Program Synthesis by Examples for Object Repositioning Tasks

Ashley Feniello and Hao Dang and Stan Birchfield
Microsoft Robotics, Redmond, WA 98052
{ashleyf, haodang, stanleyb}@microsoft.com

***Abstract*— We deal with the problem of synthesizing human-readable computer programs for robotic object repositioning tasks based on human demonstrations. A domain specific language (DSL) is introduced for generic robotic object repositioning tasks. A learning algorithm is designed and implemented to synthesize a generalized program in this DSL based on human demonstrations. Experiments show that a great variety of object repositioning tasks can be programmed through human demonstrations using our system. Synthesized programs can be executed in simulation for verification and refinement and then on a robot to accomplish the corresponding learned tasks in the physical world.**

I. Introduction

Commercially viable robots today require explicit programming by a robotics expert in order to accomplish even the most basic manipulation task. The tediousness and difficulty of such programming greatly reduce the flexibility of robots, making it cost-prohibitive for manipulators to be used for anything other than highly-repetitive tasks over long periods of time. The ability for a non-expert to train a robot on the fly — without explicit programming — would open up a host of new application domains.

To overcome this limitation, much attention in the research community has been devoted over the past decade to *learning by demonstration* [4]. In this paradigm, a robot is taught not by explicit programming but rather by being guided through the task by a human, or alternatively by watching a human perform the task. Learning by demonstration is a broad area of research, covering low-level sensorimotor learning [5], [24], [1], [19], learning grounded symbols from continuous data [16], [18], [10], learning trajectories from multiple demonstrations [9], segmenting continuous trajectories into discrete tasks, skills or keyframes [17], [21], [3], the role of the human teacher [7], [6], [22], [8], learning object affordances [20], and motion planning [23].

In this paper we propose a learning by demonstration approach using *visuospatial skill learning (VSL)* [2], in which the robot learns to perform object repositioning tasks (e.g. sorting, kitting, or packaging) through human demonstrations. Figure 1 illustrates the components of our system. A camera detects the objects in the workspace of the robot, along with their visual properties. A tablet serves as the user interface on which the teacher demonstrates a repositioning task in a simulated environment. Based on one or more demonstrations, a synthesized program is generated which can be used to accomplish the task in the workspace even with an arrangement of objects different from the training inputs. Our formalism of object repositioning tasks is generic

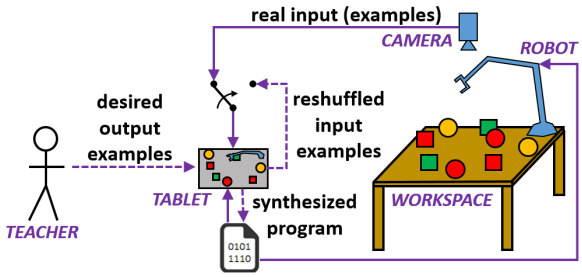


Fig. 1: Overview of our learning-by-demonstration system. Arrows indicate control flow. During training, the teacher interactively provides examples of desired output while viewing the behavior of the synthesized program via interface with the tablet. Input examples either come from the camera or are reshuffled by the tablet. After training, the synthesized program is used to drive the robot to physically manipulate objects in the workspace. (Dashed lines are used only in training.)

and can be used to specify a wide range of object repositioning tasks. Programs synthesized by our system are human-readable, enabling the user to validate the logic learned and to make necessary changes to the learned program.

Specifically, our contributions are as follows:

- A stack-based concatenative domain-specific language, inspired by [15], [13], for describing object repositioning tasks that is flexible enough to handle a wide variety of sorting, kitting, and packaging tasks.
- A learning algorithm that infers the intent of the user from one or more demonstrations by searching for the simplest human-readable program that achieves the same behavior as the demonstrated behavior on the training inputs. The program can then be applied to novel configurations of objects, some of which the system has never seen.
- A tablet-based system that facilitates rapid training by allowing multiple demonstrations to be performed, and the resulting learned program validated, before applying to the robot.

Our approach is similar to that of [11], which utilizes a formal context-free grammar (CFG) containing basic pick, place, move, and (un)grasp commands to learn assembly tasks from demonstrations. In contrast, our system is aimed at repositioning tasks, and our grammar includes a richer set of commands to enable more abstract reasoning and generalization about the objects in the scene. Our approach

is also similar to that of [10], which performs grounding of discrete concepts from continuous data for the purpose of generalizing to new objects and new scenarios. However, although we do perform some grounding (e.g. by thresholding the aspect ratio of objects to determine their shape), this is not the primary focus of our system. Rather, our aim is higher-level reasoning about the human teacher's intent in sorting, kitting, and packaging tasks. Compared with the work of [2], our approach is able to operate with noisy sensor data and ambiguous demonstrations, with varying numbers of objects that may or may not have been seen before.

II. Language for Object Repositioning Tasks

Our domain-specific language (DSL) for reasoning about object repositioning tasks is a concatenative, stack-based language that is able to concisely represent complex programs. Composing programs from permutations of well-formed pieces by simple concatenation (no formal function arguments to manage) has proven quite convenient for program synthesis, and a stack machine execution engine is simple and performs well. Moreover, its implementation in F# is itself extremely concise, making the approach extensible and flexible. Here we describe the syntax and semantics of the base DSL, followed by the operators specific to object repositioning.

A. DSL syntax

The syntax of the DSL is provided in Figure 2. The $world$ consists of a $state$ and an evaluation $stack$ (used by the program during execution). The $state$ may consist of any $value$, but for object repositioning tasks it captures the objects in the workspace, along with their properties. The properties of each object are stored in a rec, which consists of zero or more $(string, value)$ pairs, one per property. The $stack$ is a list of zero or more $value$s, where each value can be either a $bool$, $float$, $string$, or op (operator), or a $list$ or rec to enable complex data to be stored. A $program$ is a list of zero or more $value$s (generally operators or their parameters) that either modify the world directly (through a $primitive\ op$) or execute a $list$ of ops (through a $compound\ op$).[1]

B. DSL semantics

The semantics of some DSL operators are provided in Figure 3, where for simplicity we show operators that manipulate only the $stack$ while ignoring the $state$. It is important to note that compound operators need not be named as they are here, but can also exist as *anonymous* lists of operators, which are simply treated as values on the stack. Several higher order operators consume and execute such values. For example, predicate expressions used to **filter** a list of objects, selector expressions used to group objects by property (**takebygroup**), and conditionals executed by **if**.

The semantics of DSL execution are shown in Figure 4. The meta-operator **step** takes a world (state ψ, stack $[\sigma^*]$) and program $\pi = [v\ \pi^*]$ and applies the value at the front of the program: If the value v is a primitive operator (a world-to-world function), then it is applied to the current world; if it is a compound operator, then it is expanded within the program itself; otherwise it is a literal value and is simply pushed onto the stack. In any case **step** returns the new world and program. The meta-operator **run** runs a program on a world, returning the new world once the program is empty, or otherwise recursively calling itself on the result of **step**.

[1]Compound operators are also called "quotations". An operator that take a quotation as one of its inputs is known as a "combinator".

$$
\begin{aligned}
value &:= bool \mid float \mid string \mid list \mid rec \mid op \\
list &:= [value^*] \\
rec &:= \{(string, value)^*\} \\
op &:= primitive \mid compound \\
primitive &:= world \rightarrow world \\
compound &:= list \\
world &:= (state, stack) \\
state &:= value \\
stack &:= list \\
program &:= [value^*]
\end{aligned}
$$

Fig. 2: Domain-specific language (DSL) syntax, in our own simplified Backus-Naur Form. Asterisk denotes Kleene closure, brackets denote an ordered list, braces denote a set (i.e., an unordered list), and parentheses denote a tuple.

$primitive$:

$$
\begin{aligned}
\textbf{swap}: &\ [\sigma^*\ v_1\ v_2] \mapsto [\sigma^*\ v_2\ v_1] \\
\textbf{drop}: &\ [\sigma^*\ v] \mapsto [\sigma^*] \\
\textbf{dup}: &\ [\sigma^*\ v] \mapsto [\sigma^*\ v\ v] \\
\textbf{dip}: &\ [\sigma^*\ v\ [o]] \mapsto [o([\sigma^*])\ v] \\
\textbf{if}: &\ [\sigma^*\ b\ [o_1]\ [o_2]] \mapsto b\ ?\ o_1([\sigma^*]) : o_2([\sigma^*]) \\
\textbf{filter}: &\ [\sigma^*\ \ell\ [o]] \mapsto [\sigma^*\ \ell'] \\
&\ \text{s.t. } \ell' \subseteq \ell \text{ and } \forall e \in \ell', o(e) = \text{TRUE} \\
\textbf{eval}: &\ [\sigma^*\ [o]] \mapsto [o([\sigma^*])] \\
\textbf{take}: &\ [\sigma^*\ [\ell^*\ \cdots\ \ell_2\ \ell_1]\ n] \mapsto [\sigma^*\ [\ell_n\ \cdots\ \ell_2\ \ell_1]] \\
\textbf{fetch}: &\ [\sigma^*\ r\ p] \mapsto [\sigma^*\ v] \text{ where } (p, v) \in r
\end{aligned}
$$

$compound$:

$$
\begin{aligned}
\textbf{nip}: &\ [[\textbf{drop}]\ \textbf{dip}] \\
\textbf{over}: &\ [[\textbf{dup}]\ \textbf{dip}\ \textbf{swap}] \\
\textbf{keep}: &\ [\textbf{over}\ [\textbf{eval}]\ \textbf{dip}] \\
\textbf{bi}: &\ [[\textbf{keep}]\ \textbf{dip}\ \textbf{eval}]
\end{aligned}
$$

Fig. 3: Semantics of some DSL operators. For the primitive operators, only the input and output stack are shown, since these particular operators ignore the state. σ^* indicates the rest of the stack that is (in most cases) unaffected by the operator.

C. Object repositioning syntax and semantics

For object repositioning, the $state$ is initially populated with the objects and their properties, and the operators either

$$
\begin{aligned}
\textbf{step}: \quad & (\psi, [\sigma^*], [v\ \pi^*]) \mapsto \\
& v \in \mathit{primitive} : (\psi, v([\sigma^*]), [\pi^*]) \\
& v = [v^*] \in \mathit{compound} : (\psi, [\sigma^*], [v^*\ \pi^*]) \\
& \text{otherwise} : (\psi, [\sigma^*\ v], [\pi^*]) \\
\textbf{run}: \quad & (\psi, [\sigma^*], \pi) \mapsto \\
& \pi = [] : \ \mathbf{return}(\psi, [\sigma^*]) \\
& \text{otherwise} : \ \mathbf{run}(\mathbf{step}(\psi, [\sigma^*], []))
\end{aligned}
$$

Fig. 4: Semantics of DSL execution.

read or modify these properties, or they compute intermediate values. Although in theory it is not necessary to maintain the $state$ separately from the $stack$, in practice doing so avoids "stack shuffling" and greatly simplifies the formulation and resulting programs.

Repositioning an object is achieved by compound operators such as **moveall**, which sets the x, y and θ properties contained in the rec of the objects in a list. When a program is played back, each **moveall** operator causes either the objects to move on the tablet (in the case of simulation) or the robot to pick up the objects at the previous coordinates and place them at the new coordinates (in the case of physical manipulation). To determine which objects to move, the program filters on the properties. For example, the following program moves all the red objects to the pose $(x, y, \theta) = (13, 27, 1.57\ rad)$:

$$\mathbf{things}\ [\text{'color} \ @\ \text{'red} =]\ \mathbf{filter}\ 13\ 27\ 1.57\ \mathbf{moveall}$$

where **things** places the list of objects from the $state$ onto the $stack$, and @ is shorthand for **fetch**. Note that object properties are either discrete (e.g., the color as shown here) or continuous (e.g., the location or aspect ratio).

Other commands in our DSL include **between**, which tests that a value is within a certain range; **bi**, which applies two operators, leaving both results on the stack; **moveone**, which moves a single object to a specific (x, y, θ) position; **take**, in the form *n* **take**, which takes the first n-number of elements from a list, **takeone**, which is just $[1\ \mathbf{take}]$; **takebygroup**, which groups the objects in a list by a certain property; and **distribute** which consumes a list of objects along with a list of poses and distributes the objects across the set of poses. In total, our DSL includes approximately 50 operators divided fairly evenly between primitive and compound. Of these, only 14 are possibilities in final learned programs, with the others being used solely for composing other operators. Object properties include x, y, θ, dominant color, area, width, height, and aspect ratio.

In designing a language for any domain it is important to recognize that an inherent tradeoff exists between its expressiveness and its complexity. That is, increasing the number and types of commands enables the language to express more interesting concepts but also leads to a combinatorial explosion in the number of possibilities to be explored [12]. Nevertheless, as has been pointed out by [14], the tradeoff is not always so simple, because increasing expressiveness can also lead to shorter programs, thus facilitating efficient search. As a result, it is important to strike the right balance between the two. We believe our DSL achieves such a reasonable balance and, moreover, it is extremely easy to extend with new commands as needed.

III. Learning Programs from Examples

Program synthesis is an interesting and powerful approach to learning by demonstration. The idea is to represent the structure of the search space as a programming language grammar. A task is demonstrated one or more times by an operator, and a program is synthesized to carry out the task. The goal is to find a program that is consistent with the example(s) given and that generalizes correctly to future inputs, even though it may have been given an incomplete specification. Because the resulting program is often human readable, it can be verified and edited.

A. Searching for filters

A primary facility available to the program synthesizer is that of finding appropriate filter expressions. Given a set of objects and a particular subset (e.g., those objects that were moved to a particular location, or distributed across locations), we want to generate filter expressions to separate them based upon their properties as determined by the perception system. First a base set of filters is produced. For discrete values this is simply a set of program fragments for each value in the form:

$$[\text{'}\mathit{property}\ @\ \mathit{value} =]$$

which means "Select objects where *property* is *value*." For continuous values program fragments are generated to find particular ranges of property values:

$$[\text{'}\mathit{property}\ @\ \mathit{min}\ \mathit{max}\ \mathbf{between}]$$

which means "Select objects where *property* is between *min* and *max*." In addition, the negation of each of these:

$$[\mathit{predicate}\ \mathbf{not}]$$

is generated. The base filters are seeded with the simple min/max for each property within the subset of objects, as well as with ranges from clusterings across the superset of objects, thus capturing disjoint ranges.

If one of the base filters happens to select the subset of objects correctly, then we are done. In more complex situations, however, it is necessary to continue the exploration of the search space by combining filters, creating conjunctions, disjunctions, and negations in the forms:

$$[[\mathit{filter}_1]\ [\mathit{filter}_2]\ \mathbf{bi\ and}]$$

$$[[\mathit{filter}_1]\ [\mathit{filter}_2]\ \mathbf{bi\ or}]$$

$$[[\mathit{filter}_1][\mathbf{not}\ \mathit{filter}_2]\ \mathbf{bi\ and}]$$

To manage the search, filters are scored according to the number of false positives and false negatives. Those selecting the correct objects but including incorrect objects

are combined with the negation of filters selecting only incorrect objects. Similarly, pairs of filters selecting only correct objects are logically or'd together. Filters that select at least some correct objects but also some incorrect objects are used as the basis for additional expressions in the hopes that the filter is correct, and the extra objects were due to simply taking too many. Finally, filters that at least select more correct than incorrect are generated. This process is iterated several times, feeding the synthesized filters back through as the basis for additional synthesis and terminating once a set of correct filters is found, up to a fixed number of iterations. The result is potentially very complex expressions that satisfy the selection.

B. Grouping

In addition to subsets of objects selected by filter expressions, subsets consisting of groupings by discrete property values are included and may form the basis of sorting and arranging tasks. For example, "Distribute two of each color to these three bins."

$$\textbf{things}\ [\text{'color}\ @]\ 2\ \textbf{takebygroup}$$
$$[[x_1\ y_1\ \theta_1]\ [x_2\ y_2\ \theta_2]\ [x_3\ y_3\ \theta_3]]\ \textbf{distribute}$$

Essentially, objects may be grouped by a discrete property and then groups therein may be restricted to a maximum size. This allows for very general task descriptions depending on relationships between object properties rather than on particular property values.

C. Synthesis Process

The filtering and grouping process just discussed does an excellent job of constructing expressions to select reasonable candidate sets of objects to be manipulated. These are then consumed by the synthesizers which in turn produce permutations of interesting repositioning actions to be applied. For example, moving the selected objects to the centroids of destination location clusters, distributing the objects across permutations of intervals of the destination locations, etc. The result is a large number of candidate programs (hundreds of thousands) evaluated in a short period of time (several seconds).

D. Evaluating programs

Examples are essentially input/output pairs. Candidate programs are executed against inputs and the resulting actions are compared with demonstrated expected outputs. Quite often multiple candidates produce similarly high ranking outputs, thus requiring programs to be further ranked by a measure of simplicity. True to Occam's Razor, we find that the simplest programs indeed are often the best and most general solutions. To favor simplicity, programs are scored by the weighted number of nodes in an abstract syntax tree (AST). Most nodes receive a weight of 1, but a few primitive and predefined compounds are noticeably more complex (and therefore less desirable), and are therefore assigned a heigher weight as appropriate. Adjusted weights allow for domain-specific rank biasing favoring certain kinds of programs.

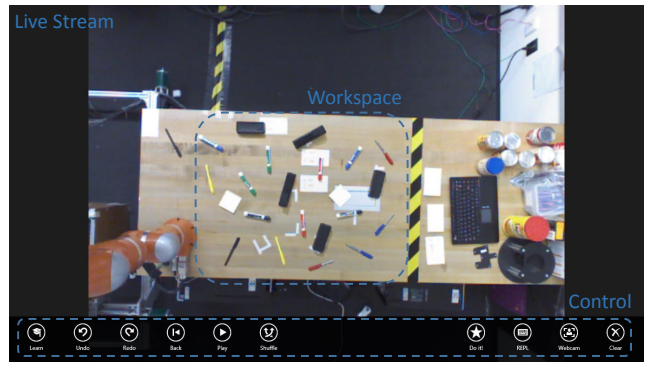


Fig. 5: A Windows application running on a Microsoft Surface tablet as the user interface of interactive teaching.

If computation were not an issue, we would exhaustively search the space of all possible programs, and the program with the lowest score would win. Obviously, this is not possible, because the size of the space is exponential in the number of symbols in the program. Even with just 50 discrete operators, the space of all programs containing 10 operators is $O(50^{10})$. To handle combinatorial explosion, we must conduct a highly directed search.

If the highest ranking program exhibits behavior entirely consistent with all to the examples provided then the process is complete. However, it is often the case that the user has in fact demonstrated what is essentially multiple distinct actions, causing multiple distinct and relatively high ranking programs to be generated. The system discovers this by iteratively exploring combinations of program fragments.

E. System Implementation

A Windows application running on a Surface tablet serves as a teaching interface, shown in Figure 5. Initially, the human teacher places some objects on the workspace table and takes a picture with an overhead camera; the objects are segmented, and their feature properties are calculated using image processing. The teacher then interactively trains the system by presenting input / output pairs by repositioning object sprites on the tablet. When a new scene is needed, the teacher either presses a button on the tablet to reshuffle existing inputs, or physically rearranges objects in the workspace and takes a new picture. The input / output pairs are iteratively fed to the program synthesizer, which generates a program that conforms as best as possible to the examples given. After the first demonstration, the currently learned program is always available for the human teacher to visualize its learning results on new inputs via the tablet in order to facilitate rapid iterative teaching, or to drive the robot to physically interact with the world if the teacher is satisfied. On the tablet, the teacher can also view the program itself at any time to verify its correctness. Once the program is learned and confirmed by the teacher, it can be saved and executed in the workspace.

IV. Experimental Results

In this section we describe our experimental setup, the object repositioning tasks demonstrated, and the results of a robot learning and performing these demonstrated tasks.

A. Experimental Setup

We chose tabletop object repositioning tasks as our experimental scenario, in which all the objects of repositioning tasks were placed on a tabletop. The robot arm we used was a 7 degree-of-freedom Kuka lightweight robot arm, LBR 4+. A 3-finger adaptive Robotiq gripper was attached to the end of the robot arm as the end effector. Due to the fact that the objects involved in our experiments are relatively simple in shape, only pinch grasps were used in our system to pick up objects. All visual sensing was performed by an overhead camera which was installed above the tabletop to provide a top-view image of the workspace.

Due to limited space we only show a sampling of some of the experiments that we have run, to try to convey the types of problems that the system can handle. These experiments are described in the following subsections. Figure 6 shows snapshots corresponding to six experiments, while Figure 7 shows the synthesized program of each of these experiments.

B. Sorting Tasks

The first two experiments involve sorting tasks, in which objects that are different from each other are separated based on their feature properties.

Reorganizing Go stones In this experiment, a human teacher demonstrated putting Go stones into two different bowls, as in a cleanup scenario. In the training process, the teacher dragged the white sprites to one location and the black sprites to a different location. The synthesizer quickly learned a filter that separates objects based on their dominant color, and it then generated a correct program to put away the Go stones into different bowls based on their colors.

Reorganizing office supplies In this experiment the intention was to sort office supplies by placing markers, pens, tape, and an eraser into separate bins of an office supply tray. After a single demonstration the algorithm discovered that the aspect ratio of different objects was associated with the task in separating markers, pens, tapes, and erasers. The algorithm synthesized a program to move two markers to one compartment of the tray, two pens to another compartment, one roll of tape to another, and one eraser to another. The fact that the teacher did not move all markers or all pens indicated that the intention was not to move all of those items, but rather that the number was important. If, in addition, color was important, then additional demonstrations could have been used to indicate that fact.

C. Kitting Tasks

The next two experiments invole kitting tasks, in which different objects are grouped together as a single unit. In contrast to sorting tasks, kitting tasks do not require separating different objects from each other while organizing them.

Preparing a drawing kit In this experiment, a human teacher demonstrated collecting one red and one green marker and preparing them as a drawing kit. The teacher therefore placed one of each marker in each of two bins. From the training process, the algorithm learned the correct combination of markers for the intended drawing kit. Because the colors were the same for both bins, the program learned that color was important.

Preparing an office supply kit In this experiment, a human teacher demonstrated a much more complex task, namely, to place one of each color (among the colors available) in the bin. In the demonstration the teacher moved one red, one green, one blue, and one black marker to the bin, leaving a red marker. After another demonstration with a different initial input, the synthesizer learned the correct program, which the system was then able to apply to any number of any colored markers. Here the robot moves one green and one blue marker, leaving the other green and blue markers untouched.

D. Packaging Tasks

The final two experiments involve packaging task, in which a set of objects are placed at specific locations in a given container. Packaging tasks share similar logic with sorting and kitting tasks but are more industrial oriented.

Packaging a tennis ball canister In this experiment, a human teacher demonstrated placing 3 tennis balls into each of 2 individual tennis ball canisters. The synthesizer learned a program to move exactly 3 balls to each canister. Note the difference with the sorting tasks, in that here the objects are distributed across multiple locations.

Packaging a router box In this experiment, a human teacher demonstrated packaging a box using components of an off-the-shelf wireless router. The synthesizer learned a program to place the router, ethernet cord, power cord, and additional piece into their correct destinations, despite the initial positions of the objects.

E. Progression of Program Synthesis in Novel Scenes

A program is synthesized with one or a given set of static training scenes. The correctness of the synthesized program is validated based on the demonstrated scenes. However, it is likely that the training scenes do not include all the potential objects that could be present in the training scenes. Thus, although the synthesized program is consistent with the given demonstrations, the logic behind the demonstrations may not be correctly interpreted in the programs. Under these circumstances, it is necessary to provide further demonstrations to train the robot such that the actual intent of the human teacher can be discovered. In this experiment, we show how our system performs when it is trained on an incomplete set of objects and how it learned the correct program with two demonstrations.

The actual task we tried to teach was to separate solid black pens from other office supplies. Figure 8 shows the two consecutive demonstrations and the synthesized programs after each of the two demonstrations. In the first training

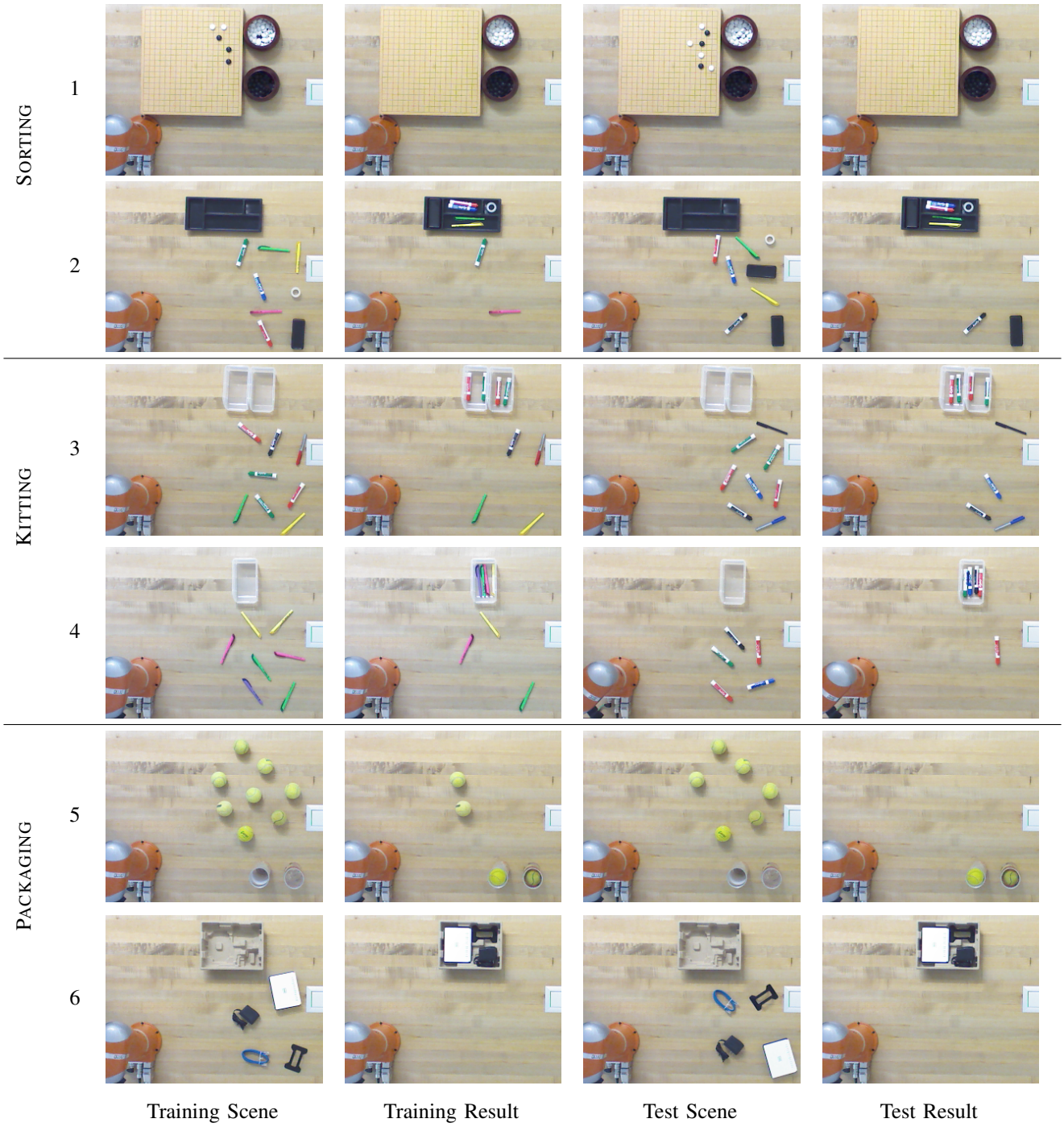


Fig. 6: Snapshots of the six experiments. For each experiment we show the initial and final conditions of the first demonstration, along with the initial and final conditions of a run in which the synthesized program was played on a scene that had never been seen. Several experiments required a single demonstration, while others required two demonstrations; space constraints do not permit other demonstrations and runs to be shown.

#	Synthesized Programs
1	things ['color @ 'white =] filter 561.67 350.67 1.36 moveall things ['color @ 'black =] filter 559.00 472.67 0.87 moveall
2	things ['color @ 'black =] filter 515.00 303.00 -1.57 moveone things [['aspect @ 0.78 >] ['width @ 117.80 139.65 between] bi and] filter 631.50 297.50 3.12 moveone things ['width @ 5.80 6.93 between] filter 2 take 581.00 291.50 0.07 moveall things ['width @ 7.03 8.72 between] filter 2 take 584.10 328.73 moveall
3	things ['color @ 'red =] filter 2 take [[656.25 289.00 1.71] [602.75 287.25 1.69]] distribute things ['color @ 'green =] filter 2 take [656.25 289.00 1.71] [602.75 287.25 1.69]] distribute
4	things ['color @] 1.00 takebygroup 637.50 314.50 1.55 moveall
5	things ['color @] 6.00 takebygroup 6.00 take [[715.33 542.33 1.15] [715.33 542.33 1.30] [715.33 542.33 1.51] [647.33 540.33 0.95] [647.33 540.33 0.96] [647.33 540.33 0.36]] distribute
6	things [['color @ 'blue =] ['color @ 'white =] bi or] filter 564.74 312.25 1.69 moveall things ['length @ 30.92 34.33 between] filter 623.53 332.15 0.74 moveall things [['length @ 36.57 42.22 between] ['color @ 'blue = not] bi and] filter 620.11 287.19 0.06 moveall

Fig. 7: Synthesized programs of the six experiments. Snapshots of the experiments are shown in Figure 6.

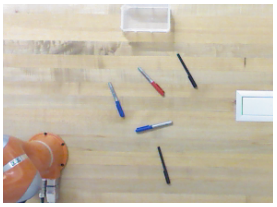
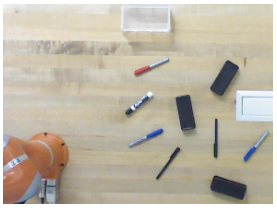
#	Demonstration	Program
1		**Synthesized Program:** things ['color @ 'black =] filter 590.50 272.50 3.14 moveall **Descriptive Interpretation:** move all the things where dominant color is black to location 590.50, 272.50 (3.14)
2		**Synthesized Program:** things [['color @ 'black =] ['aspect @ 0.11 0.52 between not] bi and] filter 588.20 270.00 3.14 moveall **Descriptive Interpretation:** move all the things where dominant color is black and aspect ratio is not between values 0.11 and 0.52 to location 588.20, 270.00 (3.14)

Fig. 8: Program progression. In the first demonstration, only the color was considered. With the second demonstration, both the color and the aspect ratio are considered in the synthesized program. This experiment illustrates how the synthesized program progresses as more demonstrations are given.

scene, only two solid black pens and three markers with different colors were present. With demonstrations on these objects, the synthesized program was "move black objects to the bin". Although this program would suffice to interpret the human teacher's demonstrations on this training scene, it is not fully correct. In the second scene, three black erasers and a black marker were present. The first synthesized program would have mistakenly put the eraser and the black marker into the bin with other solid black pens. However, with a second set of demonstrations on the second scene, the correct program was learned.

V. Conclusion

We have presented our work on synthesizing computer programs for object repositioning tasks based on human demonstrations. We first introduced our design of a domain specific language (DSL) for object repositioning tasks. We then described our learning algorithm which synthesizes a program of this DSL for an object repositioning task. Several experiments were performed to evaluate our work and the experimental results have shown that many object repositioning tasks can be programmed and executed using the proposed language and learning algorithm.

Our domain specific language is designed as a generic stack-based programming language. Although we have shown our language is capable of describing many object repositioning tasks, more ingredients can be added in the future to expand the capability of representing different tasks. One of them is the temporal dependence between actions. By adding temporal dependence into our system, we believe that multi-step tasks can be modeled and learned more effectively.

Beyond repositioning tasks are more generic assembly tasks, e.g. assemble a toy car from individual parts. Although our current work is focused on learning the spatial relationship and association of objects in a repositioning

task, our work can be extended to more complicated and generic assembly tasks. In a generic object assembly task, not only the relative spatial relationship between parts would be learned, but how they come into that relationship should also be modeled and learned, e.g. the mating procedure. We will be extending our work along this direction as our next step.

To scale up our system to an even wider range of objects, many other advanced perception techniques can be exploited, including more complicated feature extraction and classification methods. Distinctive features extracted from objects can provide our learning algorithm with more reliable information to distinguish objects from each other and to facilitate the process of discovering the intent of the teacher.

Finally, to improve our learning-by-demonstration system, we will be looking into other user interface components including program editability and synthesizer efficiency.

REFERENCES

[1] P. Abbeel and A. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2004.

[2] S. R. Ahmadzadeh, P. Kormushev, and D. G. Caldwell. Visuospatial skill learning for object reconfiguration tasks. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 685–691, Nov. 2013.

[3] B. Akgun, M. Cakmak, K. Jiang, and A. L. Thomaz. Keyframe-based learning from demonstration. *International Journal of Social Robotics*, 4(4):343–355, Nov. 2012.

[4] B. D. Argall, S. Chernova, M. Veloso, and B. Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, May 2009.

[5] C. G. Atkeson and S. Schaal. Robot learning from demonstration. In *International Conference on Machine Learning (ICML)*, pages 11–73, 1997.

[6] M. Cakmak. *Guided teaching interactions with robots: Embodied queries and teaching heuristics*. PhD thesis, Georgia Tech, Aug. 2012.

[7] M. Cakmak, C. Chao, and A. L. Thomaz. Designing interactions for robot active learners. *IEEE Transactions on Autonomous Mental Development*, 2(2):108–118, 2010.

[8] S. Calinon and A. Billard. Incremental learning of gestures by imitation in a humanoid robot. In *ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 255–262, Mar. 2007.

[9] S. Calinon, F. Guenter, and A. Billard. On learning, representing and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(2):286–298, Apr. 2007.

[10] C. Chao, M. Cakmak, and A. L. Thomaz. Towards grounding concepts for transfer in goal learning from demonstration. In *IEEE International Conference on Development and Learning (ICDL)*, pages 1–6, Aug. 2011.

[11] N. Dantam, I. A. Essa, and M. Stilman. Linguistic transfer of human assembly tasks to robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 237–242, 2012.

[12] D. Gabelaia, R. Kontchakov, A. Kurucz, F. Wolter, and M. Zakharyaschev. Combining spatial and temporal logics: Expressiveness vs. complexity. *Journal of Artificial Intelligence Research*, 23:167–243, 2005.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 317–330, 2011.

[14] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[15] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 317–328, 2011.

[16] N. Jetchev, T. Lang, and M. Toussaint. Learning grounded relational symbols from continuous data for abstract reasoning. In *ICRA Workshop on Autonomous Learning*, May 2013.

[17] G. Konidaris, S. Kuindersma, R. Grupen, and A. Barto. Robot learning from demonstration by constructing skill trees. *International Journal of Robotics Research*, 31(3):360–375, Mar. 2012.

[18] J. Kulick, M. Toussaint, T. Lang, and M. Lopes. Active learning for teaching a robot grounded relational symbols. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Aug. 2013.

[19] M. Lopes and J. Santos-Victor. Visual learning by imitation with motor representations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 35(3):438–449, June 2005.

[20] L. Montesano, M. Lopes, A. Bernardino, and J. Santos-Victor. Learning object affordances: From sensory-motor coordination to imitation. *IEEE Transactions on Robotics*, 24(1):15–26, 2007.

[21] S. Niekum, S. Chitta, B. Marthi, S. Osentoski, and A. G. Barto. Incremental semantically grounded learning from demonstration. In *Robotics: Science and Systems*, June 2013.

[22] A. L. Pais, B. D. Argall, and A. G. Billard. Assessing interaction dynamics in the context of robot programming by demonstration. *International Journal of Social Robotics*, 5(4):477–490, 2013.

[23] M. Phillips, V. Hwang, S. Chitta, and M. Likhachev. Learning to plan for constrained manipulation from demonstrations. In *Robotics: Science and Systems*, June 2013.

[24] S. Schaal, A. Ijspeert, and A. Billard. Computational approaches to motor learning by imitation. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 358(1431):537–547, 2003.