a new task, proposes an autocompletion based on a recognized similar Generalized Instruction Graph. We show results with large task libraries, in particular for the Baxter robot, in which the instruction of new tasks benefits from this generalization and autocompletion approach.

We discuss future work in terms of achieving the complete deployment of our approach on multiple robot platforms as available. We also discuss conditional representations of robot primitives, alternative filters for generalization and autocompletion, and the potential to interleave human instruction, multi-step correction, and the automated planning of Instruction Graphs.

Acknowledgements

First and foremost, I want to thank my advisor Manuela Veloso. Starting from my undergraduate studies, Manuela has provided so much support and guidance. She has always pushed me to do my best work. This thesis would not have been possible without her vision and contagious enthusiasm. I would also like to thank my committee member Bernardine Dias for her insightful comments and suggestions.

Secondly, I want to thank all of the members of the CORAL lab. In particular, I am grateful for all of the help that Çetin Meriçli and Juan Pablo Mendoza provided when I first started in the lab. Without their assistance, I would never have gotten past turning on the robots!

I am also especially thankful to Guglielmo Gemignani. Our collaboration during his visit to CMU was instrumental in finishing Chapters 3 and 4 of the thesis.

Lastly, I would like to thank my family for their love, support, and encouragement.

Contents

1	Introduction	1
1.1	CoBot and Baxter	2
1.2	Outline	2
2	Instruction Graphs	5
2.1	Formalization of Instruction Graphs	6
2.1.1	Execution of Instruction Graphs	7
2.1.2	Instruction Graphs on CoBot and Baxter	8
2.2	Generating Instruction Graphs from User Input	12
2.2.1	Generating Instruction Graphs	12
2.2.2	Generating Instruction Graphs on CoBot and Baxter	14
2.3	User Demonstrations	17
2.3.1	Participants	17
2.3.2	Procedures	17
2.3.3	Analysis	18
2.3.4	Discussion and Motivation for Future Work	19
2.4	Related Work	20
2.5	Summary	22
3	Multi-Robot Sparse-Coordination	23
3.1	Instruction Graphs for Sparse Coordination	23
3.2	Demonstrative Examples	25
3.2.1	Store Task	26
3.2.2	Deliver and Store Task	29
3.3	Related Work	31
3.4	Summary	32
4	Task Management	33
4.1	Task Correction	33
4.1.1	Example Correction on CoBot	34
4.2	Task Generalization and Autocompletion	35
4.2.1	Task Generalization	35
4.2.2	Task Autocompletion	38
4.2.3	Experiments	41

4.3	Related Work	44
4.4	Summary	45
5	Conclusion	47
5.1	Contributions	47
5.2	Directions For Future Work	48
A	Robot Primitive Language Descriptions	49
A.1	CoBot Language Descriptions and Examples	49
A.2	Baxter Language Descriptions and Examples	50
A.2.1	Landmarks	50
B	Planning with Instruction Graphs	51
B.1	Inferring Instruction Graph Preconditions and Effects	51
B.2	Planning For Task Proposals in Blocks World	52
B.3	Summary	54
	Bibliography	55

Chapter 1

Introduction

As robots become more pervasive in day-to-day life, we want them to interact with people, respond to their instructions, and acquire *new* tasks. However, it does not seem feasible for the average user to know how to program a robot. Instead, language appears to be the most natural medium for humans to command a robot, since we use it to teach one another. There are many challenges associated with teaching tasks to a robot through language.

One challenge is to define a robot-hardware independent approach to task teaching through verbal instructions. In particular, if we consider robots equipped with motion and sensing primitives known the user, then the approach should only depend on these robot-primitives. To this end, we introduce robot-primitive based task representation and a framework for a robot to acquire tasks incrementally from user interactions.

A second challenge is that users may provide tasks that require the coordination of multiple, possibly different, robots. Specifically, the user may want to teach robots task where their execution depends on the state of another agent. To this end, we introduce an extension to the above approach allowing robots to sparsely interact with one another to complete their goals.

Finally, a new set of challenges arise when we consider long-term deployed robots that acquire large libraries of tasks from many users. For instance, it becomes useful for users to be able to correct a task previously taught to the robot. Additionally, it becomes unlikely that an individual knows all of the tasks the robot has previously acquired. However, it becomes very likely that any *additional* task provided to the robot is similar to one it already knows. This motivates an approach where the robot recognizes when it is being provided a task similar to one it already knows and proactively recommends *autocompletions* to assist the user .

1.1 CoBot and Baxter

We demonstrate our work using a CoBot mobile service robots and a Baxter manipulator. The CoBot service robot has an omnidirectional mobile base, camera, laser range finder, microphones, and Microsoft Kinect sensors. Users interact with CoBot through its touch screen and microphone [4, 38]. Baxter is equipped with two 7-DOF arms and a Microsoft Kinect. Users interact with Baxter by typing text on a keyboard. Both robots are shown in Figure 1.1.



Figure 1.1: CoBot (left) and Baxter manipulator (right) that our approach is tested on.

CoBot has many robust robot primitives to navigate within our multi-floor building performing tasks like delivering messages, transporting items, and escorting visitors. Our approach allow us to compose these primitives to request and teach multi-step tasks without resorting to writing code. They help make CoBot usable without programming.

By contrast, Baxter is confined to a controlled workspace where a fairly small set of robot primitives can be composed to define most of the tasks we ask of it. With just a few primitives for grasping and manipulating objects, we have used our approach to specify many different tasks for Baxter.

1.2 Outline

This thesis is presented in three parts that address the challenges discussed above.

In Chapter I, we introduce *Instruction Graphs*, a task representation for incrementally composing robot sensing and actuation primitives in sequences, conditionals, and loops. Instruction Graphs are executed on a robot by an interpreter in place of directly writing code. We demonstrate Instruction Graphs with examples from our CoBot mobile service robot and Baxter manipulator. Then we present demonstrations of people using Instruction Graphs to teach the robots.

In Chapter II, we extend Instruction Graphs by associating STRIPS preconditions and effects

with each robot primitive. We show how this enables tasks requiring *sparse* multi-robot interaction. Such tasks require infrequent, high-level cooperation, instead of a tight-coupling at every decision step. To accomplish this, robots condition on the state of one another by formulating STRIPS queries.

In Chapter III we consider task management for long-term deployed robots with libraries of tasks. First, we present an approach for allowing users to correct Instruction Graphs during test executions. Then, we consider a robot taught by many users, where any specific user may not understand all of the capabilities of the robot. We present an algorithm for performing structure-based generalization of Instruction Graphs to learn general classes of tasks. Then, we present an algorithm that suggests task autocompletions given the initial steps of a new task. We show that these task autocompletions lead to a significant reduction in the number of new task steps that need to be provided during teaching episodes.

We conclude with a discussion of the contributions and directions for future work. Additionally, in Appendix B, we present preliminary work on using planners with our task representations.

Chapter 2

Instruction Graphs

In this chapter, we introduce Instruction Graphs, a graph-based task representation for creating new tasks by composing robot actuation and sensing primitives. Instruction Graphs can be saved, loaded, and executed on a robot through language interactions.

An overview of the approach is shown in Figure 2.1. Our approach consists of a natural language input module (either through speech or by typing), a parser that processes the raw language instructions and converts them to the Instruction Graph representation, and an execution module that takes the graph-based representation, translates the generated behaviors into robot primitives, and executes the task.

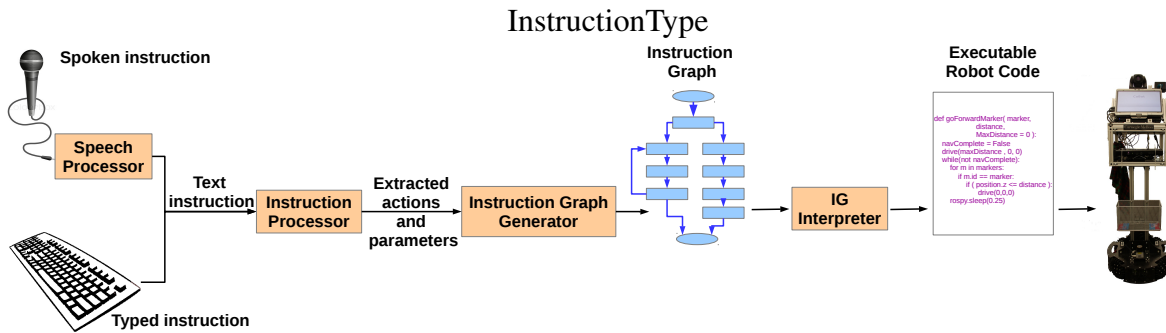


Figure 2.1: Overview of our task specification approach.

In the remainder of the chapter, we first define the Instruction Graph task representation, and then explain how they are executed and constructed from natural language interactions. In each section, we provide real-world examples from CoBot and Baxter. Then, we demonstrate the usability of our approach with a group of users. Finally, we present an overview of related work and a summary of the work in this chapter.

2.1 Formalization of Instruction Graphs

In this work, we model the actuation and sensing capabilities of a robot as *robot primitives*. Formally, a robot primitive is a function f , parameterized by a set of parameters P , that performs sensing or actuation on the robot. Additionally, each primitive has an associated language description of its inputs, outputs, and functionality. For instance, $SetGripper(x)$ is an actuation primitive for Baxter with a language description “Opens gripper x cm”.

Tasks are represented as compositions of sequences, conditionals, and loops of available robot primitives using a special graph structure we developed called an Instruction Graph (IG). An Instruction Graph is a graph $G = \langle V, E \rangle$, where the vertices V correspond to instructions provided by the user, and the edges E correspond to transitions between the instructions. For $V = \{v_i \mid i \in [0, n]\}$, vertices v_1 through v_n correspond to robot sensing and actuation primitives requested by the user, and v_0 is a special starting node from which execution begins. A directed edge $(v_i, v_j) \in E$ denotes a possible transition from the instruction at v_i to the instruction in v_j .

Each vertex is a 3-tuple of the form $\langle ID, InstructionType, Action \rangle$. The first field (ID) corresponds to a unique identification number. In practice, input i corresponds to vertex v_i which has ID i . So, the ID field also specifies the vertex’s relative order in execution. The *Action* field of a vertex defines a parameterized robot primitive that can be executed by an interpreter. Lastly, the *InstructionType* field describes the type of action being executed to distinguish between open-loop, closed-loop, conditionals, loops, or references to other Instruction Graphs. In practice, this is used during execution to determine how the return value of a robot primitive should be interpreted. There are five defined instruction types that a vertex can have:

- **Do Action:** Refers to actions performed completely in open loop.
- **DoUntil Action:** Refers to actions with a closed-loop sensory component. Execution continues until a specific condition about the sensory input is satisfied.
- **Conditional:** Refers to vertices with more than one outgoing edge, representing a fork in the flow of execution. Typically, the *Action* contains a robot sensing primitive and a condition that is evaluated to determine which branch of execution to follow at runtime.
- **Loop:** Similar to conditionals, but one path of execution has a back-edge, repeating a set of instructions while a condition is true.
- **Reference:** Refers to vertices where the *Action* field references to another Instruction Graph, allowing for the construction of hierarchical tasks.

When the distinction between open-loop and closed-loop actions is not important, we sometimes group the *DoUntil* and *Do* instructions together, simply as *Actions*. *Do*, *DoUntil*, and *Reference*

vertices compose actions, or groups of actions, sequentially. By contrast, *Conditional* and *Loop* vertices change the flow of execution. Using them, we can represent standard programming constructs such as if-statements, while-loops, and do-while loops.

2.1.1 Execution of Instruction Graphs

Instruction Graphs are executed by an interpreter that traverses the graph, executing robot primitives and evaluating conditionals along the way. The traversal operation starts from the first node, v_0 and follows the transitions shown in Algorithm 1. These transitions are based on the *InstructionType* and *Action* of each vertex. In particular, *Do* and *DoUntil* vertices have one out-edge, which is followed after the action is performed (Lines 6, 10). *Conditionals* and *Loops* have two out-edges and an *Action* field that evaluates to true or false at runtime. The interpreter transitions along the first edge if the condition is true, and the second edge if it is false (Lines 14, 16). For *Reference* nodes, the execution function is called recursively on the referenced graph (Line 20).

Algorithm 1 Executing a task.

```

1: procedure EXECUTEIG( $G$ )
2:    $v_{current} = G.vertices[0]$ 
3:   while  $v_{current} \neq \emptyset$  do
4:     if  $v_{current}.instructionType == \text{"Do"}$  then
5:       executeAction( $v_{current}.action$ )
6:        $v_{current} \leftarrow v_{current}.child[0]$ 
7:     else if  $v_{current}.instructionType == \text{"DoUntil"}$  then
8:       while  $v_{current}.senseCondition$  is not True do
9:         executeAction( $v_{current}.action$ )
10:         $v_{current} \leftarrow v_{current}.child[0]$ 
11:      end while
12:     else if  $v_{current}.instructionType == \text{"Conditional"}$  or  $\text{"Loop"}$  then
13:       if evaluateConditional( $v_{current}.action$ ) then
14:          $v_{current} \leftarrow v_{current}.child[0]$ 
15:       else
16:          $v_{current} \leftarrow v_{current}.child[1]$ 
17:       end if
18:     else if  $v_{current}.instructionType == \text{"Reference"}$  then
19:       executeIG(loadIG( $v_{current}.reference$ ))
20:        $v_{current} \leftarrow v_{current}.child[0]$ 
21:     end if
22:   end while
23: end procedure

```

2.1.2 Instruction Graphs on CoBot and Baxter

In this subsection, we show several tasks that have been taught to CoBot and Baxter with Instruction Graphs. Before describing the tasks, we briefly describe some of the primitives available on the robots. On both robots, action and sensing primitives are implemented as programming functions defined in separate source files, which are loaded when the robot is initialized.¹ Thus, the *Action* of a vertex contains a function and its parameters.

On CoBot, we have several robot primitives such as:

- *Move*: Executes open loop forward and turning motions specified by the tuple:

$$m = \langle \Delta_x, \Delta_y, \Delta_\Theta, v_t, v_r \rangle$$

where Δ_x , Δ_y , and Δ_Θ represent the requested forward, lateral and angular displacement. v_t and v_r represent the maximum translational and rotational velocities respectively.

- *MoveTo*: Positions the CoBot a provided distance away from a landmark in the CoBot's field of view.
- *Say*: Allows CoBot to speak a provided text message.
- *IsLandmarkVisible*: Conditions on the presence of a landmarks.

Baxter has several primitives such as:

- *Move*: Executes open loop motion to move an arm to a location in 3D Space.
- *MoveToL*: Moves an arm to the location of a visible landmark.
- *SetGripper*: Opens or closes the gripper on a given arm.
- *Display*: Shows a text message on Baxter's screen.
- *IsLandmarkVisible*: Conditions on the presence of a landmark.

Appendix A shows the language descriptions for each of these robot primitives.

Both robots use Augmented Reality (AR) tags as their primary form of sensory input.² AR tags are visual signs that can be detected through image processing. The robot can detect each tag's unique identification number and the relative 6 degree of freedom pose of the tag with respect to the robot. Thus, the robot can condition on the visibility, relative distance, and relative angle of an AR tag.

¹The Instruction Graph framework, and access to the robot action and sensing primitives are implemented in Python using the Robot Operating System (ROS)

²We use the ROS *ar_track_alvar* package to detect AR tags.

CoBot Following a Visual Landmark

The first task we show illustrates the use of loops to have a continuous behavior as long as the loop condition holds. The purpose of the task is to have the CoBot follow a visual sign until it loses sight of it. Figure 2.2 shows the full Instruction Graph for this task.

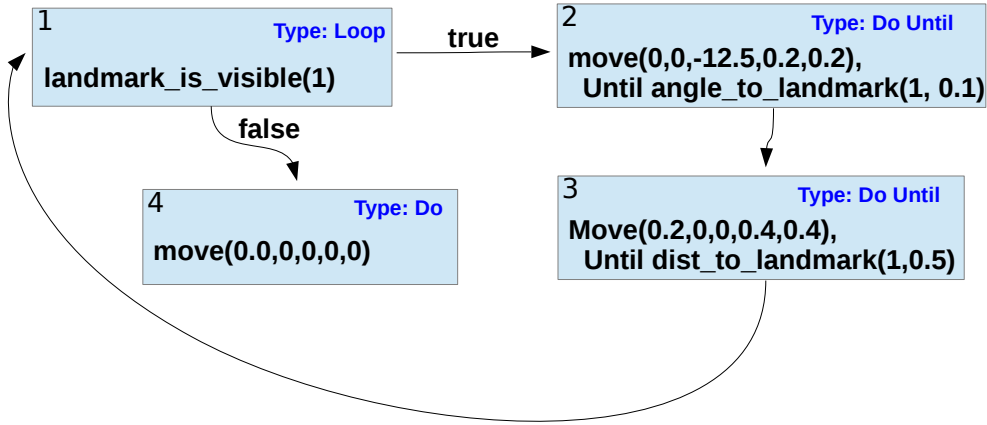


Figure 2.2: Instruction graph for a “follow the sign” task.

At each execution cycle, the CoBot first checks that it can see Landmark 1 (ID 1) and stops moving if it cannot (ID 4). So long as CoBot can see Landmark 1, it first turns towards it, stopping when it is within 0.1 radians of it (ID 2). CoBot then moves forwards until it is 0.2 meters from the landmark (ID 3). Finally the whole process is repeated as long as the visual landmark stays in the sight of the robot. Instead, if the robot loses sight of the landmark, it stops moving (ID 4). Both the turning and going forward motions are conditioned upon the specified visual landmark, therefore they both translate to *DoUntil* nodes.

CoBot Getting Coffee

The second example we present instructs the CoBot how to go to a cafe from a starting point to order a cup of coffee. The task consists of open-loop and closed-loop motions. This task also demonstrates an example use of an *if* clause to change the course of execution depending on the presence of a visual landmark. Figure 2.3 shows the full Instruction Graph for this task.

In the first part of this task, the robot navigates out of our lab and around a bend. It performs these actions in closed-loop using properly placed landmarks to orient itself. The robot conditions on the presence of landmarks to decide when it should stop moving forward or turning (ID 1,2). Figure 2.4a shows a moment from the “get coffee” task execution where the CoBot is navigating the bend. The motion commands in the first part of this task are conditioned on sensing landmarks, so they are translated into the *DoUntil* instruction type.

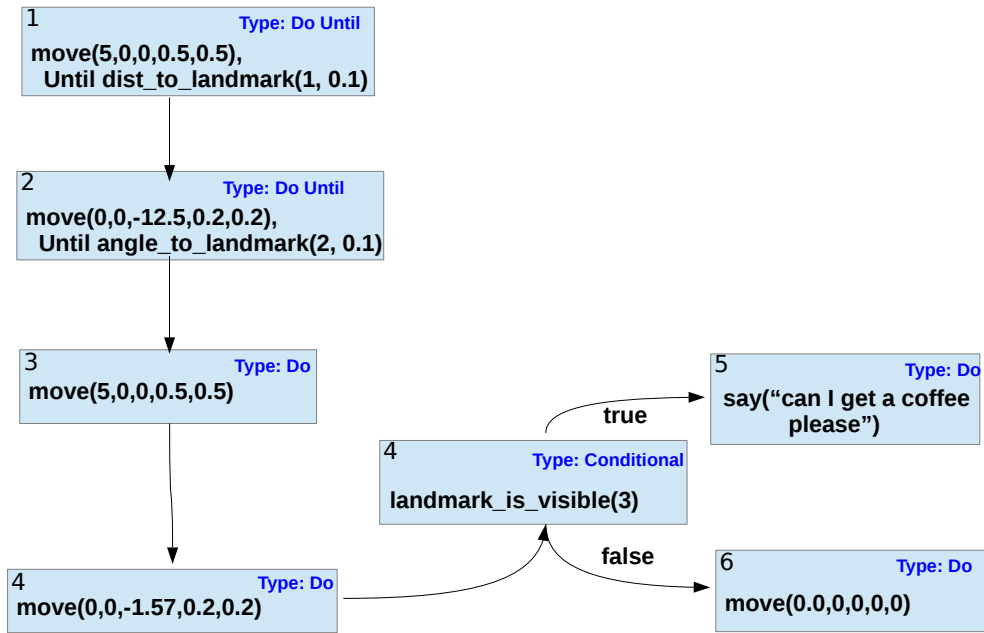


Figure 2.3: Instruction Graph for the “Get Coffee” task.

In the second part of the task, the robot approaches the cafe by going forward past the bend and turning to the left (ID 3,4). Then, it checks for the presence of a visual landmark (ID 4). If the landmark is visible, the robot infers that the cafe is open, and therefore it proceeds with the ordering (ID 5). Otherwise, the robot infers that the cafe is closed, and waits in place for further instructions (ID 6). Figure 2.4b depicts the CoBot checking to see if the cafe is open.

The motion in the second part of the task does not depend on sensory input, so they are translated into the *Do* instruction type, and are executed in open-loops. The check to determine if the cafe is open is translated into an *If* clause, conditioned on the visibility of the specified visual landmark. Finally, speaking is also translated into a *Do* node.

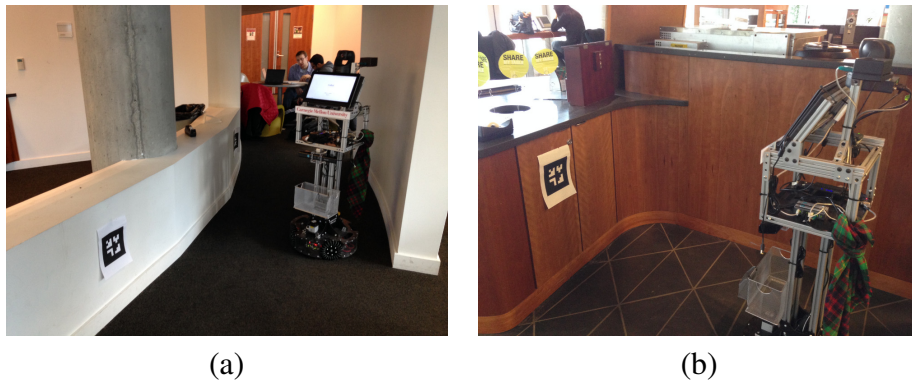


Figure 2.4: (a) The CoBot navigates a bend by orienting itself with respect to several landmarks. (b) The CoBot checks if the cafe is open by conditioning on the presence of a landmark.

Baxter Storing an Object

The third example we present illustrates hierarchically composing Instruction Graphs. The purpose of this task is for Baxter to pick up an object near one landmark and store it near another landmark. Specifically, we define an Instruction Graph for Baxter to pick up an object at Landmark 1. Then, we reference this Instruction Graph to have Baxter pick up the object and store it at Landmark 2.

Figure 2.5a depicts the Instruction Graphs for moving to an object at Location 1 (ID 1) and closing the gripper (ID 2). Moving to a landmark is a closed-loop action, which terminates when the gripper is 0.01 meters from the object. The tolerance is very small because this landmark represents the location of an object the gripper will pick up.

Then, the Instruction Graph in Figure 2.5b references this task to pick up the object (ID 1), moves to location 2 (ID 2), and open the gripper (ID 3). Moving to Landmark 2 is a closed-loop command, that terminates when the gripper is 0.1 meters from the location. The tolerance is larger here, because it is okay for Baxter drop the object within 10 centimeters of the target.

By representing the “Pick Up” task as a separate Instruction Graph, it can easily be referenced in other tasks that require the same functionality. For instance, we can reuse it in a task that stores the object near another Landmark, or even in CoBot’s basket.

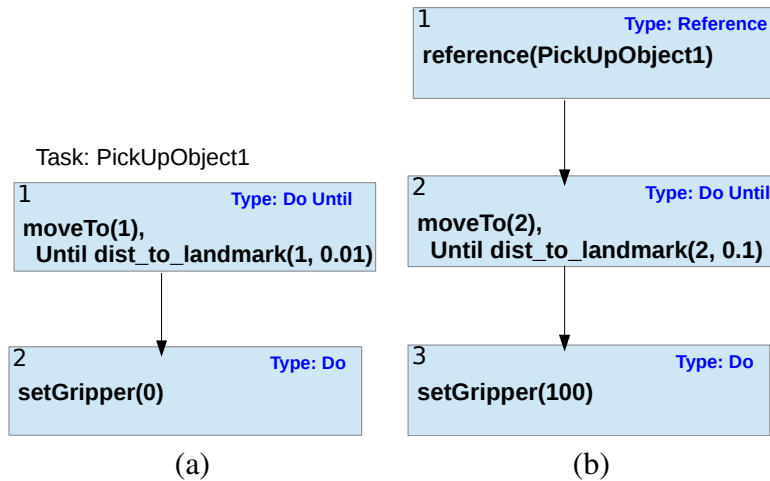


Figure 2.5: Instruction Graph for picking up an object (a) and dropping it at another location (b).

2.2 Generating Instruction Graphs from User Input

The user creates an Instruction Graph by incrementally providing instructions to the robot. The robot processes each instruction and updates the best accordingly. An important issue is the mapping from natural language to modifications of the Instruction Graph. In this section, we discuss our approach for grounding natural language to modifications of the Instruction Graph, and present examples from our implementations on Baxter and CoBot.

2.2.1 Generating Instruction Graphs

We expect commands to have loose syntax. Specifically, they must contain a keyword and any parameters in a predefined order. This keyword parser ensures that there is always a constrained natural language instruction that grounds to each action. First, we associate keywords to *InstructionTypes* and each robot primitive. We also associate keywords to common programming constructs like If-Statements, While-Loops, and Do-Whiles, as shown in Table 2.1. When given a natural language instruction, we attempt to ground the instruction by checking if these keywords are provided, with any parameters in an expected order.

Instruction Type	Keyword(s)
Do	No Keyword
DoUntil	“until”
Reference	“reference”
If-Statement	“If”, “end if”
Do-While	“do while”, “end while”
While	“while”, “end while”

Table 2.1: Instruction Types and programming constructs with their corresponding keywords.

After extracting the instruction type, robot primitive name, and corresponding parameters (if any), the robot can update the current Instruction Graph. Since instructing a task is an interactive process, the robot asks for confirmation for each inferred action. After each confirmation, a relevant node is created and added to the current Instruction Graph.

Algorithm 2 shows the algorithm for creating *Do*, *DoUntil*, and *Reference* nodes. It is straightforward as these vertices have one out-edge. Algorithm 3 shows the algorithm for creating conditionals. This is more complicated as closing the body of a conditional requires knowledge of its starting location. To accomplish this, we use a stack to keep track of the scope of each conditional. The first function creates the initial branch in execution and begins accepting instructions for when the conditional is true. The second function transitions to accepting in-

structions for when the conditional is false. The last function closes the conditional by having the ends of both branches connect to the same vertex.

Algorithm 2 Creating a Node for *Do*, *DoUntil*, or *Reference* Commands.

```

1: procedure ADDACTUATION( $v_{current}, id, instruction$ )
2:   ( $function, params$ )  $\leftarrow$  Parse( $instructions$ )
3:    $instructionType \leftarrow RobotPrimitiveType(function)$ 
4:    $id \leftarrow id + 1$ 
5:    $v \leftarrow Vertex(id, instructionType, function, params)$ 
6:    $v_{current}.child[0] \leftarrow v$ 
7:    $v_{current} \leftarrow v$ 
8: end procedure

```

Algorithm 3 Creating Conditional Nodes

```

1: procedure BEGINCONDITIONAL( $stack, v_{current}, id, instruction$ )
2:   ( $function, params$ )  $\leftarrow$  Parse( $instructions$ )
3:    $v \leftarrow Vertex(id, "Conditional", function, params)$ 
4:    $stack.push(v)$ 
5:    $v_{current}.child[0] \leftarrow v$ 
6:    $v_{current} \leftarrow v$ 
7:    $id \leftarrow id + 1$ 
8: end procedure
9: procedure NEXTCASE( $stack, v_{current}, id$ )
10:   $v_{conditional} \leftarrow stack.pop()$ 
11:   $stack.push(v_{current})$ 
12:   $v_{current} \leftarrow v_{conditional}.child[1]$ 
13: end procedure
14: procedure ENDCONDITIONAL( $stack, v_{current}, id$ )
15:   $v_{TrueBranch} \leftarrow stack.pop()$ 
16:   $v = Vertex(id, "", no-op, \emptyset)$ 
17:   $v_{TrueBranch}.child[0] = v$ 
18:   $v_{current}.child[0] = v$ 
19:   $v_{current} = v$ 
20:   $id \leftarrow id + 1$ 
21: end procedure

```

For loops, a similar algorithm with a stack is used. However, when closing the true branch of a loop, a back-edge is created that redirects back to the condition. The difference between *While* and *DoWhile* constructs is whether to place the loop condition at the beginning or end of the loop. In the case of a *While* loop, the *Loop* node is inserted at the beginning and in the case of a *DoWhile* loop, it is placed at the end of the body. This decision influences whether the loop

is guaranteed to run at least once (*DoWhile*).

We also have three special commands:

- **Shutdown:** This command terminates the task execution.
- **Save:** Saves the current task in memory to a file under a specified name.
- **Load:** Loads a previously saved task to be run or merged with the current task.

The first two commands are straightforward. The *Shutdown* command terminates any tasks being executed, and also ends any teaching episodes. The *Save* command allows a fully-taught Instruction Graph to be stored as a file for later use. If the user attempts to save an Instruction Graph with incomplete conditionals or loops, the robot will instead ask them to finish teaching the task.

The *Load* command allows a previously saved task to be loaded into memory, either to run the task, or to combine it with the task currently being taught. Sometimes, it is desirable to create the union of two Instruction Graphs, instead of using *References*. For instance, a user may want their task to be independent of any future changes to a dependency. Given that the user is creating an Instruction Graph $G = (V_g, E_g)$, we denote the most recently created node as v_{g_k} . The *Load* command loads another Instruction Graph $H = (V_h, E_h)$ by creating:

$$Load((V_h, E_h)) = (V_h \cup V_g, E_h \cup E_g \cup \{(v_{g_k}, v_{h_0})\})$$

The resulting graph is the union of G and H with one additional edge connecting the most recently created vertex of G to the source-vertex of H . Then, the IDs of the vertices in H are updated so that no two vertices have the same ID, and the user continues teaching from the last instruction in H .

2.2.2 Generating Instruction Graphs on CoBot and Baxter

The user interacts with Baxter through typing. However, users can instruct CoBot through speech. The first step on CoBot is to convert speech to text through the use of an automated speech recognizer.³

Next, the parser processes the input, and grounds to a robot primitive function call with proper parameters. First, the rule-based parser that checks for the presence of keywords and parameters in an expected order. Table 2.2 and Table 2.3 show sample keyword assignments for actions on CoBot and Baxter.

³We use the Google Speech Recognition API for our deployments.

Instruction Type	Action	Keyword
Do	move forward	“forward”
Do	turn the robot	“turn”
Do	speak to the user	“say”
DoUntil	move forward in closed-loop	“forward”
DoUntil	turn in closed-loop	“turn”

Table 2.2: Sample actuation commands and corresponding keywords on CoBot.

Instruction Type	Action	Keyword
Do	move [arm] to (x, y, z)	“move”
Do	open [arm] gripper	“open”
Do	close gripper	“close”
Do	display a message	“say”, “display”
DoUntil	move [arm] to landmark	“move” + “landmark”

Table 2.3: Sample actuation commands and corresponding keywords on Baxter.

For example, a rule maps the keyword “forward” to a specific parameterization of the CoBot’s *Move* primitive. Likewise, the keywords “say” or “display” map to Baxter’s *Display* function. Then, to determine the parameters of our functions, we make a simplifying assumption that they appear near these keywords. For example, if the user mentions the word “forward”, we assume that the nearest numerical value next to the “forward” keyword is the distance in meters denoting the distance that the robot should move forward. Similarly if the user mentions the word “turn”, first we look for the direction specifier keywords “left” and “right” next to the “turn” keyword. Once we find the direction specifiers, the first numerical value after the direction specifier is considered as the amount of turn towards the specified direction in degrees. We note that the users specify angles in degrees, but that the robots internally represent angles as radians. We made this choice because users find it easier to reason in degrees than in radians.

This filtering approach allows us to interpret the instructions such as “turn right for 60 degrees” and “turn to your right 60 degrees” the same without having to examine the instruction grammatically or run sophisticated parsing techniques. In the future, we may examine statistical parsing techniques as well.

Once the function call and parameters are determined, a new vertex is created and appended to the current Instruction Graph. This approach makes defining the robot primitives and processing the user input completely independent from the Instruction Graph implementation. New robot primitives or sensing components can easily be added in just two steps. First, the user provides the functions for accessing the action primitive or the sensory input. Then, they specify keywords for the primitive, and an expected list of parameters.

Figure 2.6 shows a textual interaction for teaching the “Follow the Sign” task. Likewise, Figure 2.7 shows an interaction for teaching the “Get Coffee” task. Lastly, Figure 2.8 shows an interaction for teaching Baxter to pick up and store objects. In all of these interactions, we note that after the robot infers the meaning of a verbal instruction, they confirm their understanding with the user. If the robot misinterprets an instruction, the user can say that the instruction was “Incorrect” and re-provide it to the robot. We also note that our Baxter only has one gripper, which is why the user is not required to specify which arm should open or close its gripper.

```
User : While landmark 1 is visible
Robot: What should I do while "Landmark 1 is visible"?
User : Turn until landmark 1 is at 0 degrees
Robot: What should I do after "Turn until Landmark 1 is at 0.0 degrees"?
User : Move forward until you are 0.5 meters from landmark 1 max 0.2 meters
Robot: What should I do after
      "Forward at most 0.2 meters until 0.5 meters from Landmark 1"?
User : End Loop
Robot: Loop Ended. What should I do next?
User : Stop
```

Figure 2.6: The instruction conversation for the “follow the sign” task, provided by speech.

```
User : Move forward until you are 0.1 meters from landmark 1 max 5 meters
Robot: What should I do after
      "Forward at most 5 meters until 0.1 meters from Landmark 1"?
User : Turn right until you are 30 degrees from landmark 2.
Robot: What should I do after "Turn until Landmark 2 is at 30.0 degrees"?
User : Forward 5 meters
Robot: What should I do after "Forward 5 meters"?
User : Turn left
Robot: What should I do after "Turn -90.0 degrees"?
User : If landmark 3 is visible
Robot: What should I do if "Landmark 3 is visible"?
User : Say ``I would like to order a cup of coffee"
Robot: What should I do after "say I would like to order a cup of coffee"?
User : End If
Robot: What should I do in the other case?
User : Shutdown
```

Figure 2.7: The instruction conversation for the “Get Coffee” task, provided by speech.

```

User : Move to landmark 1
Robot: What should I do after ``Move to Landmark 1"?
User : Close your gripper
Robot: What should I do after "Close Gripper"?
User : Save task as Pickup1.
Robot: Task Saved as "Pickup1".
User : Reference task Pickup1.
Robot: What should I do after referencing ``Pickup1"?
User: Next move to Landmark 2
Robot: What should I do after ``Move to Landmark 2"?
User: Finally, open your gripper.
Robot: What should I do after ``Open Gripper"?
User: Run the task.

```

Figure 2.8: The instruction conversation for Baxter to store an object, provided by keyboard.

2.3 User Demonstrations

In this section, we present demonstrations of people using the Instruction Graph framework. Specifically, we asked several users to teach a series of tasks to our mobile service robot. In this section, we first describe the details of the tasks, and then discuss our observations.

2.3.1 Participants

We focused our survey on ten participants in our research group who were familiar with robots but unfamiliar with the task teaching framework. We targeted this audience because they could compare the Instruction Graph framework to programming a robot’s behavior directly. The aim was to see how these people would compare the experience of writing code to using Instruction Graphs, which provide a higher level of abstraction. Furthermore, before testing with general users, we wanted to learn what issues roboticists would find with our approach.

2.3.2 Procedures

We provided the participants a list of commands the robot understood, each with a description and sample user input. Then, users were asked to teach the CoBot four tasks in increasing order of difficulty. The tasks were designed to build upon one another, each utilizing a new robot primitive or feature of the Instruction Graphs. Before each task, the participant watched a demonstration of the new features.

- **Task #1 - Open-loop Motion:** The participants were shown how to teach the CoBot to move forward and to turn. Then, they were asked to instruct CoBot to move in a square.
- **Task #2 - Closed-loop Motion:** The participants were shown how to teach the CoBot to

search for and navigate towards landmarks. Then, they were asked to instruct CoBot to perform waypoint navigation to avoid an obstacle. Figure 2.9 shows a user teaching the CoBot to perform waypoint navigation.

- **Task #3 - Conditionals:** The participants were shown how to teach the CoBot to condition its actions based on the presence of a landmark. Then, we simulated a scenario where landmarks denoted hallways closed for construction work. We asked the participant to teach CoBot to navigate to a location if the hallway was open for use.
- **Task #4 - Loops:** The participants were shown how to teach the CoBot to repeat a task while a landmark was visible. Then, they were asked to teach CoBot to follow a person. To help the participant, the person was wearing an Augmented Reality tag.



Figure 2.9: A user teaches CoBot to use waypoint navigation to avoid an obstacle.

During each task, we recorded the time it took each participant to teach the CoBot and the number of times they restarted. At the end of the experiment, we asked the participants to describe their experience.

2.3.3 Analysis

All users learned how to teach the CoBot very quickly. Specifically, the users were able to teach CoBot the four tasks in a reasonable amount of time (below 5 minutes), with few restarts (below 3 restarts). Figure 2.10 shows the average time and number of restarts necessary to complete each trial.

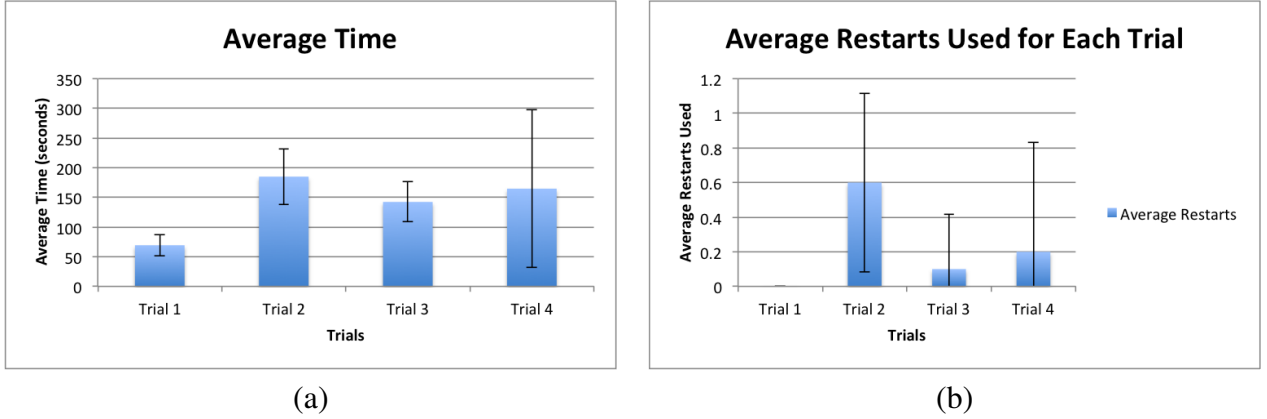


Figure 2.10: Average time (a) and average number of restarts (b) for participants to complete each trial. Notably, the average times are all under five minutes. And, the average user did not need to restart.

2.3.4 Discussion and Motivation for Future Work

After each participant completed the four tasks, we asked them for qualitative feedback on their experience with the Instruction Graph framework.

All of the participants agreed that writing code is generally a more expressive medium for teaching a robot. However, for tasks involving a composition of robot primitives, they were all pleased with the expressivity of the language. On average, teaching each of these tasks required fewer than ten sentences. Performing the same exercises in a programming language would require many more lines of code.

We found that many users had difficulty understanding the range limitations of sensors. These participants tried to search for landmarks far outside the range of our sensors. They also had trouble estimating distances for open-loop commands. Since these people were also familiar with robots, we will need to address this problem. We may add a command to visualize some sensor data to give the users a better understanding of their limitations.

Several participants noted that it was difficult to remember each step of a complex task. In response to this feedback, we have introduced a command that allows the user to visualize the Instruction Graph that the robot is executing. In the future, we may add support for modifying the Instruction Graph through a graphical interface.

Even though the participants were programmers used to working with constrained commands, some desired the ability to teach tasks with freeform speech. We have designed our approach modularly so that any natural language parser can be added in the future. In particular, we are interested in investigating a hybrid approach that combines our keyword-based parser with a probabilistic one that learns from a corpus of data. For instance, one proposed probabilis-

tic parser represents a sentence as a sequence of *frames* [24]. Each frame consists of an action and variable number of parameters. Sentences are parsed into frames with a model based on a conditional random field. Phrases in a frame are grounded to specific actions and parameters by keeping track of the frequency in which each phrase grounds to each symbol.

These trials with roboticists show three important directions for improvement for non-programmers instructing tasks. First, it is difficult for users to understanding the sensing and actuation limitations of a robot, even if they are familiar with it. Secondly, it is difficult for users to remember a very long task taught to a robot verbally. Thirdly, more advanced natural language processing approaches can improve the quality of the user-robot interactions.

2.4 Related Work

The work in this chapter draws upon ideas from two research topics: providing commands to robots through natural language and incrementally teaching tasks to a robot.

Approaches for commanding robots through natural language can roughly be divided into two groups. Some approaches allow unconstrained natural language with probabilistic grounders, and others require verbal instructions to have a specific syntax. The former approaches leverage probabilistic models trained on a labelled corpus. For instance, one method uses Spatial Description Clauses (SDC) to parse a spoken natural language command, extracting the spatial information and grounding to actions that can be executed by the robot [23]. Generalized Grounding Graphs (G^3) have been proposed to parse unconstrained natural language commands [44]. In their work, a Conditional Random Field (CRF) is trained and used to infer the most likely G^3 representation of a natural language command. Similarly, a parser that uses statistical machine translation methods has been presented for enabling a robot to follow navigation instructions [29].

The other set of approaches require commands to have a specific syntax or structure, which mitigates the ambiguity of unconstrained natural language at the cost of robustness. For instance, a system has been proposed for instructing a robotic forklift to handle cargo in which commands are expected to have a specific syntax [9]. Our approach employs a keyword-based parser that expects commands to contain certain keywords and parameters in order. This ensures that the system can ground constrained natural language commands without a large training corpus.

Researchers have also proposed approaches for updating plans generated by a planner based on user commands [7]. Furthermore, some approaches utilize gestures along with the verbal instruction for issuing commands to a robot [41]. Their parser relies on well-defined grammar, and the extracted lexical items are then mapped on the robot primitives. The main difference

between our approach and these aforementioned works is that these approaches do not allow robots to learn from the provided instructions, as the instructions are merely used to operate the robot without saving them for future use.

Many researchers have focused on the specific problem of teaching reusable tasks to a robot. Some methods rely on Learning from Demonstration to teach the task [3]. In these works, the robot learns how to perform a task from user demonstrations. The tasks are often represented as policies that determine state-action mappings [2, 8, 33, 37]. Our work differs from these approaches, because our goal is to teach a task by combining robot primitives through verbal instructions, instead of providing demonstrations.

Others have proposed a framework for automatically generating controllers for executing a task specified as Linear Temporal Logic (LTL) formulas [25]. The input formulas are processed to generate a discrete finite state automata based controller. The resulting discrete controller is then used in conjunction with a set of continuous controllers that implement each state’s low level behaviors. Representing a task as a state machine, and associating states with continuous robot primitives is similar with our approach. However, their method requires a formal description of the task in logic formulas, whereas we focus on accepting natural language instructions.

Researchers have studied the problem of teaching a task by mapping verbal instructions to sequences of robot action primitives [28, 34]. The later work additionally supports revising the taught task and generalizing the task representations over multiple demonstrations of the same task [34]. In both works, mapping natural language instructions to action primitives resembles our approach. However, our task representation differs as our primitives are parameterized, and our tasks have conditionals and loops.

Another method represents tasks as a directed acyclic graph, where a branch in the tree corresponds to a conditional statement. This method was introduced for teaching tasks composed of available action primitives to a service robot using spoken verbal instructions [40]. The main difference between our approach and this work is that our representation allows for repetitions (cycles) in the task and for hierarchical tasks. Their approach also requires a strict syntax for commands.

In contrast to the aforementioned approaches that represent tasks as a composition of robot primitives, another approach translates natural language instructions into formal logic descriptions of goals and actions [10]. Instruction parsing is done through the use of predefined associations between the lexical items in the instructions and the corresponding λ -expressions. One immediate difference between our approaches is that our task representation is graph-based.

Researchers have addressed the problem of teaching soccer skills to robots via spoken language [46]. In this work, natural language commands are mapped to a predetermined set of

robot actions. An interesting aspect of the proposed framework is that the teacher can query and condition on the robot’s internal state, which is similar to our work. Their vocabulary includes a set of actions like shoot and pass, and if-then-else control expressions that can be coupled with queries about the state features.

Finally, our work is most similar to teaching tasks represented as Petri Net Plans [13]. One very interesting aspect of their approach is that the user can teach a parameterized tasks to the robot, where the parameters are instantiated by a user at runtime. Additionally, they represent tasks as Petri Nets instead of graphs, which enables them to naturally support parallelism. One difference is that their approach uses grammars to ground verbal commands.

2.5 Summary

In this chapter, we presented an approach for teaching a task, represented as a composition of robot primitives with sequences, conditionals, loops, and hierarchical references. The tasks are taught through verbal instructions. Then, we asked users to teach tasks to our CoBot mobile service robot to see the effectiveness of our approach. The feedback was generally positive and gave us several directions for future work such as using statistical natural language parsing techniques and allowing users to modify Instruction Graphs via a GUI.

In the next chapter, we extend the Instruction Graph task representation to support tasks robot-primitive preconditions and effects. This enables us to to represent tasks that require multiple robots to cooperate to complete their goals.

Chapter 3

Multi-Robot Sparse-Coordination

In the previous chapter, we presented an approach for incrementally providing a task to one robot. In this chapter, we consider a related problem where a robot’s task may require coordination with others. In particular, we focus on tasks where robots can complete most of their tasks independently, and coordinate infrequently. In the literature, the concept of coordinating while not being coupled at every decision step is known as *sparse-coordination* [30]. In terms of task representation, sparse-coordination represents the joint state space only when the robots need to cooperate.

Our goal is to enable heterogeneous robots, acquiring tasks from different providers, to solve problems requiring sparse coordination. To accomplish this, we first extend the Instruction Graph representation by incorporating robot primitive preconditions and effects, represented in a shared domain language. Then, we present an approach to sparsely coordinate robots by allowing them to condition on each other’s state by making queries.

In the next section, we introduce the technical details of our approach. Then, we present demonstrative examples, a review of related work and conclude with a summary of the contributions and a brief discussion of future work.

3.1 Instruction Graphs for Sparse Coordination

To sparsely coordinate, robots must keep track of their state and be able to query the state of one another. We define Sparse-Coordination Instruction Graphs (SCIG) as graphs $G = \langle V, E \rangle$ where each vertex v is a tuple:

$$v = \langle id, InstructionType, f, P, Prec, Eff \rangle$$

where the additional elements *Prec* and *Eff* respectively represent sets of preconditions and effects of the robot-primitive *f*. More generally, each function *f* has an associated set of literals $\mathcal{L}_f$ that represents its preconditions and effects. Thus, we define:

$$\mathcal{L} = \bigcup_{\forall f} \mathcal{L}_f$$

as the common domain language used by all of the robots. We represent each literal using STRIPS semantics [11]. In particular, each action adds or removes literals from the robot’s current state. While robots may represent their internal state differently, their primitives express this state in terms of the common set of strips literals, $\mathcal{L}$. Importantly, we only require robot-primitive effects to be defined for literals other robots will condition on.¹

To associate these preconditions and effects to actions, each robot sensing and actuation primitive is defined in the Planning Domain Definition Language (PDDL) [15]. For example, a Baxter manipulator may have an action *pick_up(object_id)*, to pick up an object with a given ID. Internally these objects are represented as a 3D point in space and bounding boxes. However, the effects of the action are to remove the literal *hand_empty*, and then add the literal *holding(object_id)*. Figure 3.1 shows an example PDDL definition for Baxter’s “pickup” action. Currently, we assume that all changes in state are captured by the robot primitives and that each robot can only modify its own state.

```
(:action pickup
:parameters (?x)
:precondition (and (OBJECT ?x)
                  (hand_empty))
:effect      (and (holding ?x)
                  (not (hand_empty)))
```

Figure 3.1: Example PDDL definition for the primitive “pickup”. As preconditions, its parameter must be an object, and the hand must be empty. The effects are that the hand is no longer empty, and the robot is holding an object.

During execution, each robot keeps track of its own state. Specifically, the state predicates are either appended or deleted from the robot’s state according to the effects of the executed action. We introduce a special function *check_literal*, used in Conditional and Looping vertices that can condition on the state of any agent. The *check_literal* function takes as input a unique robot identifier and a query. In our framework the query is represented as a set of STRIPS predicates, possibly composed with the *and*, *or* and *not* operators. The query is routed to the the robot with

¹Appendix B shows how a planner can generate Instruction Graph plans when a complete set of robot-primitive preconditions and effects is provided.

the corresponding identifier.

When a robot receives a query, it is evaluated against its current state. Each robot adopts a closed-world assumption when responding to queries. In particular, the robot checks that positive literals are present in its state and that negated literals are absent in its state. The result of this query is returned to the requesting robot. In this way each robot has only a representation of its own state, and makes no assumptions about the state of another.

Figure 3.2 shows a partial example of a SCIG for a CoBot mobile base, where the *check_literal* function is used to condition on the state of a Baxter manipulator. Specifically, the CoBot will perform the *move_to* action if Baxter’s state does not contain *hand_empty*.

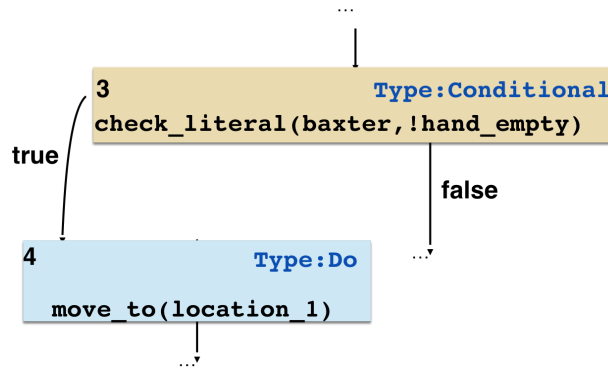


Figure 3.2: Partial example of a SCIG for a CoBot conditioning on the state of a Baxter manipulator. If Baxter’s state does not contain *hand_empty*, CoBot will perform the *move_to* action.

With this approach we are able to implement coordination at a high level. In particular, we define several useful coordination actions from Loops and Conditionals:

- **Wait Until:** The robot waits until another robot is in some state. This is implemented with a Loop.
- **Act Until:** The robot repeats some actions until another robot is in some state. This is implemented with a Loop.
- **Ask:** The robot conditions on the state of another robot. This is implemented with a Conditional.

We provide examples of each of these forms of coordination in the next sections.

3.2 Demonstrative Examples

In this section we show how multiple robots can be taught to perform different tasks that involve sparse-coordination. In particular, we show how a Baxter robot can be instructed to perform a

manipulation task during which its two arms need to sparsely interact with each other. Then, we extend this example by considering also a CoBot mobile base. In this case, we show how the two arms and the mobile service robot can be instructed to deliver and store an object.

3.2.1 Store Task

We first show how the two arms of a Baxter manipulator can be treated as separate agents and coordinated. We denote the arms as *left_arm* and *right_arm*. Table 3.1 shows the set of robot primitives for both arms, with their associated preconditions and effects. Our task is to have the left arm assist the right arm in finding an unobstructed location to place an object.

Table 3.1: Baxter arms primitives with associated preconditions and effects.

Action Primitives	Preconditions	Effects
wave()	-	-
wait(time)	-	-
is_landmark_visible(landmark_id)	-	-
move_to(location)	-	-pointing, -at(old_location), +at(location)
pick_up(object_id)	hand_empty	-hand_empty, +holding(object_id)
drop(object_id)	holding(object_id)	-holding(object_id), +hand_empty
point(location_id)	-	-at(old_location_id), +at(location_id), +pointing, +pointing_at(location_id)

In this example, a user describes the task to each agent in two separate teaching sessions through natural language. Specifically, the left arm is instructed to wave until the right arm picks up an orange wooden block (Figure 3.3a). At this point, the state of the right arm is changed to *holding(object_1)* and the function *check_literal(right_arm, hand_empty)* returns false. After realizing this fact, the left arm starts checking if a landmark can be detected at the drop position (Figure 3.3b). In the case a landmark is detected, the left arm points at it, reaching the *pointing_at(location_1)* state. The function *check_literal(left_arm, !pointing)* now returns false and the right arm drops the block at *location_1* (Figure 3.4a). Instead, when the landmark is not detected the left arm points at an alternative location (*location_2*) where the orange block can be dropped (Figure 3.4b).

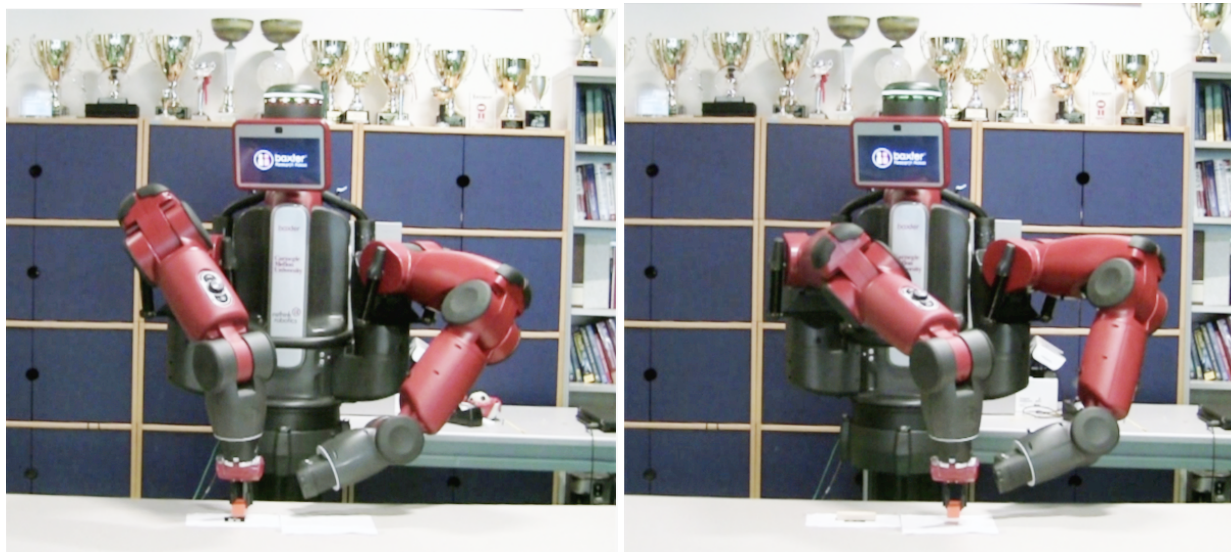
Figure 3.5 shows a natural language description of the task provided to the two arms. Instead, Figures 3.6a and 3.6b show the corresponding Sparse Coordination Instruction Graphs. In this



(a)

(b)

Figure 3.3: (a) The left arm is shown waving until the right arm picks up an orange wooden block. (b) The left arm checks if the drop position is open.



(a)

(b)

Figure 3.4: (a) Since this position is open, the left arm points at it where the object will be placed. (b) In the second run, since the drop position is not open, the left arm points at an alternative position.

example we assume that the two robots have the same high-level representation of the environment, which effectively means that they agree on the position of the two possible locations. The execution of the task can be seen in an online video.²

Left arm:

```

wave while right hand is empty
move to location 1
if landmark 1 is visible
point to location 1
otherwise point to location 2

```

Right arm:

```

pick up object 1
wait while left arm is not pointing
if left arm is pointing at location 1
move to location 1
otherwise move to location 2
end if
drop object 1

```

Figure 3.5: Natural language input provided to the two arms. The names of the agents are shown in red, and their states are shown in blue.

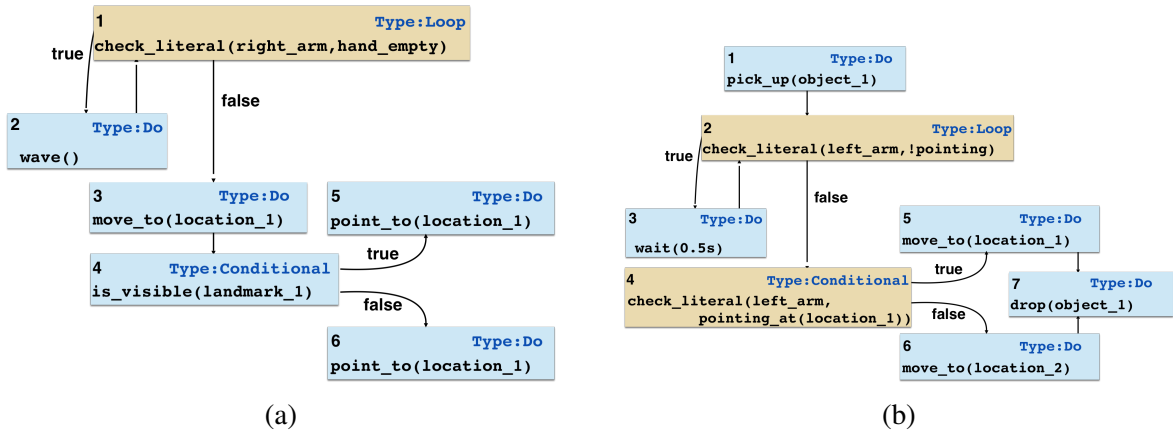


Figure 3.6: Sparse-Coordination Instruction Graphs extracted from the text in Figure 3.5. Specifically, (a) represents the graph for the left arm, while (b) represents the graph for the right arm. The nodes in yellow represent the vertices that require a query to another robot's state.

²<http://youtu.be/4nLjuLhFUvk>

3.2.2 Deliver and Store Task

Next, we extend the previous example by considering also a CoBot robot. CoBot must coordinate with the Baxter’s arms in order to deliver and store an object. To this end, we modify the previous example by making the *right_arm* wait for CoBot to be at a specific location. For simplicity, the *right_arm* is instructed to wait, but it could also be instructed to do other work while waiting for CoBot’s arrival (Figure 3.7a). When CoBot reaches the state *at(location_3)*, the *right_arm* will pick up the delivered object from CoBot’s basket (Figures 3.7b and 3.8a), storing it at the location pointed out by the *left_arm* (Figure 3.8b). For CoBot, we defined the robot primitives shown in table 3.2. For Baxter we used the previously described primitives.

Table 3.2: CoBot primitives with associated preconditions and effects.

Action Primitives	Preconditions	Effects
Say(message)	-	-
Move_to(location)	-	-at(old_location_id), +at(location_id)

The user teaches the tasks to the robots in three separate teaching sessions. Figure 3.9 shows a natural language description of the tasks provided to CoBot and to the *right_arm*. The task given to the *left_arm* is the same as in the previous example. Figure 3.10 depicts the Sparse Coordination Instruction Graph extracted for CoBot. Since the task for the right arm is almost identical to Figure 3.6b, we omit it.



Figure 3.7: (a) CoBot reaches the pick up location while the right arm is waiting and the left arm is waiving. (b) The right arm can now pick up the object.



Figure 3.8: (a) Detailed view of Baxter picking up the object. (b) The left arm points at location 1, the right arm drops the object, and CoBot leaves.

CoBot:

```
move to location 3
say ``I am here to deliver a package"
wait while left arm is not pointing
move to location 4
```

Right arm:

```
wait while CoBot is not at location 3
pick up object 1
wait while left arm is not pointing
if left arm is pointing at location 1
move to location 1
otherwise move to location 2
end if
drop object 1
```

Figure 3.9: Natural language input provided to CoBot and the *right_arm*. The description of the *left_arm* task instead is the same shown in Figure 3.5. The names of the robots are shown in red while their states are shown in blue.

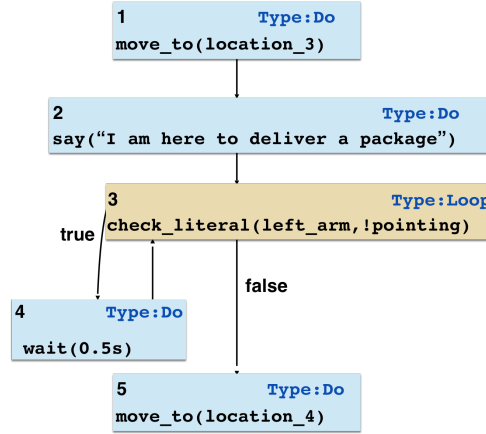


Figure 3.10: SCIG extracted from the CoBot’s task description in Figure 3.9. Node 3 requires a query to another robot’s state, in this case Baxter’s left arm.

3.3 Related Work

Multiple approaches have been proposed to represent multi-agent plans. The simplest solution is to represent the entire joint state space and to define a policy for each robot at each state. However, this is inefficient if robots only need to coordinate infrequently.

There are several approaches to divide joint tasks into smaller tasks that each agent can execute autonomously or as part of a small group. Coordination Graphs compactly represent dependencies between the actions of different agents, thus capturing the local interaction between them [22]. Local interactions have also been exploited to minimize communication overhead during policy execution [39] and in game-theory to obtain compact game representations [18]. These approaches rely on dependency analysis to decompose independent parts of the initial representation. All of these approaches require that a complete plan is given to each robot a priori. In our work, tasks are acquired by the robot incrementally, and the provider specifies the dependencies between different robot tasks.

There are many methods for implementing coordination between multiple agents. These techniques can rely on auction protocols [14], on dedicated architectures [36, 43], and on collections of conventions followed by all team members [42]. In our work, we have a common task representation (Instruction Graphs), and a coordination mechanism based on query the state of other robots.

3.4 Summary

Sparse-Coordination Instruction Graphs allow users to teach a wide-variety of tasks that require multi-agent coordination. In particular, they are well suited to tasks that require high-level cooperation between robots. We have found the approach especially effective with robots that have separate goals. For instance, our fleet of CoBots perform many tasks, such as escorting people and picking up objects. Some of these tasks require brief interaction with Baxter, which has its own goals to accomplish. The robots coordinate infrequently because their goals require a limited amount of interaction.

In this work, we assumed that robot-primitives were fault tolerant and that there were no communication problems. We note that for sparse-coordination, many of the typical problems of multi-robot communication occur less often. For instance, issues of consistency or high-latency are less important for high-level coordination when robots query each other’s state infrequently. However, robots often have faulty sensors and nondeterministic actions. In future work, we may consider other representations of actions and state that reason about this nondeterminism.

Chapter 4

Task Management

In the past chapters, we focused on building a robust task representation for incrementally providing tasks to robots. In this chapter, we instead consider the challenges of one robot learning from multiple people over a long period of time. In particular, we first address the issue of *correcting* a task previously taught to the robot. We introduce an approach for allowing the user to provide corrections during test executions. Then, we consider the problem of efficiently providing an *additional* task to a robot that already has a large library of tasks. The new task is possibly composed of parts of known tasks, or even a complete repetition of a known task. Our goal is to enable the robot to recognize when the new task is similar to a past task in its library, given some initial steps of the new task. Then, the robot can recommend *autocompletions* of the remaining parts of the task. At a high-level, our aim is to allow long-term deployed robots that accumulate experience to use their additional knowledge to improve interactions with a user.

4.1 Task Correction

A major feature of the approach compared to related work is the ability to let the instructor correct parts of the task as desired. Specifically, the robot allows the user to edit the parameters of an action or replace an action with another during test executions. We envision three major reasons why a user may want to correct a portion of a learned task:

- Changing open loop parameters to make instructions more accurate
- Switching from open loop to closed loop, or vice-versa
- Modifying a few instructions of an existing task to populate new tasks (code reuse)

The first example is likely to occur when a parameter value is not as accurate as predicted. This can happen due to miscalculations, unexpected changes, or faulty calibration of the robot. The second example is likely to happen if a robot gains new sensory information, or if a sensor

become unavailable. To that end, we support switching from open loop commands to their closed loop equivalents, and vice-versa. For example, if necessary sensory data are no longer available, it is possible to change the task where the data were used from closed loop to open loop without re-instructing the entire task. Lastly, when writing a new function that is similar to an existing one, it should be possible to reuse the bulk of the task and to only specify the new piece.

In our approach, the task correction and modification occurs during a test execution phase. When the correction mode is set, the robot speaks out the next action and asks for confirmation. If the user wants to correct or change that action, the robot then asks for a replacement instruction. Once the corrected instruction is processed, the original action and parameters are substituted by the new instruction.

Currently, the user is limited to providing vertex-level corrections that modify the robot-primitive and/or parameters in individual vertices. These do not include adding new vertices to the graph or removing existing vertices. We leave for future work the issue of performing these graph-level corrections.

4.1.1 Example Correction on CoBot

If the user observes that the parameters for the open-loop actions are imprecise, he or she can make a revision without re-creating the task from scratch using the debugging and correction mode. In the correction mode, the robot speaks the action description for each action, and asks whether the user confirms that action or wants to make a correction. If the user confirms the action, the robot leaves that action intact and proceeds to the next action. However, if the user decides to correct the action, the robot then asks for a replacement action. Once the user specifies the replacement action with its parameters, the original node is updated as to reflect the correction. Figure 4.1 shows an example interaction for correcting an action on the “get coffee” task presented above.

```
Robot: Go forward for 5 meters.  
Robot: Should I confirm this action or revert it?  
User : Revert.  
Robot: Please give me a new command to  
       replace this with.  
User : Forward 6 meters  
Robot: Action revised. The new action is  
       go forward for 6 meters.
```

Figure 4.1: An example interaction during step-by-step execution to correct an action.

4.2 Task Generalization and Autocompletion

In this section, we first present an algorithm that, given a library of Instruction Graphs, generalizes over structurally-similar parts of the tasks where the robot primitives are parameterized differently. Then, we present an algorithm that, given a library of generalized tasks and the initial steps of a new task, recommends autocompletions.

We note that the general problem of finding labeled subgraph isomorphisms is NP-Hard. To overcome this, our approach first creates a tree-representation of each Instruction Graph. Then, we perform frequent labeled subtree mining on these trees to efficiently find common patterns in the tasks. The tree patterns which are executable by the robot and frequently occurring are added to our library of generalized tasks. For each robot primitive in a generalized task, we maintain a distribution over the parameters of the corresponding instantiated task graphs.

When a new task is being provided to the robot, the robot compares the partial task to those in its generalized task library. If the robot finds a generalized task similar to the partial task it recommends a *task autocompletion* with robot primitive parameters sampled from its distribution. Specifically, the robot demonstrates the proposed autocompletion, which are accepted, modified, or rejected by the user. This process is similar to autocompletion provided by search engines. The result is that not all steps of a new task need to be provided by the user, which reduces the effort of giving a new task to the robot.

In the next sections, we introduce the technical components of our approach, and present results from experiments with Baxter. Then, we present an overview of related work, focusing on past work in task learning from experience. Finally, we conclude the chapter with a review of the contributions and a discussion of future work.

4.2.1 Task Generalization

We consider a robot with primitives that represent its action and sensing capabilities. If the robot has a library of tasks composed of these primitives, our goal is to identify common frequent subtasks and generalize over them with limited user assistance. In this section, we first present an extension to the Instruction Graph framework for representing generalized, parameterized, tasks. Then, we present the details of our algorithms for structure-based task generalization and autocompletion.

We define a general task as a *Generalized Instruction Graph* (GIG). In a GIG, the parameters of some actions are ungrounded. In such cases, we know the type of these ungrounded parameters, but not their value. For each parameter we associate an empirical distribution over all known valid groundings. In the case of a grounded parameter, the distribution always returns

the grounded value. Formally, a GIG is also a graph $GIG = \langle V, E \rangle$ where each vertex v is a tuple:

$$v = \langle id, InstructionType, GeneralPrimitive \rangle$$

$$GeneralPrimitive = \langle f, P, \Phi \rangle$$

where f, P are the robot primitive and its parameters. $\phi_i \in \Phi$ is a distribution over groundings of the parameter $p_i \in P$.

These distributions are learned during task generalization and are used to propose initial parameters during task autocompletion. A GIG can be instantiated as an IG by grounding all of the uninstantiated parameters. This process consists of replacing any unspecified p_i with an actual value.

Our approach generates a library of GIGs from a library of IGs, as shown in Algorithm 4. The general problem of finding labeled subgraph isomorphisms is NP-Hard. However our problem can be reformulated into the problem of finding common labeled subtrees in a forest of trees. To this end, we create a tree representation of each IG. As the first step, we define a mapping from IGs to Trees (T):

$$toTree : IG \rightarrow T$$

and its corresponding inverse:

$$toIG : T \rightarrow IG$$

The function *toTree* computes a labeled spanning tree of an input Instruction Graph (line 3). Specifically, *toTree* creates a spanning tree rooted at the initial vertex of the input IG, by performing a depth first search and by removing back edges in a deterministic manner. This ensures that instances of the same GIG map to the same spanning tree.

Each node in the tree is labeled with the *InstructionType* and function f of the corresponding node in the IG. In this label, we do not include the parameters because we eventually want to generalize over them.

Next, we use a labeled frequent tree mining algorithm to find frequently occurring tree patterns (line 5). A frequently occurring tree pattern is a subtree that appears more than a threshold σ , called the *support*. A tree-mining algorithm *ftm* takes as input a set of trees and the support. As output, it provides a mapping from each tree pattern to the subset of trees that contain it. Then, since each tree pattern is associated to a set of trees and each tree corresponds to a specific IG, we can create a mapping directly from tree patterns to IGs (line 7). We denote this mapping as *IGP*.

A tree mining algorithm will return many tree patterns. In particular, for any tree pattern, any

Algorithm 4 Task Generalization

```
1: procedure GENERALIZE TASKS( $IGs, \sigma, L$ )
2:   // IG library is converted to trees
3:    $IGTrees \leftarrow \{toTree(g) \mid g \in IGs\}$ 
4:   // Tree patterns are found by a tree mining algorithm
5:    $tp \leftarrow ftm(IGTrees, \sigma)$ 
6:   // Mapping from tree patterns to IGs is created
7:    $igp \leftarrow \{\langle p, toIG(T) \rangle \mid \langle p, T \rangle \in tp\}$ 
8:   // Filters remove unwanted tree patterns
9:    $igp \leftarrow filter\_not\_exec(igp)$ 
10:   $igp \leftarrow filter\_by\_length(igp, L)$ 
11:  // Tree patterns of full tasks are reintroduced
12:   $igp \leftarrow add\_full\_igs(IGs, igp)$ 
13:  // Vertices and edges of the GIGs are constructed
14:   $gigs \leftarrow create\_ugigs(IGs, igp)$ 
15:  // Parameters and distributions are computed
16:   $gigs \leftarrow parametrize(IGs, igp, gigs)$ 
17:  return  $gigs$ 
18: end procedure
```

subtree of it will be returned. This is because each subtree will have a support at least as large as its parent. Rather than keeping all these patterns, we focus on storing those that are the most applicable. There are many possible ways to filter the patterns. We propose several heuristic filters that select patterns based on their *executability*, *frequency*, and *usefulness*.

- **Executable** patterns are those that the robot can run.
- **Frequent** patterns are statistically likely to appear in the future.
- **Useful** patterns reduce many interactions when correctly proposed.

Each filter is formally defined as a function:

$$filter : IGP \rightarrow IGP$$

We first filter patterns that cannot be executed by the robot. In particular, we remove patterns with incomplete conditionals and loops (line 9).

Then, there is a tradeoff between highly frequent patterns and highly useful patterns. Patterns that occur with a large frequency are typically smaller, so they provide less utility during task autocompletion. Larger patterns provide a lot of utility, but they are usually very specific and occur rarely.

Less frequent patterns are already filtered out by the tree-mining algorithm when we provide

it a minimum support σ . To optimize for larger patterns that save more steps during autocompletion, we also remove patterns that are shorter than a threshold length L (line 10). This ensures that we do not keep any pattern that is too small to justify a recommendation to the user.

Finally, since we are dealing with autocompletion for robots, one desirable feature is to be able to propose entire tasks. Even if a full task has a low support, or is below the threshold length, there is value in being able to propose it if a reparameterized copy is being provided to the robot. Consequently, we reintroduce the tree patterns corresponding to full IGs (line 12).

Even with these filters, we still keep some tree patterns that are complete subtrees of another pattern. In practice, many of these patterns provide useful task autocompletion suggestions. However, in memory-limited systems, we suggest also filtering them.

Finally, the algorithm processes the filtered set of tree patterns to create GIGs by creating vertices and edges from the tree pattern and then parameterizing the vertices.

First, to create the GIG’s vertices and edges, we copy the subgraph corresponding to the tree pattern from any of the IGs containing the pattern (line 14). This gives us a completely unparameterized GIG (*uGIG*), with no parameter distributions. Next, we determine which parameters are grounded in the GIG, and which are left ungrounded. A parameter is instantiated if it occurs with the same value, with a frequency above a given threshold, in all corresponding IGs. Otherwise, the parameter is left ungrounded with an empirical distribution.

This process is repeated for every subtree pattern found that has not been removed by our heuristic filters, creating a library of GIGs (line 16). In the case where this algorithm is being run incrementally, this library is unioned with the previous library.

On their lefts, Figures 4.2 and 4.3 show example Instruction Graphs for a task that picks up an object, and drops it at one of two locations. On their rights, Figures 4.2 and 4.3 depict their corresponding spanning trees. Finally, Figure 4.4 shows the general task that is extracted. In this GIG, the parameters in nodes 3 and 4 are kept instantiated, since they were shared by the two original Instruction Graphs. The others parameters instead are left ungrounded. These parameters have a type, *id*, and distribution over the landmark ids $\{1, 3\}$, which were extracted from the Instruction Graphs in Figures 4.2 and 4.3.

We can now provide task autocompletion suggestions based on the library of GIGs.

4.2.2 Task Autocompletion

We now consider an agent that is provided a task incrementally through a series of interactions. Each interaction consists of adding a vertex to the graph or modifying an existing vertex. At any step of this process, the agent knows a *partial task*. After each interaction, this partial task is compared against the library of GIGs to measure task similarity and perform autocompletion.

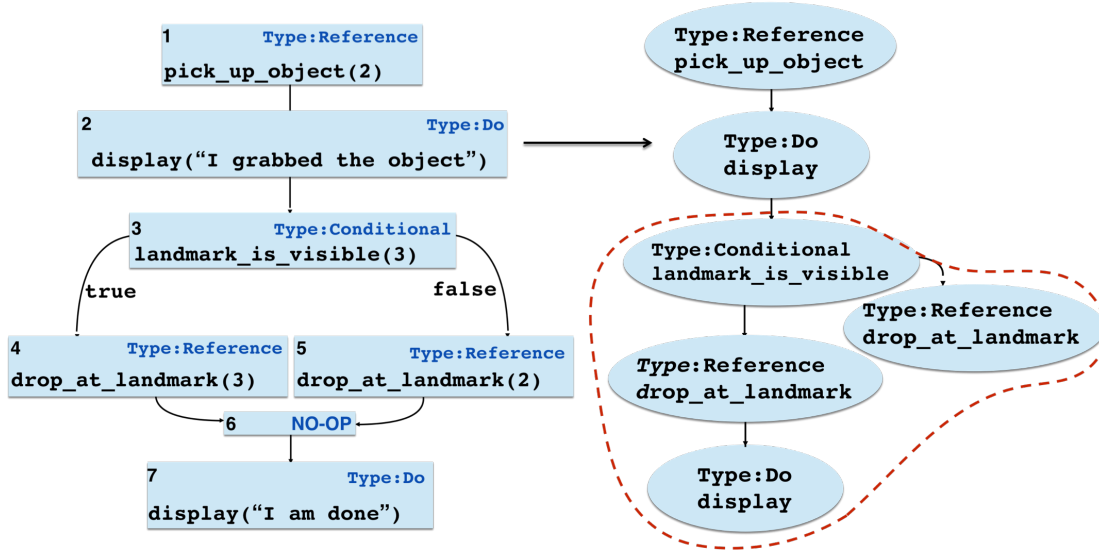


Figure 4.2: Example of an Instruction Graphs (left) converted into its corresponding spanning tree (right). The tree pattern shared with the Instruction graph in Figure 4.3 is circled in red.

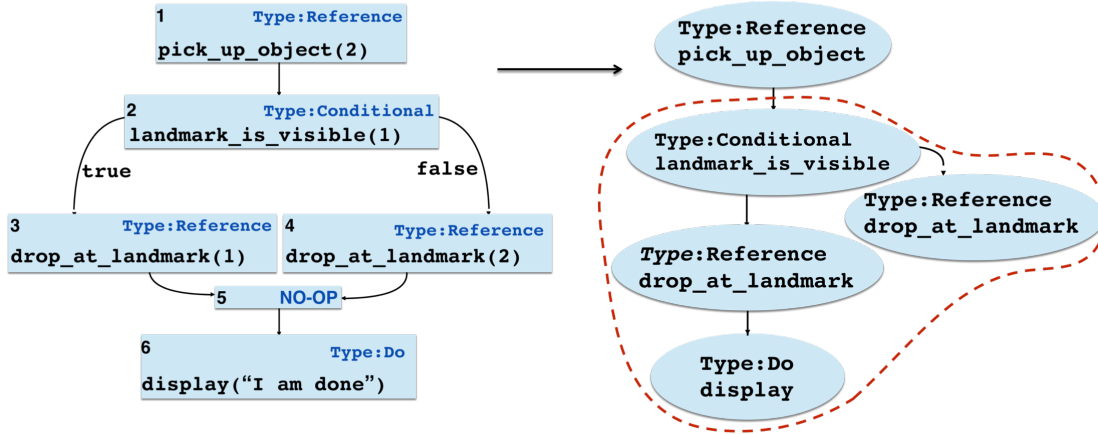


Figure 4.3: Example of an Instruction Graphs (left) converted into its corresponding spanning tree (right). The tree pattern shared with the Instruction graph in Figure 4.3 is circled in red.

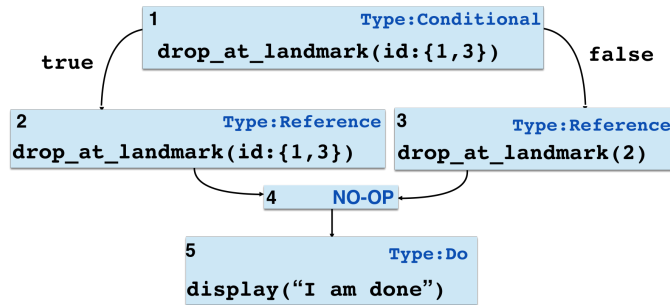


Figure 4.4: Example GIG that is extracted from the graphs in Figures 4.2 and 4.3. The parameters in nodes 3 and 4 are instantiated, since they were shared by the two original IGs. Instead, the parameters in nodes 1 and 2 are left ungrounded.

The algorithm performs this comparison by checking if the end of the partial task being provided is similar to a GIG (Algorithm 5). Specifically, we keep a set of candidate proposals, denoted *props*, that match the final part of the partial task. When the partial task changes (line 2), we first update this set to remove any elements that no longer match the task being taught (line 3). Then, we add new elements for every GIG that starts with the new vertex (line 4). When a threshold percentage τ of one or more GIGs in this set matches the current partial task, the robot proposes the largest GIG and breaks ties randomly (lines 6 and 7).

Algorithm 5 Task Autocompletion

```

1: procedure AUTOCOMPLETE(GIGs, ig, props,  $\tau$ )
2:   if hasChanged(ig) then
3:     props  $\leftarrow$  deleteNotMatching(ig, props)
4:     props  $\leftarrow$  addNewMatching(ig, props, GIGs)
5:     (best, similarity)  $\leftarrow$  bestMatch(ig, props)
6:     if similarity  $\geq \tau$  then
7:       propose(best)
8:     end if
9:   end if
10: end procedure

```

When a specific proposal is found, the robot displays a representation of the GIG and asks for permission to demonstrate the task. Having previously filtered all the incomplete GIGs, all the proposals can in fact be executed. When granted permission, the agent demonstrates an instance of the GIG, noting when a parameter is ungrounded.

At the end of the demonstration, the agent asks if the partial task should be autocompleted with the demonstrated task. If so, the agent asks for specific values for all of the ungrounded parameters. At this stage, the agent suggests initial values for each ungrounded parameter p_i by sampling from its corresponding distribution ϕ_i .

After all of the parameters are specified, the nodes matching the general task in the partial task are replaced with one *Reference* node. When visited, this node executes the referenced GIG, instantiated with the provided parameters. With this substitution, the length of the task is reduced.

Figure 4.5 shows a sample task acquisition for a Baxter manipulator interacting with a user. After the first command, the robot finds at least one general task starting with *display_message*. However, none of the GIGs recognized surpass the similarity-threshold τ . When the third instruction is given to the agent, this threshold is surpassed, and the autocompletion procedure is started. First, the robot asks permission to perform a demonstration of the general task. After completing the demonstration, the robot asks if the autocompletion is correct. If so, it also asks

for ungrounded parameters to be specified and suggests values using each parameter’s distribution.

Example Interaction

```
U: Open Gripper
R: I will open my gripper.
R: What should I do next?
U: Display message "Hello".
R: Ok, what should I do next?
U: Set your left arm to 80 degrees.
R: I think you are teaching me
    something similar to: GIG_l4.
R: Can I demonstrate it to you?
U: Yes.
R: First I will display the message "Hello".
R: Then I will set my left arm to 80 degrees.
R: Now I will set my right arm to 90 degrees (open parameter).
R: This is my full suggestion.
R: Would you like to use it?
U: Yes.
[User specifies open parameters]
[User can rename the GIG]
```

Figure 4.5: Sample autocompletion interaction during task acquisition.

4.2.3 Experiments

In order to demonstrate the value of our approach in reducing interaction steps, we define two sets of tasks. Intuitively, the first set of tasks represents a robot that is still acquiring completely new capabilities. Instead, the second set of tasks represents a robot that is acquiring instances of tasks that it already knows. More formally, in the first set, S_d no tasks are repeated. They share only a small fraction of similar components that can be generalized. To show that generalization takes place, we use a second set, S_r consisting of two repetitions of each of elements in S_d with different parameterizations. We see that the algorithm recognizes and autocompletes the second instance of each task.

An additional benefit of this approach is that we can keep one common library for all of our robots. Since the robots have different primitives, their tasks are automatically generalized apart. One concern is that this library could be very large. We have accumulated libraries of up to 10,000 tasks, by creating parametric variations of a smaller core of tasks. Even on libraries this large our approach runs in under 4 seconds.

In the rest of this section, we show in detail experiments run on Baxter.

Experiments with Baxter

The frequent subtree mining algorithm we employ is an open source version of SLEUTH ¹. The tasks that Baxter can perform range from waving to making semaphore signs to pointing at landmarks. Many of Baxter’s tasks involve picking up an object and moving it to another location. For instance, Figure 4.4 showed a task where Baxter searches for a landmark to see if a location is unobstructed to drop a block. Without task generalization and autocompletion, a new task must be provided for each starting location and ending location in Baxter’s workspace. With generalization and autocompletion, these locations become ungrounded parameters that can be instantiated with any value.

For this experiment, 15 tasks were taught by two users familiar with robots but not the teaching framework. S_d has 15 distinct tasks and S_r has 30 tasks. The average length of a task in both sets is 9.33 nodes.

Experimental Results

As we accumulate the library incrementally, the order in which tasks are provided affects the generalization. To account for this, we ran 1000 trials where we picked a random ordering to our sets and had a program incrementally provide them to Baxter. The GIG library was updated after each task was provided. At the end of every trial, we measured the number of steps saved using autocompletion compared to providing every step of the task. This measurement includes steps added due to incorrect autocompletion suggestions. For this particular experiment, the support was fixed to 2, the minimum GIG length to 2, and task autocompletes were suggested if $\tau = 30\%$ of a GIG matched the partial task.

We compare our results to the number of steps saved by a near-optimal autocompletion algorithm. For any set of tasks that take n total steps to all be provided, and are only proposed once τ percent of a GIG matches a task, we have:

$$OPT \leq (1 - \tau) \cdot n$$

This corresponds to perfectly generalizing every task after τ percent of it has been acquired. We note that this is near-optimal because the robot could do slightly better by proposing additional smaller task autocompletions for the initial τ percent of each task. For S_d and S_r we have:

$$OPT_d \leq 92$$

¹www.cs.rpi.edu/~zaki/software/

$$OPT_r \leq 184$$

Table 4.1 reports the result of the experiment. Specifically, in this table we report the following measures:

- **Maximum Steps Saved (%)**: maximum percentage of steps saved over all permutations using autocompletion, in comparison to the theoretical upper bound.
- **Average Steps Saved (%)**: average percentage of steps saved over 1000 permutations using autocompletion, in comparison to the theoretical upper bound.
- **Average Partially Autocompleted**: average number of tasks that were *partially* autocompleted with a GIG.
- **Average Completely Autocompleted**: average number of tasks that were *completely* autocompleted with a GIG.

Table 4.1: Results obtained for the two sets taught to the Baxter robot.

	1st Set (S_d)	2nd Set (S_r)
Max. Steps Saved	70.65%	100%
Avg. Steps Saved	$33.44 \pm 14\%$	$81.92 \pm 7.05\%$
Part. Autocompleted	4.72 ± 1.56	4.70 ± 1.58
Compl. Autocompleted	0 ± 0	15 ± 0

As expected, S_r benefits the most from the autocompletion method. Specifically, the average percentage of steps saved compared to OPT_r is approximately 82%. For S_d , the average compared to OPT_d is 33%. In the former case, the robot is able to leverage the knowledge of the similar tasks it already knows. Indeed, our approach is near-ideal when provided tasks from S_r in the optimal ordering. This fact is additionally underlined by the number of tasks in which the robot suggested any correct GIG. In particular, on average the robot proposed a correct autocompletion suggestions for 65% of graphs in the second set, and 30% in the first set.

We also notice that the additional 15 IGs added to the second set are all completely autocompleted from their other similar instance. Furthermore, this happens with a statistically insignificant change to the number of partial autocompletions of the other IGs.

Finally, we report that the size of the GIG library for the first set was 21 and that the size of the GIG library for the second set was 45. This shows that the heuristic filters we proposed achieves a good balance between saving steps and library size.

4.3 Related Work

Of all the task teaching frameworks discussed in Chapter 2, one other considers the problem of correcting previously taught tasks [13]. The notable difference is that they modify a task by directly referring to a part of a task and replacing it with another one. In our approach, task correction and modification occurs during a test execution

This generalization and autocompletion work is similar to research in accumulating and reusing tasks, which has been widely studied in multiple contexts. For instance, in case-based planning, a planner stores past solution and uses them to solve a new goal more efficiently [16, 20, 45]. In chunking, the solution to a goal or subgoal is saved as a control rule that can be applied in future situations [1, 6, 26, 27, 31]. Similar to our approach, in macro learning, solutions to specific problems are generalized into macros that can be used to solve general tasks [12].

All of these methods represent tasks as plans, where the planners use past experience to significantly reduce the effort of solving the problem. They work by learning sequences of actions that will solve goals or subgoals. In our work, we represent tasks as Instruction Graphs, which might be provided by a user. Thus, the robot may not know the goal of the task provider a priori. Instead of using domain dependencies, generalization and autocompletion are based on task structure.

Other researchers have focused on the specific problem of generalizing over tasks provided by a user, where the goal might not be known in advance. For instance, tasks have been generalized from multiple examples, where each example corresponds to exactly one past task, and the user specifies the generalized class [34]. This is different from our approach, which takes an input library of many tasks belonging to different classes. Furthermore, we also search for subtasks common to multiple generalized classes.

To represent conditionals and loops, many task representations are graph-based. Therefore, generalization from examples requires finding common subgraphs between different tasks. The problem of finding labeled subgraph isomorphisms is NP-Hard [21]. However, the problem becomes tractable on trees. To solve it, multiple algorithms have been proposed for mining common frequent subtrees from a set of trees [19]. One, Treeminer, uses equivalence class based extensions to effectively discover frequently embedded subtrees [49]. Instead, GASTON divides the frequent subgraph mining process into path mining, then subtree mining, and finally subgraph mining [35]. We use SLEUTH, an open-source frequent subtree miner, able to efficiently mine frequent, unordered or ordered, embedded or induced subtrees in a library of labeled trees [48]. SLEUTH uses scope-lists to compute the support of the subtrees, while adopting a class-based extension mechanism for candidate generation. Our mined tree patterns are further filtered out

to capture executable and parameterized task graphs.

4.4 Summary

In this chapter, we considered task management for autonomous robots with libraries of tasks. First, we presented an approach for correcting existing tasks through test executions. Then, we considered the problem of provided an *additional* task to a robot efficiently. To this end, we contributed an approach to generalize graph-based tasks, and an algorithm that enables the autocompletion of partially specified tasks.

Our generalization and autocompletion algorithms have been successfully deployed on multiple robots, acquiring large task libraries. Our experiments report in-detail the effectiveness of our contributions on a Baxter manipulator robot for two sets of tasks. With both sets, we found a significant reduction in the number of steps needed for Baxter to acquire the tasks.

In terms of future work, there may be other applicable filters for deciding which tree patterns should be converted to GIGs. Furthermore, structure-based generalization is just one way for a robot to express its capabilities. Future research may look at domain-specific forms of task generalization.

Chapter 5

Conclusion

This thesis set out to develop an approach for instructing single or multiple robots through language. Then, we looked at techniques for providing tasks more efficiently. This final chapter summarizes the presented contributions and discusses some directions for future work.

5.1 Contributions

The key contributions of this thesis are:

- **Instruction Graphs**

We introduced Instruction Graphs as a task representation that can be incrementally taught to a robot through language. They can also be corrected and modified during test executions. Most importantly, all of this can be done without requiring the user to write code.

- **Instructing Multiple Robots for Sparse Coordination**

We extended Instruction Graphs by allowing robot primitives to have STRIPS preconditions and effects. This enables tasks that require multiple robots to sparsely coordinate.

- **Task Generalization and Autocompletion**

We considered persistent robots with large task libraries, possibly accumulated from multiple users. We presented algorithms for performing generalization to learn commonly occurring structurally similar pieces of tasks. Then, we presented an algorithm that, given a partial task, proposes the most likely autocompletion from the generalized tasks.

5.2 Directions For Future Work

There are many ideas for future work mentioned in the individual chapters. Here I list some of the most interesting.

With respect to teaching Instruction Graphs, one direction of future work is to support less restrictive language from the user. There are many statistical natural language grounding techniques that could be applicable here. Along another direction, there are many ways to correct an existing task that do not require full test executions, such as directly referencing a piece of a task or using a GUI. Furthermore, corrections should allow the user to add or remove sequences of actions and control statements. Lastly, our user studies showed us that even roboticists had difficulty understanding the output and limitations of sensors on a robot. It is a challenging problem to enable a robot to explain its sensing capabilities and limitations to the user.

In terms of multi-robot coordination, our approach uses a deterministic STRIPS representation of action preconditions and effects. One natural extension is to consider actions with non-deterministic affects. In such cases, a robot may need to explicitly check its state using sensing actions before responding to a query.

Lastly, teaching a robot is not typically thought of as an interactive process. Task generalization and autocompletion is just one method to allow a robot to assist the user that is teaching it. One of the most interesting future directions for research are other ways to allow the robot to assist its instructor.

Appendix A

Robot Primitive Language Descriptions

This appendix shows the language descriptions for the primitives used on CoBot and Baxter. When requested, a robot can describe its robot primitives to the user using these language descriptions. The language primitive descriptions consistently use the variable L to refer to the integer ID of an Augmented Reality Landmark, m to refer to a string of text, A to refer to an arm of Baxter, and x, y, z to refer to displacements.

A.1 CoBot Language Descriptions and Examples

Move($x, y, \theta, v_x, v_y, v_\theta$):

- Language Description: “Moves the robot forwards x m, laterally y m, and angularly θ rads, with corresponding maximum velocities v_x m/s, v_y m/s, and v_θ rads/s.”
- Example: *Move*(0.5, 1.5, 0, 0.25, 0.25, 0)

MoveTo(x, L):

- Language Description: “Moves the robot x m away from Landmark L .”
- Example: *MoveTo*(0.5, 5)

Say(m):

- Language Description: “Speaks the text message m .”
- Example: *Say*(“Hello, I am CoBot!”)

IsLandmarkVisible(L):

- Language Description: “Sensing primitive that is true if Landmark L is visible.”
- Example: *IsLandmarkVisible*(2)

A.2 Baxter Language Descriptions and Examples

Move(A, x, y, z):

- Language Description: “Moves arm A to the location (x, y, z) in 3D space, where x is the forward displacement in meters, y is the lateral displacement in meters, and z is the vertical displacement in meters, with respect to the base of the robot.”
- Example: *Move(Left, 0.5, 0.2, 0.5)*

MoveTo(A, L):

- Language Description: “Moves arm A to the location of Landmark L .”
- Example: *MoveTo(Left, L):*

SetGripper(x):

- Language Description: “Opens gripper x cm.”
- Example: *SetGripper(0)*

Display(m):

- Language Description: “Displays a text message m on the robot’s screen.”
- Example: *Display(“Hello, I am Baxter!”)*

IsLandmarkVisible(L):

- Language Description: “Sensing primitive that is true if Landmark L is visible.”
- Example: *IsLandmarkVisible(3)*

A.2.1 Landmarks

The landmarks we use are augmented reality codes (AR-codes). They are usually printed to a piece of paper and placed in the robots’ environments. Each AR-code has an associated integer ID that can be detected by a standard camera. Additionally, the camera can determine the AR-code’s position and orientation with respect to the camera. Figure A.1 shows an example of several augmented reality codes we have used as landmarks.

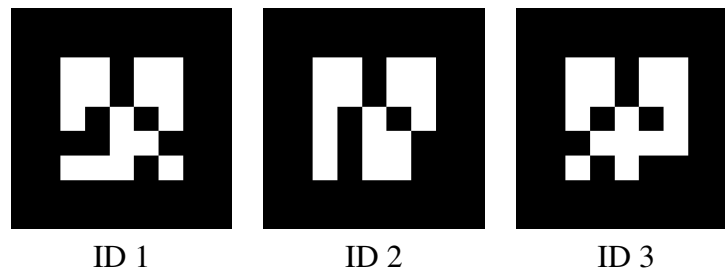


Figure A.1: AR-codes with IDs 1 (left), 2 (middle) and 3 (right).

Appendix B

Planning with Instruction Graphs

In most of this work, Instruction Graphs are presented as a tool that allows users to provide tasks to a robot. However, we have also started preliminary work in generating Instruction Graph plans, given a goal. A major benefit of this is that users and planners can both instruct tasks to a robot using the same representation.

In Chapter 3, we introduced STRIPS preconditions and effects for robot primitives. These preconditions and effects allow planners to solve user-provided goals by creating Instruction Graph plans. Initially, the planner’s plans over the robot’s actuation and sensing primitives. However, as the robot learns new Instruction Graphs, they become additional actions for the planner to consider.

To use newly acquired Instruction Graphs with a planner, they must have preconditions and effects. Since robot primitives are all defined in advance, it was possible for the robot-deployer to include their preconditions and effects in their definitions. However, Instruction Graphs are not known a priori, so this information must be determined after the robot is deployed.

In related work, other researchers have addressed similar problems with Explanation Based Learning [32]. Their approach treats tasks as chunks, and learns chunking rules for a goal, which map robot-state to applicable tasks. This learning process occurs by analyzing a task to understand how each step achieves the goal. We have investigated a similar approach for analyzing Instruction Graphs to determine their preconditions and effects.

B.1 Inferring Instruction Graph Preconditions and Effects

In general, an Instruction Graphs with sensing conditionals and loops may have preconditions and effects that are unknown until execution time. For instance, an Instruction Graph may perform two very different actions based on a sensing conditional that can only be evaluated at run-

time. We leave for future work the problem of analyzing Instruction Graphs with conditionals and loops to find if they have preconditions and effects that can be determined before execution. In this section, we present an approach for inferring the preconditions and effects of sequential Instruction Graphs by examining their robot primitives. When an Instruction Graph is composed sequentially, it has a deterministic set of preconditions and effects.

Consider an Instruction Graph G of sequential primitives and references to tasks, $T_1, \dots, T_n$. The effects E of such an Instruction Graph are:

$$E = \bigcup_{i=1}^n \text{Effect}(T_i)$$

Importantly, there is a conflict if a step T_i adds a state literal removed in step T_j . In such cases, the literal is an effect if $i > j$ and is not an effect otherwise. This is because the effect of the task should correspond to its final effect on a state literal.

Next, the task preconditions P of the Instruction Graph consist of the preconditions of T_1 and any preconditions of step T_i that are not fulfilled by the effects of a preceding action ($T_1, \dots, T_{i-1}$).

$$P = \bigcup_{i=1}^n \left(\text{Prec}(T_i) \setminus \bigcup_{j=1}^{i-1} \text{Effect}(T_j) \right)$$

Importantly, a conflict occurs if step T_i requires a STRIPS literal to be true and step T_j requires that same literal to be false. In such cases, the precondition requires the literal to be true if $i < j$, and false otherwise. This is because the preconditions of a task should correspond to their earliest requirements on a state literal.

B.2 Planning For Task Proposals in Blocks World

Now, given a set of goals, the robot can actively plan for solutions represented as Instruction Graphs. Then, these solutions can be proposed to a user, similar to the task autocompletion approach in Chapter 4. There are many valid STRIPS planners such as simple search algorithms, FF [17], and GraphPlan [5]. In this section, we present an example of Baxter using a planner to solve Blocks World problems. In Blocks World, the robot can pick up blocks, place them on a table, and stack them atop one another. Specifically, we have a table and four blocks colored blue, green, orange, and yellow. Figure B.1 shows Baxter ready to begin a task.

For this example, Baxter has been taught Instruction Graphs to pick up each colored block, place each block atop another, or to place a block on the table. These correspond to Instruction

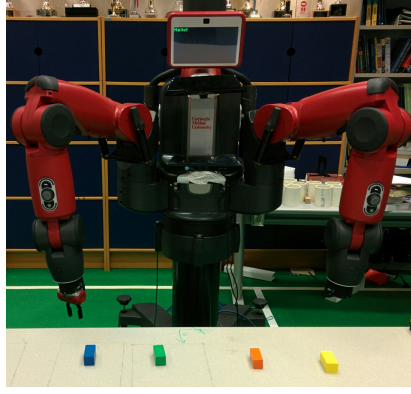


Figure B.1: Baxter setup to begin a Blocks World task.

Graphs with the following preconditions and effects:

- **Pickup Block [color]:** Picks up the [color] block. Has the precondition `hand_empty` and `clear_[color]`, and the effects `holding_[color]_block`, `¬clear_[color]`, and `¬hand_empty`.
- **Place [color] Block on [color2]:** Places the [color] block on top of [color2]. Has the preconditions `holding_[color]_block` and `clear_[color2]`, and the effects `¬holding_[color]_block`, `hand_empty`, `on_[color]_[color2]`, `¬clear_[color2]` and `clear_[color]`.
- **Place [color] Block on table:** Places the [color] block on the table. Has the preconditions `holding_[color]_block` and the effects `¬holding_[color]_block`, `hand_empty`, `on_[color]_[table]`, and `clear_[color]`.

First, the user provides a goal such as $\langle \text{on_green_blue}, \text{on_orange_yellow} \rangle$. If the planner finds a solution that achieves all of the user-provided goals, it is proposed to the user. A proposal consists of the robot telling the user the list of tasks it would like to use to solve this problem. If the planner finds a plan that solves some of the user-provided goals, it is also proposed if more than τ percent are solved. In this way, these task proposals are similar to the ones presented in Chapter 4 that may propose a piece of a task. Figure B.2 shows one example solution to the goal $\langle \text{on_green_blue}, \text{on_orange_yellow} \rangle$.

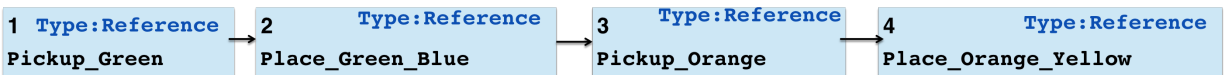


Figure B.2: IG found by a planner for having the green block on the blue, and the orange block on the yellow.

Next, Baxter will ask to demonstrate this solution to the user. After the demonstration, the

user can save the task, edit it, or reject it and begin teaching something else. These task proposals occur in the same manner as those in Chapter 4.

B.3 Summary

In this appendix, we presented preliminary work on using Instruction Graphs with planners. One major benefit of Instruction Graphs is that they can be used by both planners and people to instruct a robot. Additionally, we demonstrated how planners could be used to suggest task proposals to a user, given a goal.

Currently, our work was limited to creating sequential plans without sensing conditionals or loops. However, there are planning techniques for generating Conditional plans, and STRIPS representations that support conditional statements [47].

Bibliography

- [1] John R Anderson, Dan Bothell, Christian Lebiere, and Michael Matessa. An integrated theory of list memory. *Journal of Memory and Language*, 1998. 4.3
- [2] Brenna D Argall, Brett Browning, and Manuela Veloso. Learning robot motion control with demonstration and advice-operators. In *International Conference on Intelligent Robots and Systems, IROS*, 2008. 2.4
- [3] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 2009. 2.4
- [4] Joydeep Biswas and Manuela M Veloso. Localization and navigation of the cobots over long-term deployments. *The International Journal of Robotics Research*, 32(14):1679–1694, 2013. 1.1
- [5] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997. B.2
- [6] Daniel Borrajo and Manuela Veloso. Lazy incremental learning of control knowledge for efficiently obtaining quality plans. In *Lazy learning*. 1997. 4.3
- [7] Rehj Cantrell, Kartik Talamadupula, Paul Schermerhorn, J. Benton, Subbarao Kambhampati, and Matthias Scheutz. Tell me when and why to do it!: run-time planner model updates via natural language instruction. In *Proc. of HRI*, 2012. 2.4
- [8] Sonia Chernova and Manuela Veloso. Learning equivalent action choices from demonstration. In *International Conference on Intelligent Robots and Systems, IROS*, 2008. 2.4
- [9] E. Chuangsuwanich, S. Cyphers, J. Glass, and S. Teller. Spoken command of large mobile robots in outdoor environments. In *Proc. of Spoken Language Technology Workshop (SLT)*, 2010. 2.4
- [10] J. Dzifcak, M. Scheutz, C. Baral, and P. Schermerhorn. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *Proc of ICRA*, 2009. 2.4
- [11] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972. 3.1
- [12] Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 1972. 4.3
- [13] Guglielmo Gemignani, Emanuele Bastianelli, and Daniele Nardi. Teaching robots parametrized executable plans through spoken interaction. In *Proceedings of the 2015*

- International Conference on Autonomous Agents and Multiagent Systems*, pages 851–859. International Foundation for Autonomous Agents and Multiagent Systems, 2015. 2.4, 4.3
- [14] Brian P Gerkey and Maja J Mataric. Sold!: Auction methods for multirobot coordination. *Robotics and Automation, IEEE Transactions on*, 18(5):758–768, 2002. 3.3
 - [15] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David E Smith, et al. Pddl-the planning domain definition language. 1998. 3.1
 - [16] Kristian J Hammond. Chef: A model of case-based planning. In *AAAI*, 1986. 4.3
 - [17] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, pages 253–302, 2001. B.2
 - [18] Albert Xin Jiang, Kevin Leyton-Brown, and Navin AR Bhat. Action-graph games. *Games and Economic Behavior*, 71(1):141–173, 2011. 3.3
 - [19] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 2013. 4.3
 - [20] Subbarao Kambhampati. A theory of plan modification. In *AAAI*, 1990. 4.3
 - [21] Benny Kimelfeld and Phokion G Kolaitis. The complexity of mining maximal frequent subgraphs. *ACM Transactions on Database Systems (TODS)*, 39(4):32, 2014. 4.3
 - [22] Jelle R Kok, Pieter Jan’t Hoen, Bram Bakker, and Nikos A Vlassis. Utile coordination: Learning interdependencies among cooperative agents. In *CIG*. Citeseer, 2005. 3.3
 - [23] Thomas Kollar, Stefanie Tellex, Deb Roy, and Nicholas Roy. Toward understanding natural language directions. In *Proc. of HRI*, 2010. 2.4
 - [24] Thomas Kollar, Vittorio Perera, Daniele Nardi, and Manuela Veloso. Learning environmental knowledge from task-based human-robot dialog. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 4304–4309. IEEE, 2013. 2.3.4
 - [25] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *Robotics, IEEE Transactions on*, 25(6):1370–1381, 2009. 2.4
 - [26] John E Laird, Paul S Rosenbloom, and Allen Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine learning*, 1986. 4.3
 - [27] Pat Langley, Kathleen B McKusick, John A Allen, Wayne F Iba, and Kevin Thompson. A design for the icarus architecture. *ACM SIGART Bulletin*, 1991. 4.3
 - [28] Stanislao Lauria, Guido Bugmann, Theodoros Kyriacou, Johan Bos, and Ewan Klein. Personal robot training via natural-language instructions. *IEEE Intelligent Systems*, 16:38–45, 2001. 2.4
 - [29] Cynthia Matuszek, Dieter Fox, and Karl Koscher. Following directions using statistical machine translation. In *Proc of HRI*, 2010. 2.4
 - [30] Francisco S Melo and Manuela Veloso. Decentralized mdps with sparse interactions. *Artificial Intelligence*, 175(11):1757–1789, 2011. 3
 - [31] Steven Minton. Quantitative results concerning the utility of explanation-based learning.

Artificial Intelligence, 1990. 4.3

- [32] Shiwali Mohan and John E Laird. Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In *Proceedings of the Twenty Eighth AAAI conference on Artificial Intelligence*. AAAI Press, 2014. B
- [33] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, 2000. 2.4
- [34] Monica N. Niculescu and Maja J. Mataric. Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proc. of AAMAS*, 2003. 2.4, 4.3
- [35] Siegfried Nijssen and Joost N Kok. The gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*. 4.3
- [36] Lynne E Parker. Alliance: An architecture for fault tolerant multirobot cooperation. *Robotics and Automation, IEEE Transactions on*, 14(2):220–240, 1998. 3.3
- [37] Nathan D Ratliff, J Andrew Bagnell, and Martin A Zinkevich. Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning*, 2006. 2.4
- [38] S. Rosenthal, J. Biswas, and M. Veloso. An effective personal mobile robot agent through symbiotic human-robot interaction. In *Proc. of AAMAS*, 2010. 1.1
- [39] Maayan Roth, Reid Simmons, and Manuela Veloso. Exploiting factored representations for decentralized execution in multiagent teams. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 72. ACM, 2007. 3.3
- [40] Paul Rybski, Jeremy Stolarz, Kevin Yoon, and Manuela Veloso. Using dialog and human observations to dictate tasks to a learning robot assistant. *Journal of Intelligent Service Robots, Special Issue on Multidisciplinary Collaboration for Socially Assistive Robotics*, 1(2):159–167, April 2008. 2.4
- [41] M. Skubic, D. Perzanowski, S. Blisard, A. Schultz, W. Adams, M. Bugajska, and D. Brock. Spatial language for human-robot dialogs. *Trans. Sys. Man Cyber Part C*, 34(2):154–167, May 2004. ISSN 1094-6977. 2.4
- [42] Peter Stone. *Layered learning in multiagent systems: A winning approach to robotic soccer*. MIT Press, 1998. 3.3
- [43] Milind Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, pages 22–28. AAAI Press, 1997. 3.3
- [44] S. Tellex, T. Kollar, S. Dickerson, M.R. Walter, A.G. Banerjee, S. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proc. of AAAI*, 2011. 2.4
- [45] Manuela M Veloso. *Planning and learning by analogical reasoning*. Springer Science & Business Media, 1994. 4.3
- [46] A. Weitzenfeld, A. Ejnoui, and P. Dominey. Human robot interaction: Coaching to play soccer via spoken-language. In *IEEE/RAS Humanoids’10 Workshop on Humanoid Robots Learning from Human Interaction*, 2010. 2.4

- [47] Daniel S Weld, Corin R Anderson, and David E Smith. Extending graphplan to handle uncertainty & sensing actions. In *Aaai/iaai*, pages 897–904, 1998. B.3
- [48] Mohammed J Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 2005. 4.3
- [49] Mohammed J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 2005. 4.3