Interactive Aggregate Tasks

Task Pipelining for Arbitrary Robotic Systems

Boyuan Chen | Chad DeChant | David Watkins

Abstract

Robotic human interaction systems are typically designed around the use of one or more well defined skills but do not allow for custom, user-defined tasks. We have implemented a system that can take several primitive actions and allow the user to build up these actions into complex aggregate tasks. We have established a baseline of actions and commands that can be easily built upon. This system uses voice recognition to allow users to interact with a robot either visually, through a web interface using robot web tools, or entirely through voice. We have written an interface for the Fetch robot and created an easily extensible interface for additional robots such as the Mico and the PR2. This system as designed can implement complex functions defined by the user as a function of several primitive actions. The major contribution of this work is the ability to easily support multiple robots using a variety of simple primitives that each robot should be able to support

Description

We introduce several primitive actions that are all supported by the Fetch robot, and some of which that are supported by more stationary robots. The idea behind these primitives is that they are abstract enough to be useful to any users (for example, someone who may need assistance) but specific enough that a robot will be able to implement them. We have divided the primitives into movement, grasping, and posing. This allows us to determine whether a robot can take advantage of any of these specific modules and potentially will allow for future interfaces to be designed around such a system.

Primitive Actions		
	Movement	Go to X
		Follow me [for n seconds]
	Grasping	Grasp object X
		Place object X
	Request	Ask for object X as Y
	Posing	Enact pose X
		Move hand to X
		Move in direction D for M meters

Figure 1 A list of valid primitive actions that a robot can complete and the user can specify

Primitives

When deciding how we wanted the robot to behave we discussed what primitive actions and commands would need to be supported for the user to more easily understand what the system was doing and performing. As such we considered operations such as grasp, place, move to, etc. to be complex enough to be useful to the user but simple enough that chaining the actions together made sense. This creates a sort of language for executing a robotic process and could pave the way for creating a language to describe robotic actions. For movement we wanted to make sure the user could define a location and specify the robot to move to that location. For grasping we decided that grasping the object and placing the object would fit well into this category. For the vision category we wanted to make sure the robot could recognize and segment objects for later use in the grasping pipeline. Finally the robot can save different poses and execute them later on.

Fetch	Mico	Pepper
Vision	Vision	Vision
Grasping	Grasping	Movement
Movement	Posing	Posing
Posing		

Figure 2 The different robotic modules for the Fetch, Mico, and Pepper robots. Each robot has a different set of capabilities which means it would subscribe to a different set of package.

We wanted to make sure that the system was flexible enough for multiple robots - so we divided the primitives into different categories. The categories that are currently supported depend mostly on the robot we are using. We have only implemented this system for the Fetch, but it could be easily extended to a different robot by implementing the same packages for it instead.

Commands

In order to allow a user to record locations, objects, and poses, we defined a series of commands that the interface is able to understand. Recognize is a keyword for recognizing the point cloud of an object as some named entity. This allows the system to then refer to that object later in tasks. The system will also support the Execute keyword, which will task in one or more tasks, where a task is defined as a primitive or aggregate action, and execute them

sequentially. Record will allow the user to define a sequence of tasks as an aggregate. This will allow the user to define a series of tasks with custom names to give a more semantically convenient expression. For example, the user may have the following interaction with a robot: “Record grasp apple and then move up for 0.5 meters and then move right for 0.5 meters and then move down for 0.5 meters and then place apple as pick and place apple”. This will define a task “pick and place apple” that moves an apple up, right, and down on a table. Apple would need to have been defined using the recognize command or defined in the database upon startup. A task list would need to be separated by “and then” to indicate another task. A user can also define the robot’s location using the command “You are in location X”. We also want the ability to, during an execution, adjust the course of a command by some amount of movement in the case that the arm is behaving undesirably.

The full list of commands are:

- Recognize object as <id>

  Recognize an object in front of the camera for later recall as id

- Execute <task_list>

  Execute a set of primitives in order

- Record <task_list> as <id>

  Record a set of primitives for later recall through a referred id

- You are in pose <id>

  Define a pose for the robot

- You are in location <id>

  Define a location for the robot

- Adjust course by moving <direction> by <n> meters

  Cartesian transformation of the end effector

- Pause

  Pause the current action

- Stop

  Stop the current action

- Switch to robot <id>

  Hot swap the robot interface currently being used for actions

- Record trajectory

Unlock the arm and record the motion from a user

- Playback

  Play back any movement recorded from the record trajectory action

System Architecture

The system was designed with a pythonic focus in mind. There were four main modules: AlexaServer, CommandManager, ParaphraseDetector, and RobotInterface, as well as the existing repository of Fetch and Mico code. These modules each have a specific responsibility but none of them need to enforce their implementation, but rather need to communicate through a series of predefined messages in the system to ensure their integrity.

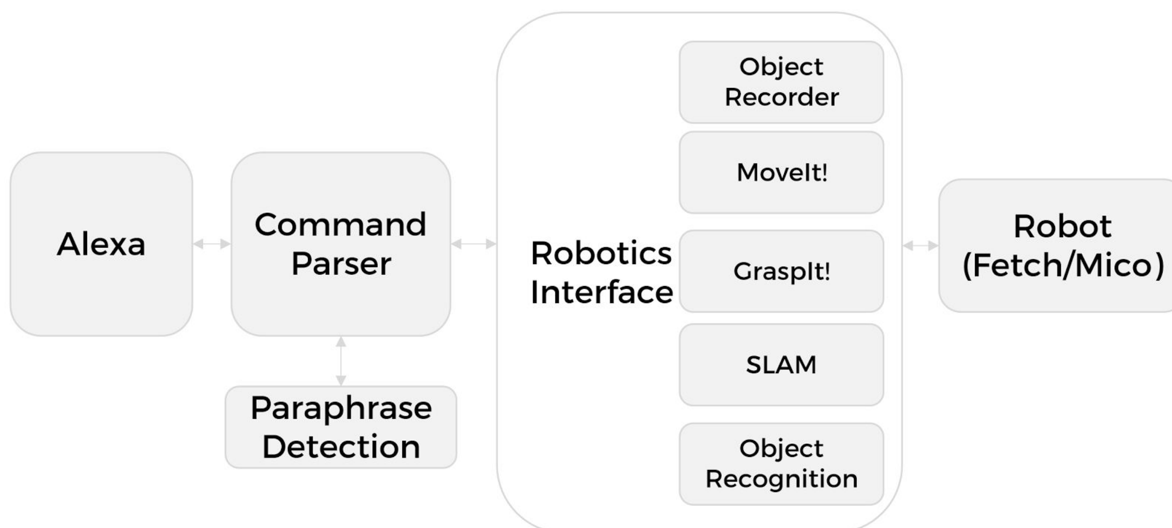

Figure 3 The overall structure of the system was to emphasize the flexibility of any one of the systems. All of these modules can be swapped out for something that uses the same interface.

AlexaServer

This server was written using the flask_ask module and supported several different command intents. This was the bulk of the command parsing from the user - beyond this a system that performed additional parsing at the task level was required, however that is discussed in a later section. The intents allowed the system to conveniently decide which command was being issued, however we frequently ran into a problem where the language model for the commands was not producing the correct intent to be called. In the future we will look to use a different pipeline that is more reliable for speech to text.

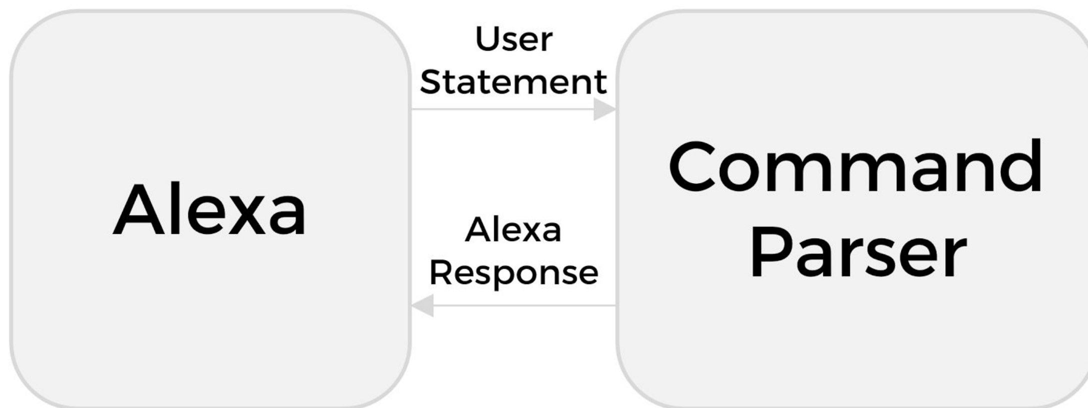

Figure 4 The Alexa and Command Parser modules should be separated so a new voice recognition system could be utilized rather than be tied to the system using it.

CommandManager

This system used python's PLY library which allows the user to build a simple context free grammar and parse text into a series of commands. This is the same architecture one would use if they were building a compiler in python. This system would take the intent responses from Alexa and then send the commands to the robot interface. It also stored information about robot poses, objects that were recognized, locations of the robot, as well as aggregate tasks. Because Alexa could not recognize arbitrary tasks, we relied on it as a speech to text tool where we then took the text and paraphrased it for both the execute and record commands. We found that this was very reliable at giving the correct task during our development process, however the Alexa server would very frequently not pass the correct result. This system would likely benefit from being able to store its state in a database later on.

RobotInterface

The robot interface is a simple template used to specify what a robot should either have defined or reject for commands. These commands include getCurrentLocation, getCurrentPose, playback, recordTrajectory, etc. By creating this interface we can easily add more robots and the same commands will be callable for those robots as well - even if they don't implement them.

ParaphraseDetector

The paraphrase detector module would take in a string and find another string that best represented the most likely sequence of tasks that is parsable using the PLY context free grammar. This system, discussed later, is what allowed us to approximate commands. It was

built using a flask web server that we communicated with through the user interface - which allowed us to have an always running paraphrase detector server and made it easier for development.

Design

We designed the system using ROS, robot web tools, and the basic fetch libraries. For each command, we require a different set of tools to be designed that will allow us to interact with the robot in the desired fashion.

Voice Recognition

We set up a server that listens for user commands and then responds by sending the registered sequence of tasks to a command server which will interpret each command dynamically.

Detection of paraphrases of commands

Use of the Echo voice interface was made difficult by its inability to accept commands if their wording was at all different from how Alexa expected them. Even filler words such as “well…” and “umm” would lead to a command’s not being recognized. To alleviate the need for a user to remember the exact phrasing of commands, we implemented a paraphrase detection system so that the system could understand input which did not exactly match what it expected.

We initially investigated the possibility of breaking up command sentences into individual words and allowing flexible user vocabulary at the word level. We discovered, however, that word-level synonym detection presented unnecessary difficulties. For example, a user may say several words which together have a meaning similar to the expected one word command, but attempts to discover this focusing on the word level may not succeed. We determined that a more robust approach was to deal with entire command sentences.

Determining whether two sentences have similar meaning is not a trivial problem. Fortunately, we were able to make use of some recent progress made in the field. A team at the University of Toronto (Kiros et al 2105) created an algorithm to embed sentences in a high dimensional vector space by training a recurrent neural network to learn how to represent sentences in such a way as to maximize its ability to output the sentences which preceded and followed the given sentence. Two sentences with similar meaning will be mapped to similar places in vector space because their context sentences (those before and after it) would be expected to be similar.

In order to learn to map sentences to a meaningful vector space, a large amount of training data and training time is required. Kiros et al used the BookCorpus dataset (Zhu et al 2015) which contains 74 million sentences in unpublished novels. In many ways this is not an ideal dataset; sentences in novels are unlikely to closely match our robot task-oriented commands. However, the sheer number of the training sentences means that the approach generalizes well

even to sentences not necessarily found in the training set. However, the less than ideal corpus used in training means that a more command- and robot-task- oriented training set could lead to better understanding paraphrases of commands than our current system.

We initially divide up the possible commands which can be given by the user into three types:

- Pose commands (e.g. “move hand to “ X)
- Object commands (e.g. “grasp object “ X)
- Place commands (e.g. “go to “ X)

We have the option of beginning a session with a user with some of these command sentences already in place (e.g. “grasp object apple”). This could be useful if the robot operates in an already defined environment with known objects. Whether or not that is the case, we can then add to those sentences as a result of user interaction. For example, a user might show an object to the robot and say: “Define object mug.” We will parse this command and add “mug” to the database of objects, along with its associated commands. When a new object such as this is first introduced, we compute the sentence vector representations for all possible commands that can involve that object, for example “place object mug” and “grasp object mug.” We use the model by Kiros et al to output the vector representation of the sentence. We find this vector at the time an object is introduced, which takes a small amount of time, because to wait until we search for a paraphrase could lead to a long delay if these vectors were not precomputed. In addition to the three types of command sentences mentioned above, we allow the user to introduce an entirely new command of a different type. In such cases we map the entire command sentence to vector space without trying to break it down into smaller elements.

Then, when a user later says a command, that command is sent to the paraphrase detection algorithm system. It then:

1. Maps the new command into the vector space of the known commands
2. Computes the cosine distance between the new command and all known commands
3. Returns the known command closest in vector space to the user’s utterance

Cosine distance is the most commonly used metric in natural language processing applications involving the distance between two representational vectors (Coccaro and Jurafsky, 1998). We experimented with other distance metrics (e.g. cartesian distance) and found no improvement in subjective measures of paraphrase accuracy for the limited number of command sentences in our repertoire.

This system helps when a user may say something different than the list of commands that are expected. It can also help if Alexa misunderstands a word - if it is only one word in a longer utterance, it is possible that the whole sentence will still be mapped to its correct expect command. For example, if Alexa misunderstood “enact pose home” as “enact poise home”, it is

likely that this would be mapped to a location near the correct command. However, the system does have its limitations. Because the training set was such a different domain from our use case, sentences which would be quite similar from the perspective of the original corpus could have quite different meanings within our narrowly focused space. We have found that this can be addressed by slightly increasing the number of core, built-in commands that are known. This gives the paraphrase detection system more vectors in the sentence vector space and increases the chances that fine-grain differences between commands will be respected.

Object Recognition

We are currently using marching cubes based on the partial point cloud of the object to recognize the object. After getting the partial mesh of the object, we load it into Graspit! to plan grasps. This method is easy to use but can generate grasps with good qualities for simple objects.

We plan to add a pipeline which enable users to pick up an object and show this object to the camera of the robot. Then the object recognition system and remember this specific object mesh and add this mesh into the database by subtracting the point cloud of human hands. The system could benefit from additional research in grasping objects out of a person's hand by subtracting the point cloud of the person's hand when initially learning the object, however we intend to implement this kind of system at a later date. For now, we can generate a completed 3D object mesh using the code base provided by this paper[1]. Users can rotate the object in front of the camera, and an sensor fusion algorithm will be applied on point clouds generated from different viewpoints. By using color detection, the human hand can be detected easily. Evenmore, we can detect where is the contact location of different fingers on the bottle by applying kd tree detection on two pieces of the point cloud (human hand point cloud and object point cloud). By far, we have all these functions up and running. However, there is a big limitation of this method: it takes about 20 mins to generate a 3D complete mesh. Thus, we may consider to use the partial mesh directly. We do not integrate this part into our system for now because of limitation of time.

The major benefits to such a system is that a user does not really have to be aware of the system or how it is storing the object meshes in order to recognize the object later. Novel objects can be directly added into our database, so that Graspit! can plan grasps based on this generated mesh.

We also attempted to fit an object-detection algorithm into our system, but ultimately did not finish integrating it with ROS in time. When completed, it will allow for the detection and

---

[1] Lending A Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions

naming of objects in the robot's field of view.

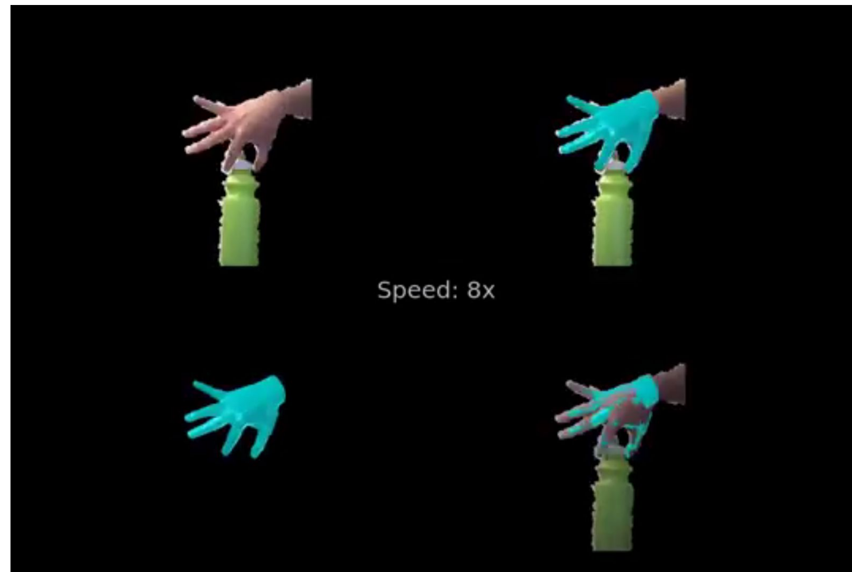

Figure 5 Hand segmentation example where they were able to color the hand in the scene

Robot Interface

We want this system to be as flexible and universal as possible. In an effort to do this we will be utilizing a fixed interface that each robot will need to implement. Which robot is currently interacting with the system will depend on the startup scheme, however for the purposes of this paper we will be implementing solely for the Fetch. If we have time we will then look at implementing this for the Mico.

Movement Interface

This interface is only relevant for the Fetch. The movement of the Fetch uses SLAM and is incredibly easy to use. We wrote an interface that allows us to acquire the current position and go to a position defined previously. When users drive the robot to different locations, robot can also remember the name of the location by using the word provided by users through speech recognition system. Later, through the name saved before, robot can go to specified location directly.

Grasping Interface

We use Graspit! commander to define the grasping interface which will be a simple utilization of existing grasp software. We will be using the version that uses shape completion so as to get a more accurate grasp on the point cloud in the scene rather than the marching cubes based on partial point cloud provided by the depth sensor on the robot. But for now, our current version works robust for simple objects and can pick up the object successfully.

Placing Interface

Users can specify the place location after robot picking up the object successfully. After deciding a place location, robot will use Moveit! to plan a arm trajectory to move the arm to place location.

Cartesian Interface

Previously, there is no code base provided online for Fetch robot to do cartesian transformation. Here in this project, we provide a cartesian transform for Fetch with the help of Moveit! library. Users can also specify the direction and distance they want the robot end effector to move to through speech recognition system. Then, our cartesian interface will take in the relative position and orientation and make changes based on current position of end effector. However, we currently decide the direction based on the current /approach_tran frame. Thus, the end effector might not move to the same direction specified by users. One way we will solve this is to change the parent frame of the next end effector as robot /base_link instead of the palm. This "global" coordinate system configuration should fix this problem.

Record and Play-back arm Trajectory Interface

After sending "start recording " command line into robot through speech recognition system, users can move the robot arm manually to different pose, for instance, closer to an object and pick it up. This is realized by adding a gravity compensation on the arm controllers, or all the arm controllers are locked up and users cannot move them around manually. The key joint values of the arm joints will be recorded into a list as long as they are different from the joint values of previous step. With Moveit!, we can send the joint value lists into it and get a set of arm trajectories from Movie!

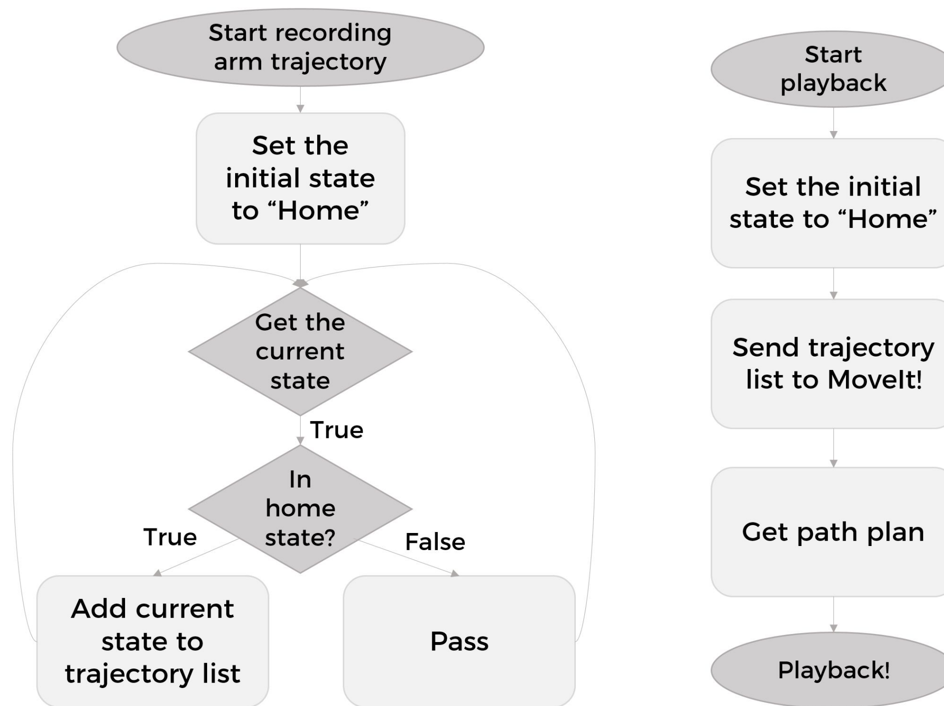

Figure 6 Pipelines for the record trajectory and playback commands

Task Interface

The task interface will be a interpreter sitting on the ROS network listening for sequences of tasks to be executed. It will control blocking and preempting of the network as well as any adjustments that may need to occur. Adjustment mostly affect grasping and posing commands so it will need to be able to splice together commands.

Conclusion & Future Directions

We have built an extensible platform for users to interact with robots using natural language and a web interface. We have addressed the most significant limitation of natural language interfaces, i.e. the ability to understand paraphrases of known commands. We have built a command parsing framework able to take user input and carry out functions, including accepting newly defined user objects and tasks.

Our system was significantly hobbled by its dependence on the Amazon Echo tool for accepting verbal input. The immediate next step to improve the system would be to switch to a better functioning application for transcribing spoken commands.

Bibliography

Coccaro, Noah, and Daniel Jurafsky. "Towards better integration of semantic predictors in statistical language modeling." *ICSLP*. 1998.

Kiros, Ryan, et al. "Skip-thought vectors." *Advances in neural information processing systems*. 2015.

Zhu, Yukun, et al. "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books." *Proceedings of the IEEE International Conference on Computer Vision*. 2015.

Videos

Pick up pringles:

https://drive.google.com/open?id=0B6Y-RTOHkfaZbGpQX1NfeEF0S1U

Playback Trajectory:

https://drive.google.com/open?id=0B6Y-RTOHkfaZRjFuNzNZR0M0TlU

Link to source

https://github.com/CURG/InteractiveAggregateCommands

Contribution discussion

Chad did research and coding on our initial idea that we then determined was too ambitious for the class project. He then researched different approaches to semantic similarity detection and implemented our paraphrase detection system and its associated server.

Bo designed different robot interfaces and exposed these interfaces as independent submodules so that other applications could use them directly. Bo also helped with the integration between robot platform and speech system. In addition, he also did research on object mesh 3D reconstruction and implement the code base provided by the paper mentioned above into our system.

David was responsible for building the frontend of the application - specifically equipping Alexa with the necessary functionality to support parsing commands and building a system to pass these messages to the robot interface. He also defined the primitive functions and commands for the system to use in conjunction with Bo’s available and potential input into the system. Finally, the system that takes in raw text from the Alexa controller and converts it into a command that could then be executed by the robot interface.