Advances in Cognitive Systems 6 (2018) 159–176 Submitted 6/2018; published 9/2018

One-Shot Interaction Learning from Natural Language Instruction and Demonstration

Tyler Frasca TYLER.FRASCA@TUFTS.EDU
Bradley Oosterveld BRADLEY.OOSTERVELD@TUFTS.EDU
Evan Krause EVAN.KRAUSE@TUFTS.EDU
Matthias Scheutz MATTHIAS.SCHEUTZ@TUFTS.EDU
Department of Computer Science, Tufts University, 419 Boston Avenue, Medford, MA 02155 USA

Abstract

One-shot learning techniques have recently enabled robots to learn new objects and actions from natural language instructions. We significantly extend this past work to one-shot *interaction learning*, from both natural language instruction and demonstration, where a robotic learner not only learns the actions appropriate for its role in the interaction, but also the roles of the other interactors. The resulting knowledge can be used immediately such that the robot can assume any role of the learned interaction, to the extent that it can perform the required actions. We demonstrate the operation of the integrated architecture in a handover task in real-time on a robot.

1. Introduction

Human actions in the social domain are intrinsically characterized by their social context, and often depend on sequences of actions performed by multiple agents. Consider simple social tasks such as shaking hands, holding a door for someone, waiting in line, or passing an object to another person. Current techniques for online robotic action learning cover a broad range of approaches, from "learning from demonstration" to "learning from (natural language) instructions", but none of them allow robots to learn *interactions* among multiple agents, let alone in "one shot".

We propose the first approach to *social one-shot interaction learning*, from both natural language instructions and human demonstrations, combining aspects of learning from demonstration and learning from instruction, while doing so (1) in "one shot", i.e., from one instruction or demonstration, and (2) for interactions among multiple agents. We leverage the perceptual, learning, reasoning, and action capabilities of a cognitive robotic architecture that allow us to ground actions through natural language instruction as well as visually observed demonstrations. In order to incorporate the new interaction learning method, we significantly enhance a cognitive robotic architecture to allow the system to model multiple agents, observe the actions of those other agents including their conditions and effects, and understand references objects involved in the agent's own actions and the actions of others.

With these extensions in place, we can integrate the novel interaction learning system into the architecture's action system. The most important benefit of this integration is that after learning an interaction from one exposure, the agent does more than perform the actions it just learned for *its role in interaction*: it can also perform *the role(s) of any other interactants* without requiring any additional training. For example, if the agent learned how to pass a knife safely, it can also safely receive the knife.

The rest of the paper expands on these ideas as follows. We start with a brief survey of recent work on robotic learning (in particular learning from demonstration) which motivates our chosen approach. After describing the functionality required for an agent to be able to learn multi-agent interactions from a single exposure, we explain our approach to interaction learning, which involves extending the functionality of several components of a cognitive robotic architecture. We demonstrate our approach in an example scenario where a human teaches a robot how to hand over an object and demonstrate that when the interaction is finished, the robot can perform the interaction in either role. We conclude with a discussion of our approach in the context of the existing body of work and suggest directions for future work.

2. Motivation and Background

Learning from demonstration is the process through which a robot is taught new actions online through a human guided interaction with the environment. Work in this field has primarily focused on teaching robotic agents how to perform actions through kinesthetic learning (e.g., Akgun et al., 2012a; Akgun et al., 2012b) and from observing demonstrations (e.g., Chernova & Veloso, 2007; Ghalamzan et al., 2015), possibly aided by natural language (Argall et al., 2009). More recently, there has also been interest in making these demonstrations more interactive by allowing agents to actively seek out information (e.g., Mohseni-Kabir et al., 2014; Hayes & Scassellati, 2014; and Mohan & Laird, 2011).

Most of these techniques (1) are unimodal (relying only on kinesthetic learning, observation, or natural language) and (2) require multiple demonstrations to learn the structure of the action or skill. The approach described in Akgun et al. (2012a; 2012b) focuses on learning low-level manipulation trajectories through kinesthetic demonstration. Instead of recording all the points along a specified trajectory only the most relevant points are acquired, and then after multiple demonstrations, the agent is able to generalize the trajectory based on those keyframes. Although this approach is effective for learning trajectories, it is context specific and requires teaching all possible trajectories.

Other approaches rely on observing a demonstration and extracting relevant information. For example, policy learning observes and learns mappings between states of the environment to actions (Chernova & Veloso, 2007; Ghalamzan et al., 2015). In a real-world setting it might not be feasible to provide enough demonstrations for each action to learn the correct policy. This is especially relevant in social situations where the actions of other agents may greatly expand the space the policy must consider. Other techniques generalize over different observations by using high-level task representations. Nicolescu and Mataric (2001; 2003) describe an approach which use natural methods for task learning through human-robot interactions. After multiple demonstrations, the agent is able to create a generalized task representation. Hayes and Scassellati (2014) have extended hierarchical task networks to encapsulate subtask ordering constraints. Their approach finds a clique of tasks that have common preconditions and postconditions. Then the method finds chains of tasks where the postconditions of one task satisfy the preconditions of the next task.

More recently, learning from demonstration has been extended to incorporate more interactive demonstrations. Mohseni-Kabir et al. (2014), Hayes and Scassellati (2014), and Mohan and Laird (2011) have developed approaches that let agents actively learn missing information. Using a mixed-initiative collaboration, Mohseni-Kabir et al. (2014) report a system that learns hierarchical task structures by finding temporal constraints and asking questions. Mohan and Laird (2011) discuss the processes and challenges of designing an agent capable of learning through interactions. They propose integration of a component into the Soar cognitive architecture which, if the agent cannot derive the required knowledge, allows the agent to ask the human for information.

All the previously discussed methods require multiple demonstrations to learn task representations. However, in social situations, robots will need to learn task representation within a single demonstration. The use of natural language instructions and a shared understanding of concepts between instructor and agent, over comes this need for multiple examples. Allen et al. (2007) developed the PLOW system, which teaches an agent how to perform a task through natural language in a single demonstration. Rybski et al. (2007) created a framework that utilizes natural language and a single demonstration to learn hierarchical task representations. More recently, the system described in Scheutz et al. (2017, 2018) learns new objects and actions through natural language from individual examples.

These approaches focus on learning actions for a single agent, but they are insufficient for learning interactions among multiple agents. To learn an interaction, the learner must be able to understand actions from the perspectives of all interactants. Consider an interaction between two agents where one agent hands the other an object. For this transfer to succeed, both agents must understand when an agent is offering the object and when it is released. Both agents need to recognize the actions the other has performed and use that information to determine their own actions. If the agent holding the object cannot detect when the other agent is ready to grasp, it may release it too soon or perhaps never release it at all. In short, an agent's representation of the interaction must include the actions of all of the participants.

Our goal of *one-shot interaction learning*, therefore, relies on the learner observing, identifying, and tracking the actions of all agents involved in the interaction, and representing them in order (i.e., when an action occurred and when its execution was successful in producing its effects), irrespective of whether the learning agent can carry out all the actions.

3. Interaction Learning among Multiple Agents

Before we introduce the interaction learning algorithm, we briefly describe the DIARC[1] cognitive-robotic architecture (Scheutz et al., in press) which we use as an implementation environment. The functionality required to support the interaction learning method was not present in DIARC, so we also describe the extensions we made to enable it. Existing functionality provides many critical auxiliary functions, including full-fledged one-shot action learning from natural-language instructions (Scheutz et al., 2017, 2018). Prior to our extensions, the actions learned using DIARC involved only those of the robotic agent itself, not the actions of others. Additionally, objects involved in these

1. DIARC is a *Distributed Integrated Affect, Reflection, Cognition* architecture for robots.

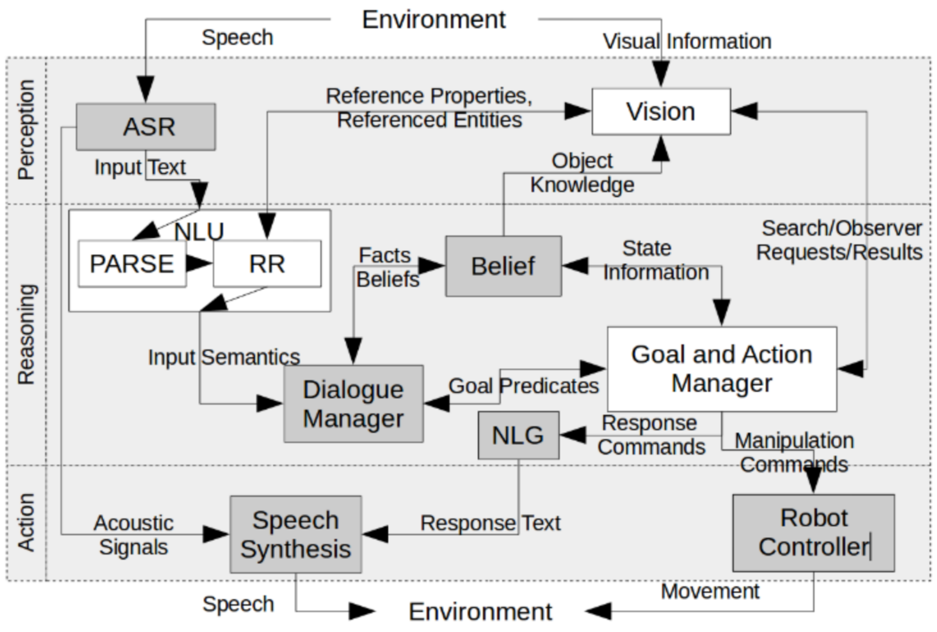

Figure 1. Overview of relevant components and their connections (modified components with white background, see text for details). The architecture receives information about the world through its Automatic Speech Recognition (ASR) and Vision components. The information from the Speech Recognition is passed to Natural Language Understanding (NLU) and the Parser (PARSE) converts text to a semantic representation. Reference Resolution (RR) then grounds the semantic information with its perceptions of the world. This semantic information is used by the Dialogue Manager, the Goal and Action Manager, and the Belief component. These components allow the agent to reason about language. They also let the agent reason about visual information. They interact with the Vision component to search for and ground visual information about entities in the world. The reasoning done by these components lets the agent interact with the world through language via the Natural Language Generation (NLG) component and Speech Synthesis component. The agent can also interact with the world physically via the Robot Controller component.

actions were not consistently modeled over the course of action execution. To enable the learning of interactions, we extended DIARC for multi-agent action execution, the detection and observation of other agents, and the tracking of task-related objects across the execution of an action. Figure 1 depicts all components involved in interaction learning, with the components extended in this work highlighted (white background). We modified: (1) the Goal Manager component to include models of other agent's actions, observations of those actions, and the ability to learn new interactions; (2) the Vision component to observe the actions of other agents, and to ground specific agents and objects in the physical world across action sequences; and (3) the Natural Language Understanding component to understand references to objects and agents across sequences of instructions, allowing the agent to know which objects and agents are involved across every step in a sequence of actions.

3.1 Action Representation and Execution with Multiple Agents

The Goal Manager component determines what the agent should do and how to do it. This section provides an overview of how the module works and what we have added to prior versions to enable *one shot interaction learning*. The module receives goals, in predicate form, from running DIARC components (including itself). These goals represent states desired by their requesting component. When a goal is received, the Goal Manager first determines if it is a suitable goal, if so, it selects an appropriate action to satisfy that goal, and finally manages the execution of the selected action in an effort to reach the desired goal state.

Action Representation. The actions available to the Goal Manager are stored within the Action Database. Each action in the database is represented by a name, arguments, conditions which must be true to execute it, and the effects of execution. An action is either primitive or an action script. Primitive actions are advertised by components and provide the core set of functionality available to the system. These map to some functionality of their advertising component. For instance, a vision component might advertise actions like *findObject*, while a manipulation component might advertise actions like *graspObject* or *moveObject*. Primitive actions are dynamically updated in the database whenever a component initializes or terminates. Action scripts are composed of a sequence of primitive actions and other action scripts with the required agents, action operators (e.g., arithmetic, comparison, etc.), and control elements (e.g., conditional statements and loops).

Action Execution. When the Goal Manager receives a goal submission (specified by a desired postcondition), it initializes the action selection process and then, if a suitable action is found, manages the execution of the action. Otherwise, execution fails and failure conditions are reported.

To start the execution of the selected action, the Goal Manager must first verify its preconditions. Previously, when checking to see if the preconditions held, the Goal Manager only checked its record of previously recorded states which have come about through the execution of previous actions. However, this record might not accurately depict the current state of the world, because there may be states relevant to the action that are not related to any of the agent's previous actions. Thus, to ensure the preconditions hold in the current environment, we extend the Goal Manager to give the agent the ability to observe the conditions of the action. The observation mechanism described in Section 3.2 can be used to make observations about the world state, checking the state of events, objects, and agents. If the preconditions are not satisfied after observation, execution is canceled, and failure conditions are reported. In this case the failure condition is that the preconditions of the action could not be met.

Because actions can also have overall conditions which need to be true throughout the execution, the Goal Manager needs to be able to continuously observe the environment. Previously, it assumed the conditions held for the entire action execution. However, with stochastic environments this is not always the case. Thus, the Goal Manager is extended to start observers for overall conditions. If at any point an observer finds that an overall condition no longer holds, it will cancel the action and the Goal Manager reports the failure conditions.

Once the agent knows the preconditions are met and it has initiated the overall condition observers, it can continue the execution process by checking if the selected action is a script or a primitive. If the action is a script, its subactions are added to an execution stack and execution continues with the first subaction. Otherwise the action is a primitive action, in which case the Goal Manager checks the agent assigned to perform the action.

Previously, the Goal Manager only handled actions with a single agent, so it did not need to determine which agent would perform the action. However, in a social environment, robots must perform actions with other agents, so we have extended the Goal Manager to check the assigned agent for each action. If the agent is a DIARC agent that the robotic architecture controls, then it will proceed normally with the execution by directing the appropriate agent. However, if it is not a DIARC agent (e.g., humans or other agents the system cannot control), the Goal Manager knows that another agent is tasked with carrying out the execution. Here, the Goal Manager must observe the other agent and the effects of the action so the agent knows when to execute the next action.

At the end of an action, the agent needs to confirm that the effects of the action have been satisfied. Previously, for each effect, the Goal Manager assumed the effects held and recorded them. However, it is not always the case that the actions complete successfully. Thus, we have extended the Goal Manager to first spawn an observer, if one is available, to check the environment and then record state. If no observer is found, the Goal Manager assumes the effect holds, and records the state. Because the effects of the action can contain the agents and objects involved in the action, the observers are used to confirm that other agents have performed their appointed tasks. If all the effects of the action are met, the action succeeds and further execution continues; otherwise execution fails and failure conditions are reported. Here the failure condition is that execution of the action did not produce the intended/expected effects. Extending the Goal Manager with the observation mechanism and multi-agent action execution allows the system to not only learn single agent actions, but also actions containing multiple agents.

3.2 Observers and the Actions of Other Agents

In order to track the progress of an action in a demonstration or execution, an agent must be able to observe the conditions and effects of every step of that action. For instance, when an agent picks up an object, it must observe that the object is lifted off the table and is in its end effector. Of course, the agent can blindly execute the action sequence, never making observations about the progress, and assume a successful completion if no motor commands failed, but it will never truly know if the action is successful unless the necessary observations are made. This blind execution, while sometimes sufficient for execution of individual actions, is never sufficient for multi-agent interactions, where an agent's only means for tracking the progress of another agent's action is through observation. Critically, this mechanism is what enables the Goal Manager to leverage perception components in the system to track the progress of action execution and follow demonstrations.

Before the Goal Manager tries to observe the environment for a condition or effect, it checks to see if there are any available observers for that condition/effect. If an observer is found, the Goal Manager executes the observation. If an observer is not found for a precondition or overall condition, the Goal Manager reverts to checking whether it holds using its record of states produced by previous actions. We extend the Goal Manager to let the agent observe the conditions and effects of the action. When no observer is found for an effect, then the module assumes that the effect holds and records it.

Observers are special case of primitive actions, and are automatically discovered in much the same way as action primitives. For a primitive action to be an observer, it must adhere to a particular method signature and explicitly advertise the types of observations that can be made via predicate descriptions (e.g., $touching(X, Y)$). The observers available to the Goal Manager are automatically updated in the Action Database when a component connects and disconnects from the running instance of DIARC, and can easily be looked up by their advertised predicate description.

To make use of these observers during action execution, the Goal Manager looks for available observers in the database while attempting to verify the effects and conditions for an action. If an observer is found for a condition or effect, a new observer subaction is spawned by the Goal Manager. If the observer is in service to a precondition or postcondition, then the Goal Manager waits until the observer succeeds or times out and fails. If the observer is in service to an overall condition, a concurrent observer is launched that is capable of interrupting the Goal Manager if at any point a required observation is not met.

The same observation mechanism used for individual action execution can also be used for multi-agent interactions by leveraging the multi-agent task representation presented in Section 3.1. If, for example, an agent is performing a *pick-up* action, an observer for *touching(self, object)* might be used to verify the *pick-up* action completed successfully. By swapping an agent's role, the same mechanism can also be used to observe another agent performing the same *pick-up* action, where the observation might instead be *touching(Sam, object)*. This swapping of roles is crucial, and is what lets the same action representation be used by an agent to assume any role in an action.

It should be noted that this flexible role switching within Goal Manager is tightly coupled with the system's ability to make observations about the world and other agents. The kinds of observers in the cognitive robotic architecture are dependent on the kinds of sensors and perception components available to the system. More sophisticated perception components will allow for richer observations and more complex multi-agent interactions. For the work presented here, we have extended the Vision component to include an observer capable of observing *touching(X,Y)* events. This observer is capable of detecting touching events between any two objects (i.e., *X,Y*) that can be detected by the Vision component. Here, the touching event is determined based on the proximity of the point clouds representing the detected objects in the scene.

3.3 Understanding References Across Actions

As described in Section 3.4, in order to learn generalizable representations of actions an agent must know which steps of an action involve specific entities in the outside world. To model these entities, an agent must be able to visually perceive them as well as understand when they are referenced in natural language. The GH-POWER (Givenness Hierarchical – Probabilistic Open World Entity Resolution) algorithm provides a framework with which an agent can understand references to entities in natural language, and ground them in its perceptions of the world (Williams et al., 2016; Williams & Scheutz, 2017). To integrate GH-POWER, we extend the Natural Language Understanding, Vision, and Goal Manager components beyond the versions used in Scheutz et al. (2017).

In DIARC, the Natural Language Understanding component receives natural language utterances in text form from the Automatic Speech Recognition component and converts them into the semantic representation used throughout the rest of the architecture. We extended the functionality of the Natural Language Understanding component in two ways. First, we changed the semantic representation it produces so that information required for GH-POWER is included, when previously it was ignored. Second, we added a reference resolution step which occurs after the initial parse is generated. This step grounds references to objects in terms of real world (or hypothetical) objects, which allows them to be tracked across multiple instructions, and their associated actions.

Syntactic and Semantic parsing is done in the Natural Language Understanding component using a combination of syntactic rules in Combinatorial Categorial Grammar and semantic rules in lambda calculus. The parser maps natural language to a semantic representation which can be understood by the other components in the system (Dzifcak et al., 2009; Scheutz et al., 2017). Prior to this work there was not a notion of consistency of referents across utterances. For example, if the agent is instructed to “grab the knife” and then to “release the knife” it knows that the object that it has grabbed is a knife, and the object it has released is a knife, but it does not have an explicit representation that it was the same knife in both cases. As actions grow more complex, an explicit understanding of such references is required.

The integration of GH-POWER allows for such an understanding. The semantic representation previously used in Natural Language Understanding did not include all of the necessary information required by GH-POWER. We modified the parser so that it (1) explicitly labels all of the referring expressions in an utterance, and (2) adds syntactic information to the parse which was not previously included. Take for example the semantics of the utterance “grab the knife” (spoken by the interlocutor “Sam”):

old: $INSTRUCT(Sam, self, grab(self, knife))$
new: $INSTRUCT(Sam, self, grab(self, X) \wedge kinfe(X) \wedge definite(X))$

In the new semantic representation the referring expression is denoted with the variable X and the information that referring expression X is included. Such information is present nowhere in the previous representation.

Now that there is this richer representation of the referring expression it is possible to perform reference resolution using the GH-POWER algorithm. While reference resolution is executed by the Natural Language Understanding component, it requires information that is stored in other components, and those components must be updated to provide the Natural Language Understanding component with this information. In order to ground referring expressions in real-world entities, Natural Language Understanding must consult with components that can perceive the outside world. In the case of our example the Vision component is consulted. (In the case of location a mapping component could be used, or in the case of speaker verification a speech component.) To enable the appropriate access of information, we created a new connection between the Vision component and Natural Language Understanding through which the former advertises all of the types of objects it is able to detect in the real world, as well as specific properties of objects that it can detect (shape, color, etc.). When Natural Language Understanding receives a reference to resolve it checks the reference’s semantic descriptors against the descriptors advertised to it by other components, like the Vision component. When there is a match the GH-POWER algorithm is used to consult with that component. If it is the first time the consultant has been queried about this specific reference it creates an internal representation of it. If not, the reference is matched with its previously generated representation within the consultant.

Once all of the references have been resolved the semantics of the utterance are updated to reflect the IDs of the resolved references. The utterance from our previous example would have the semantics: $INSTRUCT(Sam, self, grab(self, objects\_0))$ where the reference id *objects_0* represents the consultant and index of the reference.

This reference id representation is used during action execution in the Goal Manager. Consequently the representation of objects used in its actions must be updated to use this new format. In some cases a reference id may not be grounded in a real-world object until it has been used in an action. If an action occurs that grounds a reference (such as a visual search) the consultant responsible for the reference is updated as a result of the action. In our working example, until a visual search is done, the Vision component knows that there is some entity with the property knife that has been referenced but it has not yet associated it with any of its perceptions of the real world. When the knife is grounded through a visual search (for example the agent being instructed to "find the knife"), the result of the search is bound to the entity which is bound to the reference id for the object. The visual search results associated with the knife are now available to any subsequent actions which involve references to the knife.

The ability to understand these references and ground them in perceptions allows for actions that can have consistent representations of the objects involved in them. This ability allows for generalizations in the learning process where the agent is able to understand which arguments need to be consistent across multiple actions, and which do not.

3.4 Learning Actions With Multiple Agents

The DIARC extensions described in the previous sections enables the system to execute and learn actions with multiple agents. The action learning process is initiated through the submission of a goal containing the post condition of the action to be learned. For example, if the action to be learned is 'pick up the knife', then the goal is *startActionLearning(pickUp(self, object_0))*. This triggers the creation of a new *Learning State* which keeps track of information relevant to the action being learned: (1) name of action (pickUp), (2) initial arguments (self, object_0), (3) action step queue, (4) sets of action effects and preconditions (lines 3-4, Table 1). When a new Learning State is created, the learning framework checks to see if the agent is already in the process of learning an action, in which case the newly generated Learning State is stored as a child of the current one to allow for recursively learning actions (lines 5-8). Since subactions have their own distinct representation, they can also be used outside of the action in which they were learned.

When the learning process begins, the only information that the Goal Manager has about the action that is being learned is the goal that describes it. The Goal Manager knows the name of this goal as well as the number of arguments it has and their values. The values of the predicate which describes the action are used in the execution of its action steps. The reference resolution process described in Section 3.3 guarantees that when these arguments correspond to entities in the real world the same entity will always have the same value. The steps of the action that is being learned are built from the actions that the agent performs after the learning process has begun.

Table 1. The Learn Interactions module takes in a goal and creates a new Learning State when initiating a new action learning phase, or creates a new action when completing action learning.

```
function learnInteraction(goal):
1     action ← analyzeArguments(goal)
2     ADB ← actionDatabase
3     if action = startLearning
4        newLearningState ← LearningState(action, args)
5        if currentLearningState ≠ null
6           parent(newLearningState) ← currentLearningState
7        currentLearningState ← newLearningState
8     else if action = endLearning
9        newAction ← generateActionDB(currentLearningState)
10       add(ADB, newAction)
11       actionQueue = getActionQueue(currentLearningState)
12       if parent(currentLearningState) ≠ null
13          currentLearningState ← parent(currentLearningState)
14          add(actionQueue, newAction)
15    else
16       add(actionQueue action)
```

While the agent is learning, following Table 2 and Table 3, the learning framework pops actions off the Learning State ActionQueue. For each popped action a new action step is created with the required agent (lines 2-6, Table 2). Because each action step may be executed by a different actor, the value of its actor is matched against the input arguments of the learned action. The arguments of the popped action are checked and their values are aligned with the input arguments of the action corresponding to the current Learning State (lines 10-12, Table 3). In subsequent executions of the learned action the action step arguments whose values matched the top level action's arguments' values during learning will be replaced with the values of the top level action's corresponding arguments (lines 2-7, Table 3). Arguments that did not match will always use the values they had during learning (line 2-14, Table 3).

The learning process is completed in a similar fashion to how it is initiated, a goal indicating the ending of the process is submitted to the Goal Manager. For example if the robot is learning 'pick up the knife', then the goal is *endActionLearning(pickUp(self, object_0))*. When the process is complete the new action is added to the database and is immediately available for execution (lines 10-11, Table 1). Once the action is generated, the Goal Manager checks to see if the parent of the current Learning State is empty. If a parent Learning State exists, the Goal Manager restores the prior Learning State and adds the newly learned action into the sequence of actions in parent Learning State. After the unknown subaction is learned, the agent can continue learning the original action (lines 12-17, Table 1).

Table 2. The addActionStep procedure takes in a Learning State creates and adds a new action step.

```
function addActionStep(ls):
  while learning:
    actionQueue = getActionQueue(ls)
    if ¬ empty(actionQueue)
      actionStep ← pop(actionQueue)
      newActionStep ← new ActionStep()
      arg ← getAgent(actionStep)
      addArgument(ls, newActionStep, arg)
      append(newActionStep, getName(ActionStep))
      forall arg in getArguments(action)
        addArgument(ls, newActionStep, arg)
      addActionStep(ls, newActionStep)
```

4. System Demonstration

We can demonstrate the operation of interaction learning with an example that involves picking up and passing a knife. This shows how learning an interaction (1) requires recursively learning subactions, (2) involves references to external objects, and (3) involves the actions of multiple actors (video at `https://youtu.be/t83OovAxcCY`). Initially, the instructor asks the agent to pass the knife.

Human: Pass me the knife
Robot: Sorry, I don't know how to do that

The robot does not have "pass" in its lexicon, but is able to infer that it is a verb, and that the sentence is a command, from the syntax of the rest of the words in the utterance Scheutz et al. (2017). The resulting semantic representation for the utterance is *INSTRUCT(human,self,pass(self, human, object_0))*.

This semantic representation is used to generate the goal *pass(self, human, object_0)* which is sent to the Goal Manager, but execution fails (and is stated by the robot) because the robot has no procedural knowledge of the "pass" action.

Human: I will teach you how to pass me the knife.
Robot: Ok.

The goal derived from the human's utterance *startActionLearning(pass(self, instructor, object_0))* is passed to Goal Manager. Here, *object_0* represents *knife* as described in Section 3.3, initiating the learning process as described in Table 1. Because the Goal Manager is not currently learning an action, it starts the learning phase, extracts the first argument of the *startActionLearning* predicate, *pass(self, instructor, object_0)*, and creates a new Learning State for the action "pass" with the arguments: *self, instructor, object_0*. The agent then acknowledges that it has started learning.

Human: Pick up the knife.
Robot: Sorry, I do not know how to do that.

Table 3. The addArguments function takes in a Learning State, an action step, and the argument to be added to the action step. The method then adds the argument or argument value to the Learning State and action step based on the Learning State input arguments.

```
function addArguments(ls, actionStep, arg):
    arguments ← getArguments(ls)
    if contains(arguments, getName(args))
        argument ← get(arguments, getName(arg))
        if getValue(argument) $\neq$ getValue(arg)
            modifyName(arg)
            addArgument(ls, arg)
            append(actionStep, arg)
        else
            inputArgs = getInputArgs(ls)
            if contains(inputArgs, getValue(arg))
                addArgument(ls, arg)
                append(actionStep, getName(arg))
            else
                append(actionStep, getValue(arg))
```

The Goal Manager receives the goal *pickUp(self, object_0)*, but the robot does not have knowledge of this action, which it indicates to the human.

Human: I will teach you how to pick up the knife.
Robot: Ok.

In order to continue learning the "pass" action, the instructor now needs to teach the robot how to pick up the knife, initiating a sub-learning phase for "pickUp", which generates the semantics *startActionLearning(pickUp(self, object_0))*.

The Goal Manager then creates a new Learning State for *pickUp(self, object_0)* and the parent Learning State for "pass" is stored within the new Learning State. Once again, the agent acknowledges that it has started learning.

Human: First, find it
Robot: Ok

To ensure the agent knows what it will be manipulating, the teacher instructs the agent to find the knife. This results in the submission of the goal *findObject(self, object_0)* to the Goal Manager (the reference to "it" is correctly resolved to "object_0"). The Goal Manager then selects and executes the "find" action. Once complete, the action is added to the ActionQueue in the current Learning State causing the Goal Manager to create an action step by considering the actor tasked with executing the action and its arguments. The Goal Manager pops an action off the Learning State ActionQueue. The action contains the bounded arguments for the *findObject* action: *?actor: self* and *?object: object_0*. To let the agent know who is executing the different parts of the action, the action name is added to the beginning of the action step followed by the actor "findObject ?actor".

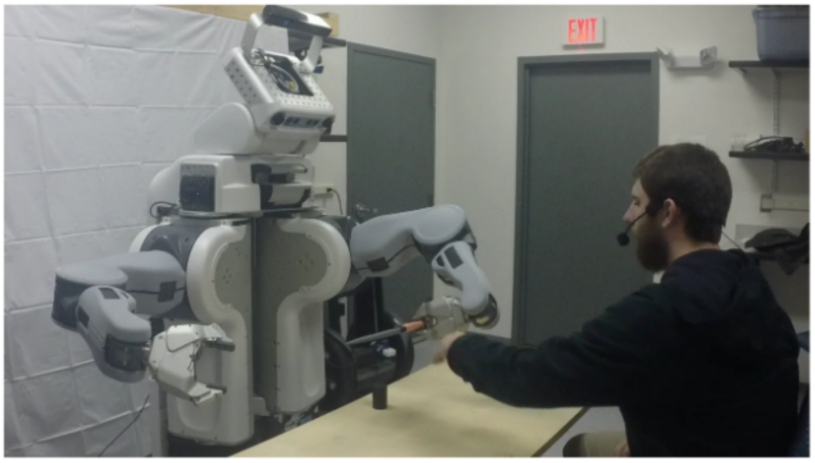

Figure 2. The PR2 observing the human grabbing the knife.

The Goal Manager proceeds to check if the *?object* argument is already stored in the Learning State. Since it is not, the Goal Manager checks and notices it matches the value found in the argument of the *pickUp* predicate. Now, the Goal Manager adds it to the current arguments in the Learning State and adds it to the action step. Finally, the Goal Manager adds the *findObject* action step to the "pickUp" Learning State.

Learning State	
Name:	pickUp
Args:	**?actor ?object**
Steps:	**findObject ?actor ?object**

Human:	Grasp the knife.
Robot:	Ok.

Next, the teacher tells the robot to grasp the knife. The system is able to determine, through reference resolution, that the knife in this context is the same one from the previous action. It then proceeds to execute the *graspObject* action, and adds it to the Learning State. The Goal Manager creates a new action step and, since both values self and knife are already in the Learning State's current arguments, the *?actor* and *?object* are added to the action step.

Learning State	
Name:	pickUp
Args:	?actor ?object
Steps:	findObject ?actor ?object
	graspObject ?actor ?object

Human:	Move it up.
Robot:	Ok

The teacher instructs the agent to move the object up.

Learning State	
Name:	pickUp
Args:	?actor ?object
Steps:	findObject ?actor ?object
	graspObject ?actor ?object
	moveObject ?actor ?object up

Human: That is how you pick up the knife.
Robot: Ok.

Finally, the teacher indicates the end of learning of "pickUp" by saying, "That is how you pick up the knife" and the agent acknowledges the command. The Goal Manager constructs a new action script from the Learning State with the added effect *did(pickUp(?actor, ?object))*. This script is then inserted into the Action Database and can now be executed any time.

Learning State	
Name:	pickUp
Args:	?actor ?object
Steps:	findObject ?actor ?object
	graspObject ?actor ?object
	moveObject ?actor ?object up
Effects:	**did(pickUp(?actor,?object))**

Because the agent was interrupted during the learning of "pass," the Goal Manager restores the parent Learning State and adds the newly generated "pickUp" action to the Learning State, which then is processed.

Learning State	
Name:	pass
Args:	**?actor ?object**
Steps:	**pickUp ?actor ?object**

Human: Move the knife forward.
Robot: Ok.
Human: Look up.
Robot: Ok.

Now the original learning phase, "pass," continues with the instruction to move the knife forward and then look up. The agent moves the knife forward and looks up, then processes them and adds the action steps to the Learning State ActionStepQueue.

Learning State	
Name:	pass
Args:	?actor ?object
Steps:	pickUp ?actor ?object
	moveObject ?actor ?object forward
	look ?actor up

At this point the robot is holding the knife out in front of its body where the instructor can grab it.

Human: I will grab the knife.
Robot: Ok.
Human: Release the knife.
Robot: Ok.

The teacher tells the agent, "I will grab the knife," resulting in the goal *graspObject(instructor, object_0)*. Unlike the prior goals submitted, the actor is the instructor. When checking the preconditions for "graspObject", Goal Manager detects that the actor is not a DIARC agent, and that the preconditions of the action are not observable. Therefore, the robot assumes the condition is true and continues with the execution sequence. However, since the action is being executed by the human, Goal Manager does not direct the robot to perform the action, but rather continues to the effect checking phase by starting up an observer for the *touching(instructor, object_0)* goal.

Because the action is part of the overall "pass" action, it is added to the Learning State. When processing the action, Goal Manager notices that the *?actor* does not equal *self* and since an *?actor* variable already exists in the Learning State with a different value, it creates and adds a new variable *?var_0* to the Learning State. Goal Manager then adds the *?object* to the action step. Finally, the "releaseObject" action is processed and added to the Learning State.

Learning State	
Name:	pass
Args:	?actor ?object **?var_0**
Steps:	pickUp ?actor ?object
	moveObject ?actor ?object forward
	look ?actor up
	graspObject ?var_0 ?object
	releaseObject ?actor ?object

Human: That is how you pass me the knife.
Robot: Ok.

Now the teacher indicates the end of learning phase by saying, "That is how you pass me the knife." Then Goal Manager constructs a new action script from the Learning State.

Learning State	
Name:	pass
Args:	?actor ?object ?var_0
Steps:	pickUp ?actor ?object
	moveObject ?actor ?object forward
	look ?actor up
	graspObject ?var_0 ?object
	releaseObject ?actor ?object
Effects:	**did(pass(?actor, ?var_0, ?object))**

After the agent has learned the new action and added it to the Action Database, it becomes immediately available for execution. To verify that the agent properly learned the new action, the instructor asks the agent to pass a plate. Changing the knife to a plate demonstrates that the action is not specific to the knife and can be executed on various objects. After picking up the plate and moving it forward, the agent starts an observer for the touching event and waits for the instructor to grab the plate. Once the agent has confirmed the instructor is grasping the plate, it releases the object.

In addition, we have also verified in simulation, without action execution, that the actions learned during this demonstration can be executed by the agent from the perspective of the other agent, effectively using the learned action in reverse, to be handed an object. This simulation environment does not perform any actual perception or manipulation and is aimed at evaluating the core action execution mechanisms. The only addition needed to run this configuration on the physical robot is the addition of more complex observers, but that is beyond the scope of this paper.

5. Discussion and Conclusion

We presented the first interaction learning method by significantly extending past work on one-shot action learning to one-shot multi-agent interaction learning. The trace above shows how to learn about multi-agent interactions with only a single demonstration. Unlike other systems that learn from demonstrations, which focus on individual behaviors or skills and do not take into account who is performing the action, our representation explicitly denotes the agent carrying out each step, letting our system handle tasks with multiple agents. Although we only showed the agent learning a joint action, with some extensions to the dialog capabilities of DIARC, the system can learn social action consisting of joint action collaboration, question answering, and explanations of reasoning.

We introduced a mechanism which allows an agent to observe the actions of other agents and determine if those actions are successful and showed how these observations can be grounded in a cognitive robotic architecture. We also provided the learner with a mechanism that allows it to understand how objects are used across the steps of an interaction and thus which entities in an interaction can vary across instances of that interaction. Understanding what can vary, and when, enables generalization of learned actions to different objects and contexts. In addition, because the task representation has been extended to incorporate the agent associated with each subaction, an agent can partake in the interaction in any role as long as it is able to execute the actions defined for that role in the interaction.

While the current system is still fairly limited in the way it can be instructed, the focus in this paper was not on the natural language side or on learning all actions from scratch, but on the architectural representations and mechanisms enabling interaction learning for agents that are already capable of performing *some actions* and *some observations* with *sufficient natural language understanding* capabilities to be able to understand the teacher's instructions. Before conducting a full-fledged evaluation with different types of interaction learning, the system must be more robust when learning new actions. The learning system needs an additional mechanism to let the teacher provide additional information about conditions and effects, as well as the capability to modify action scripts if there are issues during learning. Future work should allow the teacher to provide feedback to the agent so the action scripts are more robust, expand on the number and types of observers, and extend the instruction understanding capabilities to allow for more free-form teaching, as in the case of recursive instructions.

Acknowledgements

This work has in part been funded by Grant N00014-16-1-2278 from the Office of Naval Research, which is not responsible for the views and conclusions of the authors contained herein.

References

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

Akgun, B., Cakmak, M., Yoo, J. W., & Thomaz, A. L. (2012b). Trajectories and keyframes for kinesthetic teaching: A human-robot interaction perspective. *Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction* (pp. 391–398). Boston, MA: ACM.

Allen, J., Chambers, N., Ferguson, G., Galescu, L., Jung, H., Swift, M., & Taysom, W. (2007). Plow: A collaborative task learning agent. *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence* (pp. 1514–1519). Vancouver, Canada: AAAI Press.

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

Chernova, S., & Veloso, M. (2007). Confidence-based policy learning from demonstration using gaussian mixture models. *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems* (p. 233). Honolulu, HI: ACM.

Dzifcak, J., Scheutz, M., Baral, C., & Schermerhorn, P. (2009). What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. *Proceedings of the 2009 IEEE International Conference on Robotics and Automation* (pp. 4163–4168). Kobe, Japan: IEEE.

Ghalamzan, A. M., Paxton, C., Hager, G. D., & Bascetta, L. (2015). An incremental approach to learning generalizable robot tasks from human demonstration. *Proceedings of the 2015 IEEE International Conference on Robotics and Automation* (pp. 5616–5621). Seattle, WA: IEEE.

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

Mohan, S., & Laird, J. E. (2011). Towards situated, interactive, instructable agents in a cognitive architecture. *Proceedings of the AAAI 2011 Fall Symposium: Advances in Cognitive Systems*. Arlington, VA: AAAI Press.

Mohseni-Kabir, A., Chernova, S., & Rich, C. (2014). Collaborative learning of hierarchical task networks from demonstration and instruction. *Proceedings of the AAAI 2014 Fall Symposium: Artificial Intelligence for Human-Robot Interaction*. Arlington, VA: AAAI Press.

Nicolescu, M. N., & Mataric, M. J. (2001). Learning and interacting in human-robot domains. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, *31*, 419–430.

Nicolescu, M. N., & Mataric, M. J. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 241–248). Melbourne, Australia: ACM.

Rybski, P. E., Yoon, K., Stolarz, J., & Veloso, M. M. (2007). Interactive robot task training through dialog and demonstration. *Proceedings of the Second ACM/IEEE International Conference on Human-robot Interaction* (pp. 49–56). Arlington, VA: ACM.

Scheutz, M., Krause, E., Oosterveld, B., Frasca, T., & Platt, R. (2017). Spoken instruction-based one-shot object and action learning in a cognitive robotic architecture. *Proceedings of the Sixteenth International Conference on Autonomous Agents and Multiagent Systems* (pp. 1378–1386). SÃčo Paulo, Brazil: ACM.

Scheutz, M., Krause, E., Oosterveld, B., Frasca, T., & Platt, R. (2018). Recursive spoken instruction-based one-shot object and action learning. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (pp. 5354–5358). Stockholm, Sweden: IJCAI.

Scheutz, M., Williams, T., Krause, E., Oosterveld, B., Sarathy, V., & Frasca, T. (in press). An overview of the distributed integrated cognition affect and reflection DIARC architecture. In M. I. A. Ferreira, J. S. Sequeira, & R. Ventura (Eds.), *Cognitive architectures*. Cham, Switzerland: Springer.

Williams, T., Acharya, S., Schreitter, S., & Scheutz, M. (2016). Situated open world reference resolution for human-robot dialogue. *Proceedings of the Eleventh ACM/IEEE Conference on Human-Robot Interaction* (pp. 311–318). Christchurch, New Zealand: IEEE.

Williams, T., & Scheutz, M. (2017). Reference resolution in robotics: A givenness hierarchy theoretic approach. In J. Gundel & B. Abbott (Eds.), *The Oxford handbook of reference*. Oxford, England: Oxford University Press.