Hierarchical Task Networks and Plan Management

James Forkey and Trevor Hodde

Worcester Polytechnic Institute

Prof. Sonia Chernova

April 30, 2013

Abstract

Current robot systems are programmed for their intended function. Current methods of intelligent learning designs are highly inefficient and computationally expensive. This paper introduces the topic of task models and expresses the basic concept and implementation of utilizing a collaboration manager, Disco, integrated with the Robot Operating System (ROS). A Hierarchical task network (HTN) is an effective method of building a model easily understood by humans while being efficient to process and implement with robots.

Introduction

Hierarchical task networks are an efficient method of constructing an understandable task structure. We attempt to answer the long standing issue of *knowledge acquisition bottleneck*. Currently, most robots are pre-programmed with their intended purpose. Even those that learn need to be given a significant amount of knowledge about their intended task, making them less transitional and highly domain specific. Other works have shown hierarchical task networks are significantly helpful in many learning different areas of plan modelling (Garland and Lesh 2002).

We aim to build a plan model that is easy to comprehend for general users while maintaining the versatility to use with a robot. The goal is make learning from demonstration (LfD) more efficient and handle complex tasks not currently achievable. This will potentially move robots out of the domain of repetitive tasks and into the world of realistic learning and execution of every day human tasks.

Generating a set of tasks to construct an HTN may sound simple in theory, but it is quite difficult in real-world activities. This is due to the breakdown of abstraction. That is, how do you distinguish the difference between primitive and non-primitive tasks? For example, the task of petting a cat intuitively seems like a primitive task. However, it may be better broken down into more literal tasks, such as, place hand on cat head, gently stroke to tail, lift hand. Now it becomes obvious that the general construction of tasks is a difficult model to engineer and maintain an understanding for both humans and robot interpretation.

Approach

As a proof of concept, we use the daily task of setting a dinner table, where the actual executed task on behalf of the robot is performed by a turtlebot via a simulator provided by ROS. The general setup follows a client-server setup. Disco, the collaboration manager, sits within the client. The robot acts as the server. Tasks are issued to the robot by requests. The robot performs the tasks, then issues a response. Disco interprets the response and can decide on the next step. See Figure 1 for a general overview of message flow.

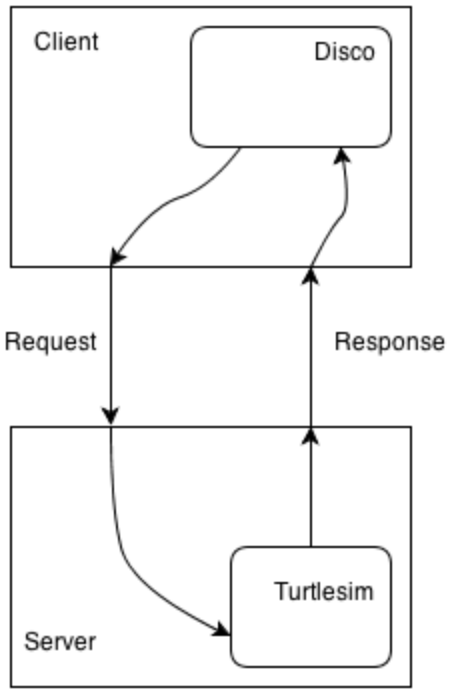

Figure 1: Message flow of system.

Hierarchical Task Network

Hierarchical task networks are composed of primitive tasks, compound tasks (commonly referred to as non-primitive tasks), and goal tasks. Primitive tasks are actions that can be executed. They represent the leaves of the HTN. Non-primitive tasks are more complex tasks that need to be decomposed further before an action can be executed. Generally, a non-primitive task is composed of primitive tasks. In addition to having non-primitive tasks, there may be ordering constraints within the non-primitive task. For example, the task of executing a table-

place setting may be composed of setting a plate, napkin, fork, knife, spoon, and glass. Clearly, the napkin must be set before the fork or knife. Therefore, there is an ordering restraint on these tasks. However, it is not necessary to set the napkin before the glass or spoon. See figure 2 for a visual representation of the HTN for a possible task model for setting a dinner table.

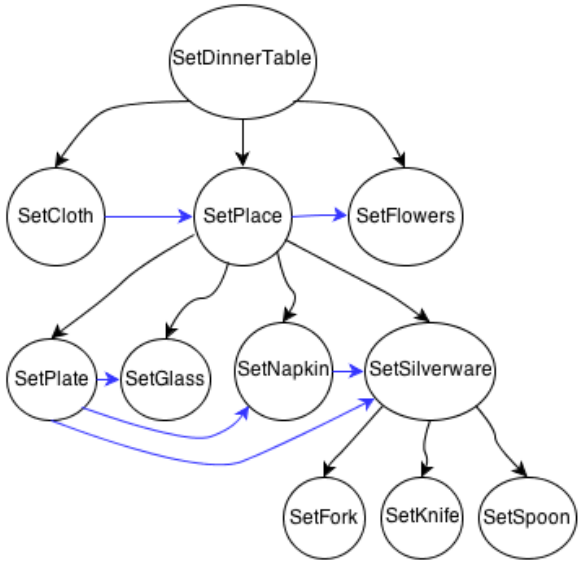

Figure 2: Potential HTN for setting a dinner table.

The tree above was the HTN we used as a foundation for the project. The black nodes in the tree represent tasks. Nodes with black lines leaving them are non-primitive tasks. The overall goal task in this example is SetDinnerTable. As said before, the nodes without any black lines leaving them (leaves) are primitive tasks. These are the tasks that will trigger the robot to execute an action.

For simplicity, we made setting the plate a precondition to setting the silverware. The only reason for this was to take into account the fact that the robot may need a point of reference to place the other objects and the plate justified a good point of reference for all other objects.

Preconditions

The best course of action from a user would be to construct a hierarchical task network, defining primitive tasks, non-primitive tasks, and any preconditions. See Figure 3 for an example list of non-primitives and primitive tasks.

Non-Primitive: SetDinnerTable, SetPlace, SetSilverware

Primitives: SetCloth, SetPlate, SetGlass, SetNapkin, Setfork, SetKnife, SetSpoon, SetFlowers

Figure 3: Set of primitives and non-primitives for the SetDinnerTable task model.

Disco reads this set of declarations from an XML file and a properties file. The XML file defines the name of the task (e.g. SetSilverware, SetFork), and the script to be executed if one exists. The script used in our implementation calls a buildRequest method from the client node with the name of the task to execute.

Client

The process begins with the client. After the Server node, client node, and the turtle simulator have been launched, a command-line terminal and turtlebot simulator will appear. The user issues a new task with:

task SetObject

Where object is the intended object to set.

This replicates the process of a robot successfully learning a primitive task or the robot already knowing the primitive task to be executed. The turtlebot will do some arbitrary movements depending on the executed task. The user continues by executing all of the tasks they wish to add to the current task model. Once they have completed, provided they have entered the tasks in the correct order, Disco will have

constructed a hierarchical task network similar to Figure 2.

Server

After the robot has finished executing some primitive task, it will issue a response to send to the client, which Disco will interpret and decide the next course of action. Disco supports three different responses for an executed action. True means the primitive task executed succeeded, false means the primitive task executed failed, and unknown means it cannot be determined.

The server receives a request, determines which primitive task should be executed, and proceeds to execute the primitive task. Once the robot has finished, the server will issue a response to the client. The server uses custom messages of type *RosD*. These messages were created for extensive expandability.

By using custom messages, we can construct requests and responses with whatever information we choose. Alternatively, we would be restricted to using ROS's built in message types. Currently, requests are *strings* and responses are *long integers*. This is easily changeable with custom messages by editing the files in the rosd_messages directory.

When the user has finished, they can execute a command to build an XML file of the current task model. This model can then be reproduced on demand, stored for later use, or transferred to another robot who may use the model for their own benefit. This means that robots that are linked over a network could share the plan and have immediate *knowledge*.

Results

We experienced significant complications throughout the entire process. There was a distinct lack of support and inadequate documentation from ROS, rosjava, and Gazebo. At the time, Gazebo was undergoing massive redesign and updates. This broke most of the ROS tutorials that utilized Gazebo. We originally intended to use a PR2 robot, but after further inspection and uncovering the current difficulties with Gazebo, we decided to use the turtlebot simulator.

Rosjava had also recently undergone significant changes. The documentation available was limited and support was only offered by questions on the ROS forums. In total, six questions were asked on the ROS forums. Very few questions were answered by other users. The questions that were answers didn't help very much. It appeared the entire Rosjava community was in the process of adapting to the new structure.

During initial setup, gradle, the build system Rosjava recently added, also caused some trouble. The version provided in the Rosjava tutorial was outdated. Tracking down the culprit of this issue was yet another hurdle to jump over.

Initially, we used a publisher/subscriber set up to issue messages to and from the robot. We realized the correct implementation would utilize rosjava services. After we switched to using services, it became evident that rosjava has no built-in means of blocking.

We did, however, manage to get a working structure that executes primitive tasks.

Conclusions

Unfortunately, we were unable to implement non-primitive tasks. This was the core component we wanted to address. Future work would address this and implement non-primitive tasks. The build component doesn't build the current tree. The focus of priorities was functionality and time constraints limited the amount of progress made. Progress was stifled by constant hurdles with countless dependencies, new software, and a lack of documentation and support.

Disco efficiently blocks until a response is received. However, the server does not block while the robot is executing a task. In essence, blocking works on the client side but not the server side. This was due to implementation of the robot. Currently, the robot is issued commands via processing shell commands. The effective and correct method would use a publisher to issue the commands and wait for the robot to finish, then send a response to the client.

There is substantial room for future work. This piece only scrapes the top. Future work would include adding effective blocking the server, switching the messages sent to the robot from shell commands to published ROS messages, adding support for non-primitive tasks and decomposition, and implementing the build XML file for expandability and usefulness.