An Interactive Approach for Situated Task Specification through Verbal Instructions

Çetin Meriçli, Steven D. Klee, Jack Paparian, and Manuela Veloso
cetin@cmu.edu, sdklee@cmu.edu, jpaparia@andrew.cmu.edu, veloso@cmu.edu
Computer Science Department, Carnegie Mellon University, Pittsburgh, USA

ABSTRACT

The ability to specify a task without having to write special software is an important and prominent feature for a mobile service robot deployed in a crowded office environment, working around and interacting with people. In this paper, we contribute an interactive approach for enabling the users to instruct tasks to a mobile service robot through verbal commands. The input is given as typed or spoken instructions, which are then mapped to the available sensing and actuation primitives on the robot. The main contributions of this work are the addition of conditionals on sensory information that the specified actions to be executed in a closed-loop manner, and a correction mode that allows an existing task to be modified or corrected at a later time by providing a replacement action during the test execution. We describe all the components of our approach along with the implementation details and illustrative examples in depth. We also discuss the extensibility of the presented approach, and point out potential future extensions.

Categories and Subject Descriptors

I.2.9 [**Robotics**]

General Terms

Algorithms

Keywords

Robot Task Specification, Human Robot Interaction

1. INTRODUCTION

Interacting with humans and responding to their instructions in the environment through natural language are very important capabilities for service robots. Though the robot is equipped with some task knowledge, it would be very useful—if not necessary—to also have the ability to specify new tasks to the robot easily, and preferably without having to modify the robot control software. We envision such service robots to operate around and interact with untrained people who know what tasks they want the robot to do, but does not necessarily have the technical abilities to express such tasks in terms of robot sensing and action primitives. Therefore, being able to specify new tasks through natural language instructions is a very desired ability for robots interacting with and getting instructed by such untrained people.

There are, however, various challenges in instructing robots using natural language such as robust speech recognition in case of spoken interaction, dealing with the ambiguity in given instructions stemming from the flexibility of natural language, and properly mapping given commands to robot behavior primitives that can be executed by the robot.

We contribute an approach for enabling users to instruct the steps needed to perform a task to a mobile service robot in terms of available sensing and actuation primitives. 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 a graph-based representation, and an execution module that takes the generated behavior representation, translates the generated behaviors into robot primitives, and executes the task. The user can also interactively correct and modify a part of an existing task if desired. Our language supports conditionals and loop structures based on sensing different landmarks in the environment. Main contributions of this paper are:

- A keyword based filtering approach to the natural language input that makes our method robust to different expressions of the same command to some extent as long as the proper keywords are used in correct order.
- A task specification language using the available sensing and actuation capabilities of a service robot that also supports loop structures and conditionals based on sensing.
- A correction feature that allows the instructor to modify parts of an existing task as desired.

In the remainder of the paper, we first give a brief overview of related work in the literature and how our approach differs from the existing body of research. We then describe all the components of the contributed approach thoroughly. After presenting implementation details on our service robot as well as a set of illustrative examples of task instruction and correction, we conclude the paper with a discussion of the approach and remarks on possible future work.

2. RELATED WORK

Natural language based interaction with robots has been examined in various different scenarios ranging from com-

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

manding robotic forklifts to teaching robots how to give a tour. Recently, approaches that leverage probabilistic models trained on a labelled corpus have been proposed to deal with the uncertainty in the unrestricted natural language instructions. In [1], an approach using Spatial Description Clauses (SDC) is proposed for parsing a spoken natural language command and extracting the spatial information. The extracted spatial commands are then grounded to the available actions that can be executed by the robot. The proposed approach is evaluated in a navigation scenario. [2] presents Generalized Grounding Graphs (G^3) for parsing commands given through speech using natural language. A Conditional Random Field (CRF) is trained on a manually labelled corpus and then used to infer the most likely G^3 representation for a given natural language command. They applied their method to robotic navigation and manipulation domains. Similarly, a method that uses a parser that learns through statistical machine translation methods is presented in [3] for enabling a robot to follow navigation instructions. In [4], a speech-based robot operation system is proposed for handling cargo in an outdoor scenario by a robotic forklift. Contrary to the probabilistic approaches discussed above, this approach is not robust against flexible natural language commands. Instead, the proposed system expects commands to be given according to the specified syntax. Another approach for verbal instruction is presented in [5]. Similar to the works discussed above, they also use a parser to convert the given natural language command into an action representation. If an unknown word is encountered in the command, their proposed approach first tries to find a similar word with a known concept using WordNet. If no such words can be found, that part of the command is ignored. Ignoring an unknown and unexpected part of the command bears resemblance to our approach for dealing with ambiguity, but our approach deliberately checks for the known commands instead of trying to parse the entire instruction. An approach for updating plans generated by a planner using natural language instructions given through speech is introduced in [6]. A multi-modal spatial language system that utilizes gestures along with the verbal instruction is presented in [7]. The parser of the system 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 the other 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.

Teaching tasks to a robot through verbal instruction has also been studied in various domains. Rybski *et al.* introduced a method for teaching tasks composed of available action primitives for a service robot using spoken verbal instructions [8]. Their approach does not perform any disambiguation on the received verbal command converted to text via a speech recognition module; therefore, their approach mandates a strict syntax for the commands. As a part of the teaching process, the robot can also learn navigation trajectories by following the demonstrator. The main difference between our approach and this work is that our algorithm allows repetitions (cycles) in the task representation and enables the user to modify and correct an existing task. In [9], a method that translates given natural language instructions into formal logic goal description and action languages is presented. The parsing of instructions is done through the use of predefined associations between the lexical items in the instructions and the corresponding λ-expressions.

A method for instructing the robot how to navigate is presented in [10]. The proposed method maps the received instruction to the defined action primitives. An interesting aspect of this work is that these action primitives are extracted from a corpus collected from several subjects. In [11], a method for learning representations of high level tasks is proposed. Their approach allows learning a task composed of non-repetitive sequences of predefined robot primitives; in addition, it supports revisions of the taught task and generalizations of task representations over multiple demonstrations of the same task. Although mapping the instruction to action primitives in these two studies resembles our approach, our language differs as our primitives are parameterized and our instruction language also contains conditionals. Teaching soccer skills via spoken language is addressed in [12]. In their approach, there is a predetermined set of actions and natural language commands that maps on those actions. An interesting aspect in their proposed framework is that it allows the teacher to query the robot for accessing its internal state. 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. Among the main differences with this work and our approach is the ability of our approach to execute the task step by step to enable verification of the taught task as well as modifying and correcting the taught actions.

3. APPROACH

Our interactive and situated task specification approach takes place in three consecutive operations:

- Processing verbal instruction
- Generating task representation
- Execution and correction

An overview of our approach is given in Figure 1. In this section, we review all components of our approach in detail.

3.1 Instruction Graphs

We represent tasks as a composition of available robot primitives using a special graph structure called an Instruction Graph (IG). An IG consists of a tuple $G = \langle V, E\rangle$ formed from n instructions, where $V = \{v_i \mid \forall\ i \in [0, n]\}$, and v_i corresponds to the i^{th} command given to the robot ranging from 1 to n. The starting node of G is denoted with v_0. E is a set of tuples $\langle v_i, v_j\rangle$ representing a directed edge from v_i to v_j. An edge between two vertices denotes a possible transition from one command to the other. A vertex is a 3-tuple of the form $v{=}\langle ID, ActionType, Action\rangle$. Each vertex is given a unique identification number that also specifies their relative order in execution so that $\forall\ v_i \in V$, $ID{=}\ i$.

The *ActionType* field describes the type of each vertex whereas the *Action* field tells the interpreter which actions and sensing are necessary. There are four defined action types that a vertex can have:

- **Do:** A vertex is designated with the *Do* action type if it performs an action completely in open loop.

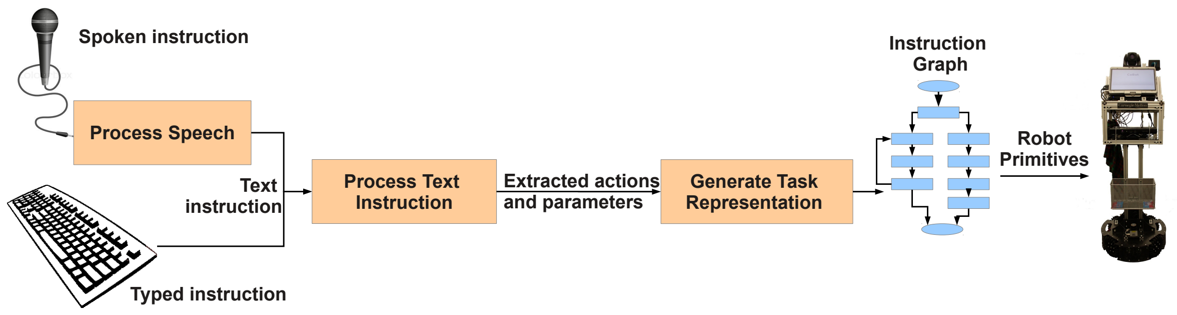


Figure 1: Overview of our task specification approach.

- **DoUntil:** As opposed to the *Do* action type, *DoUntil* refers to an action that has a sensory component, where the execution of the action continues until a specific condition about the sensory input is satisfied.
- **Conditional:** This action type refers to a vertex with more than one outgoing edge, representing a fork in the flow of execution. The *Action* element stores the specific condition to be evaluated to determine which branch of execution to follow at runtime.
- **GoTo:** This action type is used internally to implement loop structures. The *Action* field contains the *ID* of the vertex that the interpreter will jump to. *GoTo* vertices are never created directly by the user.

Our language also has three special commands without specified action types as they are not used in the tasks:

- **Save:** This command saves the current task in memory to a file under a specified name.
- **Load:** This command loads a previously saved task into the memory and appends it to the current node in the execution flow. The details of how the load command works are discussed in the next subsection.
- **Shutdown:** This command terminates the task execution.

3.2 Generating IGs from User Input

When processing the given instruction to create the instruction graph, we search for specific keywords in the user input to determine what type of command has been requested. Once we infer the action type and the action, we make certain assumptions about the input form such as the existence and order of the expected keywords. Also, we filter out any unknown words in the input, therefore, a command can still be successfully parsed even if it is expressed differently as long as the keywords are correctly used and are in order. After extracting the action type, action name, and corresponding parameters (if any), a new vertex and new edge are added to the current task instruction graph G. Instructing a task is an interactive process as the robot asks for confirmation for each inferred action. After each confirmation, a relevant node is created and added to the current IG. A graph node has an ID field, a field that indicates the type of the node, and a parameters field containing the type-specific parameters of the node. Table 1 and Table 2 shows the supported action types and actuation commands, respectively.

Action Type	Keyword(s)
Do	No Keyword
DoUntil	“until”
Conditional	“if”,“while”,“do while”,“end if”, “end while”

Table 1: Action types and corresponding keywords.

As opposed to the *Do* and *DoUntil* action types which are single-step commands, *While* and *DoWhile* commands create loops over a group of nodes that can also contain nested loops. Therefore, generation of such conditional nodes differ from the actuation nodes. For a conditional node, a *Conditional* vertex with two children is created. For an *If* node, the execution continues with the first child vertex if the specified condition evaluates to true, and the execution continues from the second otherwise.

Action 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: Defined actuation commands and corresponding keywords.

For the loops, an additional *GoTo* node is created. The difference between the *While* and *DoWhile* constructs is whether to evaluate the loop condition at the beginning or end of the loop. In the case of a *While* loop, the *Conditional* node is inserted at the beginning. A *GoTo* node is placed at the end of the loop and points back to the *Conditional* node. The body of the loop consists of everything added to the graph between these two nodes. In case of a *DoWhile* loop, the *Conditional* node is placed at the end of the body. When the condition evaluates to true, the *Conditional* node transitions to a *GoTo* node, which jumps to the first node in the body of the loop (Figure 2). This loop implementation guarantees at least one full execution of the loop body.

Table 1 and Table 2 shows the supported action types and actuation commands, respectively.

Algorithm 1 and Algorithm 2 show the algorithms for the creation of actuation and conditional nodes, respectively. The flow of execution through actuation nodes is straightforward; however, closing conditional statements requires knowledge of their starting location. To accomplish this, we utilize a stack to store a record of conditionals in the order they were entered. When the body of a conditional is closed, an element is popped off the stack. This element references the starting location of the conditional to be closed.

Algorithm 1 Creating a Node for Actuation Commands.

```
 1: function addActuation($v_{current}$,$id$,$instruction$)
 2: if checkKeyword($instruction$,"forward") then
 3:     $action \leftarrow$ "$Move$"
 4: else if checkKeyword($instruction$,"turn") then
 5:     $action \leftarrow$ "$Turn$"
 6: else if checkKeyword(instruction,"say") then
 7:     $action \leftarrow$ "$Say$"
 8: end if
 9: if checkKeyword($instruction$,"until") then
10:     $actionType \leftarrow$ "$DoUntil$"
11: else
12:     $actionType \leftarrow$ "$Do$"
13: end if
14: $params \leftarrow parseInformation(actionType, action)$
15: $id \leftarrow id + 1$
16: $v = createNode(id, actionType, action, params)$
17: $v_{current}.children[0] \leftarrow v$
18: $v_{current} \leftarrow v$
```

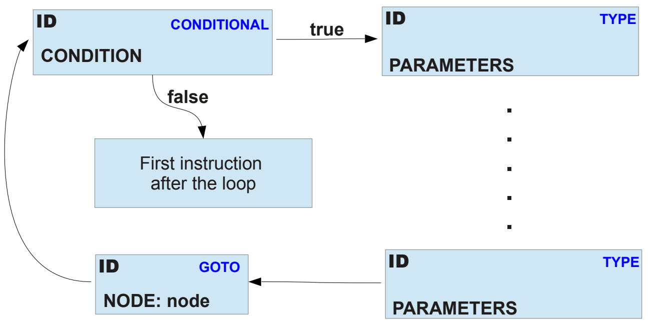


Figure 2: An illustration showing how loops are implemented as a combination of conditionals and *GoTo* nodes.

3.3 Saving and Loading Tasks

Once the instruction is completed, the resulting instruction graph in the memory can be saved to a file for future use by using the *Save* command along with a file name to save the task to.

A previously saved task can be loaded into memory with the *Load* command. The *Load* command works as follows. Given that the user is creating an instruction graph $G = (V_g, E_g)$ with n inputs, we denote the most recently created node as v_{g_n}. The *Load* command loads another instruction graph $H = (V_h, E_h)$ from the specified file. With the newly loaded task at hand, a new instruction graph is formed as $(V_h \cup V_g, E_h \cup E_g \cup \{(v, v_{h_0})\})$. In other words, the resulting graph is the union of G and H with one additional edge connected the most recently created vertex of G to the source-vertex of H. Leveraging this capability, a library of subtasks can be created and used to compose new tasks.

Algorithm 2 Creating Branches and Cycles in the Flow of Execution

```
 1: function beginConditional($stack$,$v_{current}$,$id$,$condition$)
 2: $actionType \leftarrow checkActionType(condition)$
 3: $v \leftarrow createNode(id, actionType, condition)$
 4: $stack.push(v)$
 5: $v_{current}.child[0] = v$
 6: $v_{current} = v$
 7: $id = id + 1$
 8:
 9: function endConditional($stack$,$v_{current}$,$id$)
10: $v_{conditional} \leftarrow stack.pop()$
11: $v_{conditional}.child[1] \leftarrow v_{current}$
12: if $v_{conditional}.actionType ==$ "$While$" then
13:     $v \leftarrow createNode(id,$ "$GoTo$"$, v_{conditional}.ID)$
14:     $v_{current}.child[0] \leftarrow v$
15:     $v_{current} \leftarrow v$
16:     $id \leftarrow id + 1$
17: end if
```

3.4 Executing Tasks

Execution of a task in the form of an instruction graph is a traversal operation that starts from the first node (v_0), and follows defined transitions. The *Do* and *DoUntil* nodes have only one directed edge outward. This edge is followed once the action is performed, and the execution continues with the next node. *Conditionals* have two directed outward edges. Their *Action* field is a conditional statement that evaluates to true or false at runtime. If the condition evaluates to true, the first child of the node is set as the next node to be executed. If the condition evaluates to false, the second child of the vertex is chosen. The algorithm for executing an instruction graph is given in Algorithm 3.

We have three main robot primitives that the given instructions are mapped to. The *Move* primitive is used to execute motion. The supported motion types of moving forward and turning are specified by the tuple

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

where Δ_x, Δ_y represent the forward and lateral displacement, respectively, and Δ_Θ represents the amount of rotation. v_t and v_r represent the maximum translational and rotational velocities respectively. All translational motion is specified in meters, and all rotational motion in radians. The *Say* primitive allows the robot to speak a given text message. Finally, the *Sensing* primitive makes use of the available sensory information on the robot.

3.5 Modifying and Correcting Tasks

A major feature of our approach is the ability to let the instructor correct parts of the task as desired. This can vary from editing the parameters of an action to replacing an action with a new one. 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)

Algorithm 3 Executing a task.

```
1: $G \leftarrow loadTask()$
2: $v_{current} = G.vertices[0]$
3: while $v_{current} \neq \emptyset$ do
4:   if $v_{current}.actionType == \text{“}Do''$ then
5:     $executeAction(v_{current}.action)$
6:     $v_{current} \leftarrow v_{current}.children[0]$
7:   else if $v_{current}.actionType == \text{“}DoUntil''$ then
8:     while $v_{current}.senseCondition$ is not $true$ do
9:       $executeAction(v_{current}.action)$
10:      $v_{current} \leftarrow v_{current}.children[0]$
11:    end while
12:  else if $v_{current}.actionType == \text{“}GoTo''$ then
13:    $v_{current} \leftarrow v_{current}.children[0]$
14:  else if $v_{current}.actionType == \text{“}Conditional''$ then
15:    if $evaluateConditional(v_{current}.action)$ then
16:      $v_{current} \leftarrow v_{current}.children[0]$
17:    else
18:      $v_{current} \leftarrow v_{current}.children[1]$
19:    end if
20:  end if
21: end while
```

The first example is likely to occur when a parameter value is not as accurate as predicted. This often happens due to miscalculations, unexpected changes, or faulty calibration of the robot. A successful framework must be flexible enough to adapt to inconsistencies and uncertainty in its environment. It is also necessary for a framework to easily integrate new sensory information or deal with a particular sensor becoming 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 code 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 code. For example, consider a task created to search for books in a library. This task has two unique components: searching through an area and looking for a specified object type. By modifying this object type, the same task can also be used to search for other objects.

In our approach, the task correction and modification occurs during the task 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 is substituted by the new instruction.

4. IMPLEMENTATION DETAILS

In this section, we describe the actual implementation on our mobile service robot in detail. We implemented and tested our approach on our CoBot mobile service robot [13]. The CoBot service robot has an omnidirectional mobile base, and is equipped with a variety of sensors including a camera, a laser range finder, microphones, and Microsoft Kinect sensors. The users interact with the robot using its touch screen interface and microphones (Figure 3). The CoBot robot is able to navigate within our multi-floor building and performs tasks for the building inhabitants such as delivering messages, transporting items, and escorting visitors.

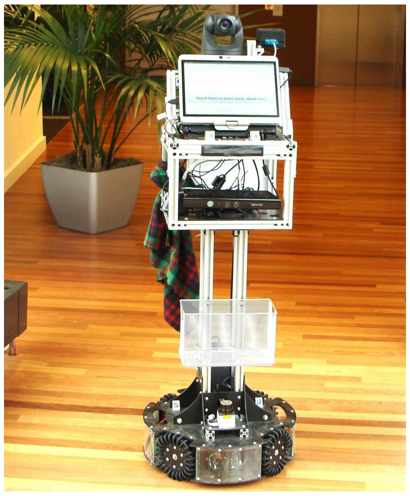


Figure 3: The CoBot mobile service robot that our approach is implemented on.

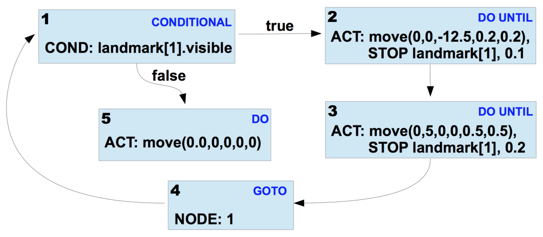
The Instruction Graph framework, and access to the robot action and sensing primitives are implemented in Python. We heavily leverage the runtime code evaluation capability of Python. Each graph node has a field for the code to be executed, and during the task execution, the Python code specified in that field is executed using the *eval()* function in Python. Therefore, each action type is implemented as a function call. We define the robot primitives as Python functions in a separate source file, and we load the primitive definitions when the robot is initialized. A separate variable dictionary is maintained for the robot primitives to provide proper variable scope for the evaluated code.

When a verbal instruction is given to the robot, the parser processes the input, and synthesizes a Python function call with proper parameters. Specifically, we use a rule-based approach that pattern matches user input to a function by searching for specific keywords. For example, a rule maps the keyword “forward” to a function that actuates the robot’s motors. The keywords “say” or “talk” map to a function that has the robot broadcast a message. 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. 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.

One downside our simple the rule based approach is that since it depends on the order of the keywords and parameters, it cannot parse instructions that contain irregularly ordered keywords and or parameters. For example, instead of

```
User : While landmark 1 is visible
Robot: What should I do in this loop?
User : Turn until landmark 1 is ahead
Robot: I will turn until I am facing
       Landmark 1.
Robot: What should I do next?
User : Forward until 0.5 meters from
       Landmark 1 max 0.2 meters
Robot: What should I do next?
User : End Loop
Robot: Loop Ended. What should I do
       next?
User : Stop
```

(a)



(b)

Figure 4: (a) The instruction conversation for the "follow the sign" task, and (b) the resulting instruction graph.

saying "go forward 5 meters", a user could say "it is 5 more meters away". With the lack of a keyword related to forward displacement, the rule-based approach cannot match this command to a function. However, the advantage of this approach is that this greatly simplifies the problem of understanding the user's request by restricting them to direct commands with recognizable keywords. As we have stated above, in case the robot fails to successfully parse a given instruction, it simply notifies the user that the last given instruction cannot be interpreted, and asks the user to repeat the instruction.

Once the function call and parameters are determined, they are placed in the code field of the created node. 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 by just providing the Python functions for accessing the action primitive or the sensory input, and specifying how the function call code for that new primitive or sensing component should be generated for a given action keyword and a set of parameters.

In our approach, the user can instruct the robot through typing the instruction or through speech recognition. For the speech recognition, we use an unofficial version of the Google Speech API. In the speech mode, the instruction giving process is triggered by pressing a *Speak* button on the user interface of the robot. Once the button is pressed, the user can speak out the instruction. An audio record process is launched when the *Speak* button is pressed, and the recording continues as long as there is a substantially strong audio signal. The recording is terminated if the input signal is weaker than a specified threshold. The recorded raw data is then converted to a mono FLAC file at 16KHz sampling rate, and then submitted to the Google servers for processing. The Google service returns a list of possible interpretations of the input, along with the likelihood values. We use the hypothesis with the maximum likelihood value as our final input. We developed our own ROS interface to the Google Speech API that runs as a synchronous ROS service. One disadvantage of this approach is that it requires Internet connectivity, and since it is a blocking service call, losing connectivity while giving an instruction might lead to a long waiting time, and an eventual error message from the robot, saying that it did not understand the instruction, and asking the user to repeat it. It should also be noted that our approach does not depend on Google Speech API or any other speech recognition software as it just accepts an instruction sentence in plain text form. Therefore, the Google Speech API can easily be discarded and substituted with a different speech recognition software.

In the current implementation of our approach, we use Augmented Reality (AR) tags as the sensory input. AR tags are visual signs that can be detected and recognized uniquely through image processing. In addition to the identification number for a detected AR tag, the relative 6 degrees of freedom pose of the tag with respect to the camera is also computed. Leveraging this detailed detection ability, the execution of a task can be conditioned upon the existence, the relative distance, and the relative angle of a certain AR tag in our approach. We use the *ar_track_alvar* ROS package with the proper frame transformations to convert the extracted relative pose of a recognized AR tag with respect to the robot camera to the coordinate frame of the robot base since the users intuitively tend to address the relative distances and angles with respect to the robot body rather than the sensor itself. The AR recognition software runs at 30Hz, and our robot primitives specification code creates a callback function to access the most recent list of visible AR tags. Another Python function wraps this functionality and the instruction parser converts instructions with sensory conditionals into the function calls to that wrapper function.

5. ILLUSTRATIVE EXAMPLES

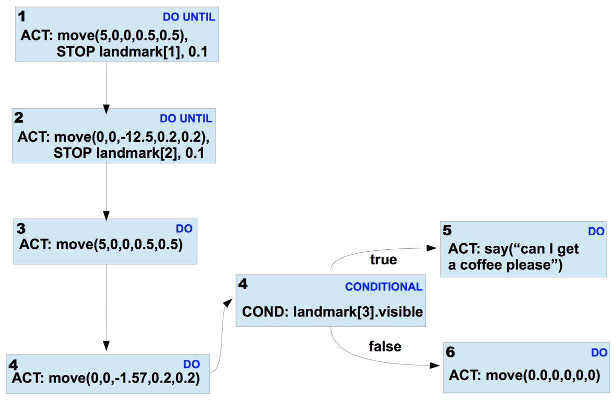
In the this section, we present various examples to task instruction and correction. When presenting examples, we first describe the tasks, followed by the initial instruction conversation between the instructor and the robot. Next, we illustrate the generated instruction graph after processing the instructions. Finally, we present an example correction scenario and discuss the changes reflected upon the instruction graph for the task as a result of the correction.

5.1 Following a Visual Landmark

This task illustrates the use of loop structures to have a continuous behavior as long as the loop condition holds. The purpose of the task is to make the robot follow a visual sign. At each execution cycle, the robot first faces itself toward the specified visual landmark if the landmark is visible, and it performs a turning motion to search for the sign if the sign is not seen by the robot. If the landmark is in the field of view of the robot and the robot is currently facing towards it, then, the robot goes toward the sign while maintaining a certain distance from it. Finally the whole process is repeated as long as the visual landmark stays in the sight of the

```
User : Move Forward until you are 0.1 meters
       from Landmark 1 max 5 meters.
Robot: What should I do next?
User : Turn right until you are
       30 degrees from Landmark 2.
Robot: What should I do next?
User : Forward 5 meters
Robot: What should I do next?
User : Turn left
Robot: What should I do next?
User : If Landmark 3 is visible
Robot: What should I do if Landmark 3
       is seen?
User : Say ``I would like to order a
       cup of coffee''
Robot: What should I do next?
User : End If
Robot: What should I do in the other
       case?
User : Shutdown
User : Stop
```

(a)


1 DO UNTIL
ACT: move(5,0,0,0.5,0.5), STOP landmark[1], 0.1
2 DO UNTIL
ACT: move(0,0,-12.5,0.2,0.2), STOP landmark[2], 0.1
3 DO
ACT: move(5,0,0,0.5,0.5)
4 DO
ACT: move(0,0,-1.57,0.2,0.2)
4 CONDITIONAL
COND: landmark[3].visible
true
false
5 DO
ACT: say("can I get a coffee please")
6 DO
ACT: move(0.0,0,0,0,0)

(b)

Figure 5: (a) The instruction conversation for the "get coffee" task, and (b) the resulting instruction graph.

robot. Both the turning and going forward motions are conditioned upon the specified visual landmark, therefore they both translate to *DoUntil* nodes. The *While* loop translates to a *Conditional* node, and a *GoTo* node. Figure 4 shows the interaction for instructing the robot to follow a visual sign and the corresponding generated instruction graph.

5.2 Getting Coffee

The second example we present instructs the robot how to go to a cafe from a starting point and order a cup of coffee. The task consists of motions performed in a mixture of open-loop and closed-loop manner. The getting coffee task also demonstrates an example use of an *if* clause to determine the course of execution depending on the visibility of a visual landmark.

In the first part of this task, the robot navigates out of our lab and around a bend. It performs these actions in a closed-loop using properly placed landmarks to orient itself. Conditionals here take the form of moving forward or turning until a landmark is seen. Figure 6 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* action type.

In the second part of the task, the robot approaches the cafe counter by going forward past the bend and then turning to the left. Then, it checks for the presence of a visual landmark. If the specified landmark is visible, the robot infers that the cafe is open, and therefore it proceeds with the ordering. Otherwise, the robot infers that the cafe is closed, so it terminates the task execution. Figure 8 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* action type, and are executed in an open-loop manner. The check for the visual landmark to determine if the cafe is open is translated into an *If* clause, conditioned on the visibility of the specified visual landmark. Finally, speaking a request for coffee requires no sensing, so it is also translated into a *Do* node.

Figure 5 shows the interaction between the instructor and the robot during instruction and the resulting instruction graph.

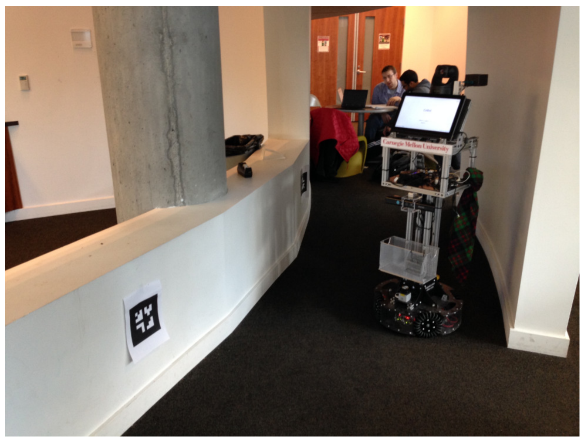


Figure 6: The CoBot navigates a bend by orienting itself with respect to several landmarks.

5.3 Task Correction

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 7 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 7: An example interaction during step-by-step execution to correct an action.

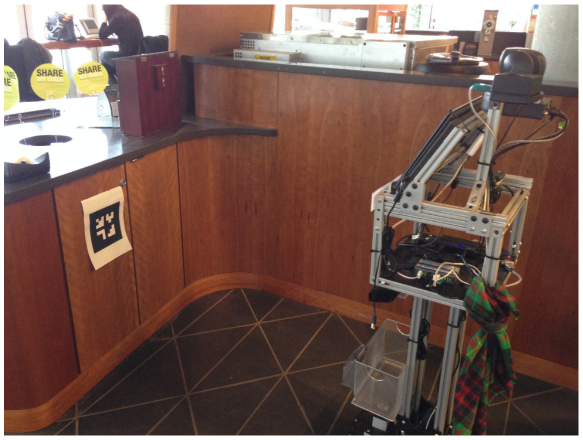


Figure 8: The CoBot checks to see if the cafe is open by conditioning on the presence of a landmark

6. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach for enabling users to compose tasks for a mobile service robot using spoken or written verbal instructions. Among the main contributions of our approach, we can name the closed-loop actuation primitives conditioned on perceiving visual signs, the ability to represent repetitive tasks using loop structures, and the correction mode that allows step by step execution and modification of the desired portion of the task. By actively seeking for the known keywords instead of trying to parse the entire instruction, our approach is partially immune to the flexibility of natural language.

We designed and implemented our approach in a modular manner. The task specification language is agnostic to the underlying robot primitives and input interface such as the different speech recognition or text input interfaces as long as the received instruction is represented as plain text. Therefore, extending the language with additional sensing and actuation elements is relatively straightforward.

We are planning to extend the current correction mode to the task instruction phase, hence, enabling the trainer to immediately see the outcome of an action and modify or correct it as desired. We are also planning to expand the correction notion to the situation-bounded corrections that can be stored with the state of the robot at the time of correction, and then retrieved and re-used when a similar situation is encountered. Furthermore, we are planning to increase the number of sensing elements based on the sensory information such as the human presence around the robot.

Acknowledgments

This research was partially supported by the National Science Foundation award numbers NSF IIS-1012733 and NSF IIS-1218932. The views and conclusions contained in this document are those of the authors only.

7. REFERENCES

[1] Thomas Kollar, Stefanie Tellex, Deb Roy, and Nicholas Roy. Toward understanding natural language directions. In *Proc. of HRI*, 2010.

[2] 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.

[3] Cynthia Matuszek, Dieter Fox, and Karl Koscher. Following directions using statistical machine translation. In *Proc of HRI*, 2010.

[4] 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.

[5] Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proc. of AAAI*, 2006.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, Johan Bos, and Ewan Klein. Personal robot training via natural-language instructions. *IEEE Intelligent Systems*, 16:38–45, 2001.

[11] Monica N. Nicolescu and Maja J. Mataric. Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proc. of AAMAS*, 2003.

[12] A. Weitzenfeld, A. Ejnioui, 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.

[13] S. Rosenthal, J. Biswas, and M. Veloso. An effective personal mobile robot agent through symbiotic human-robot interaction. In *Proc. of AAMAS*, 2010.