Program Synthesis by Examples for Object Repositioning Tasks

Ashley Feniello, Hao Dang, and Stan Birchfield

***Abstract*— We address the problem of synthesizing human-readable computer programs for robotic object repositioning tasks based on human demonstrations. A stack-based domain specific language (DSL) is introduced for object repositioning tasks, and a learning algorithm is proposed to synthesize a program in this DSL based on human demonstrations. Once the synthesized program has been learned, it can be rapidly verified and refined in the simulator via further demonstrations if necessary, then finally executed on an actual robot to accomplish the corresponding learned tasks in the physical world. By performing demonstrations on a novel tablet interface, the time required for teaching is greatly reduced compared with using a real robot. Experiments show a variety of object repositioning tasks such as sorting, kitting, and packaging can be programmed using this approach.**

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* [1]. 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 [2], [3], [4], [5], learning grounded symbols from continuous data [6], [7], [8], learning trajectories from multiple demonstrations [9], segmenting continuous trajectories into discrete tasks, skills or keyframes [10], [11], [12], the role of the human teacher [13], [14], [15], [16], learning object affordances [17], and motion planning [18].

In this paper we propose a learning by demonstration approach using *visuospatial skill learning (VSL)* [19], in which the robot learns to perform object repositioning tasks 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. With an additional automated step, programs synthesized by our system are human-readable, enabling the user to validate the logic learned or to introduce new demonstrations to refine the learned program.

All authors are with the Robotics Group of Microsoft Research, Redmond, Washington, USA.
Email: {ashleyf, haodang, stanleyb}@microsoft.com

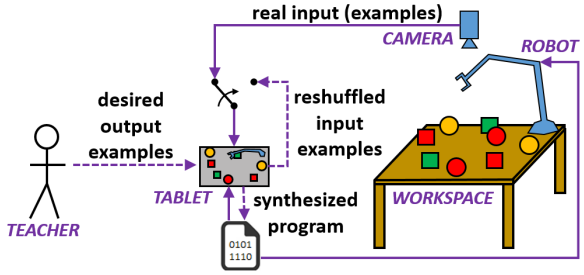


Fig. 1: Overview of our learning-by-demonstration system. Arrows indicate control flow. During training, the teacher interactively provides examples of desired output while viewing the behavior of the synthesized program via interface with the tablet. Input examples either come from the camera or are 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 during training.)

Specifically, our contributions are as follows:

- A stack-based concatenative domain-specific language (DSL), inspired by [20], [21], 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 may never have 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 [22], 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 [8], 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 [19], our approach is able to operate with 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 certain advantages over alternative methods. Moreover, its implementation is itself extremely concise (a few hundred lines in F#), making the approach extensible and flexible. Here we describe the syntax and semantics of the base DSL, followed by the operators that are 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. A $program$ is a list of zero or more $value$s (generally operators or their parameters) that modify the world either directly through a $primitive\ op$ or indirectly through a $compound\ op$, which is a $list$ of ops.[1]

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

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

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

Fig. 3: Semantics of some DSL operators. Only the input and output stack are shown, since these particular operators ignore the state. The top of the stack is on the right, and the symbol σ^* indicates the rest of the stack, which (in most cases) is unaffected by the operator. Shown are an operator o, list ℓ, integer n, value v, bool b, element e of a list, rec r, string s, and so on.

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 v at the front of the program: If the value 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 sometimes called "quotations". An operator that take a quotation as one of its inputs is known as a "combinator".

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

Fig. 4: Semantics of DSL execution.

C. Object repositioning syntax and semantics

For object repositioning, the $state$ is initially populated with the objects and their properties, and the operators either 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 $(x, y, \theta) = (13, 27, 1.57)$:

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

where **things** places the list of objects from the $state$ onto the $stack$, and **@** is shorthand for **fetch**.

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 $[1\ \textbf{take}]$; **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.

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 [23]. Nevertheless, as has been pointed out by [24], 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

Given one or more example demonstrations, the goal of the program synthesizer is to find the simplest program that is consistent with the demonstrations. Simplicity is key to ensuring that the program generalizes correctly to future inputs, even though the synthesizer may have been given an incomplete specification via the finite number of demonstrations. The synthesizer searches through the space of possible programs using the programming language grammar, with each program represented as an abstract syntax tree (AST).

A. Searching for filters

A fundamental step in the search is to find 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), the program synthesizer generates 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:

$$[\text{`}\textit{property}\ @\ \textit{value} =]$$

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

$$[\text{'}\textit{property}\ @\ \textit{min}\ \textit{max}\ \textbf{between}]$$

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

$$[\textit{predicate}\ \textbf{not}]$$

The base filters are seeded with the simple min/max for each property within the subset of objects, as well as with ranges from clusters 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 and disjunctions in the forms:

$$[[\textit{filter}_1]\ [\textit{filter}_2]\ \textbf{bi and}]$$

$$[[\textit{filter}_1]\ [\textit{filter}_2]\ \textbf{bi or}]$$

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 criteria.

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" is represented as

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

Essentially, objects may be grouped by a discrete property, 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 constructs 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, and so forth. 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 some primitive and predefined compounds are noticeably more complex (and therefore less desirable), and are therefore assigned a higher weight as appropriate. Adjusted weights allow for domain-specific rank biasing favoring certain kinds of programs.

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, that is, $O(n^m)$, where n is the number of symbols and m is the length of the program. Even with just 50 operators, the space of all programs containing 10 operators is intractable (approximately 50^{10}). To handle combinatorial explosion, we must conduct a highly directed search.

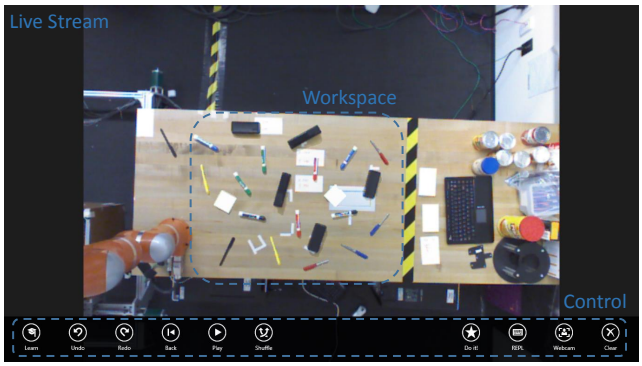


Fig. 5: A Windows application running on a Microsoft Surface tablet as the user interface of interactive teaching.

If the highest ranking program exhibits behavior entirely consistent with all of 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. Perception

Object properties are acquired via automated processing of the image acquired by the overhead camera. Background subtraction is used to detect the object, then the properties are extracted. Object properties are either discrete or continuous. The continuous object properties computed by our current implementation include x, y, θ, area, width, height, and aspect ratio, all of which are computed from the zeroth-, first-, and second-order centralized moments. Only one discrete property is computed, namely the dominant color. For each pixel in the object, the RGB triple is converted to hue-chroma-value, where chroma is the unnormalized version of saturation. Motivated by previous research showing that normalization obscures the true color properties of objects [25], we also have found chroma to be more discriminative than saturation. Each pixel is then classified into one of 8 categories — the 6 psychological primaries (red, green, blue, yellow, black, and white) along with orange and purple — using prototypes of each. Majority vote among all the pixels in an object wins for classifying the object according to its dominant color.

F. 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 the overhead camera; the objects are

segmented, and their feature properties are calculated using the procedure explained above. 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 used was a 7 degree-of-freedom Kuka lightweight robot arm, LBR 4+, with a 3-finger adaptive Robotiq gripper. Since all the objects involved in the experiments were relatively simple in shape, only pinch grasps were needed. 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 snapshots of some of the experiments that we have run, to illustrate the types of problems that our system can handle. Figure 6 shows snapshots corresponding to six experiments, while Table I shows the synthesized program of each of these experiments. Note that the system does not know which of the three different types of problems (sorting, kitting, or packaging) is being executed; these are merely examples.

B. Sorting Tasks

The first two experiments involved sorting tasks, in which objects were 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 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 involved kitting tasks, in which different objects were grouped together as a single unit. In contrast to sorting tasks, kitting tasks do not require objects of the same type to be placed in the same location.

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 object of each color (among the colors available) in the bin. In the first demonstration, the teacher moved one red, one green, one blue, and one black marker to the bin, leaving the extra red marker untouched. After another demonstration with a different initial set of markers, the synthesizer learned the correct program, which the system was then able to apply to any combination of any number of any colored objects. In the test, the robot was given a set of pens with different colors. With the learned program, the robot moved one green, one yellow, one pink, and one purple pen into the bin, leaving all the other pens untouched. Notice that the system is able to, at run time, handle object types and colors for which it was never specifically trained — this shows the power of the generalization capabilities of the system.

D. Packaging Tasks

The final two experiments involve packaging tasks, in which objects were 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, and several objects occupy the same location (at least in 2D).

Packaging a router box. In this experiment, a human teacher demonstrated packaging a box using components of

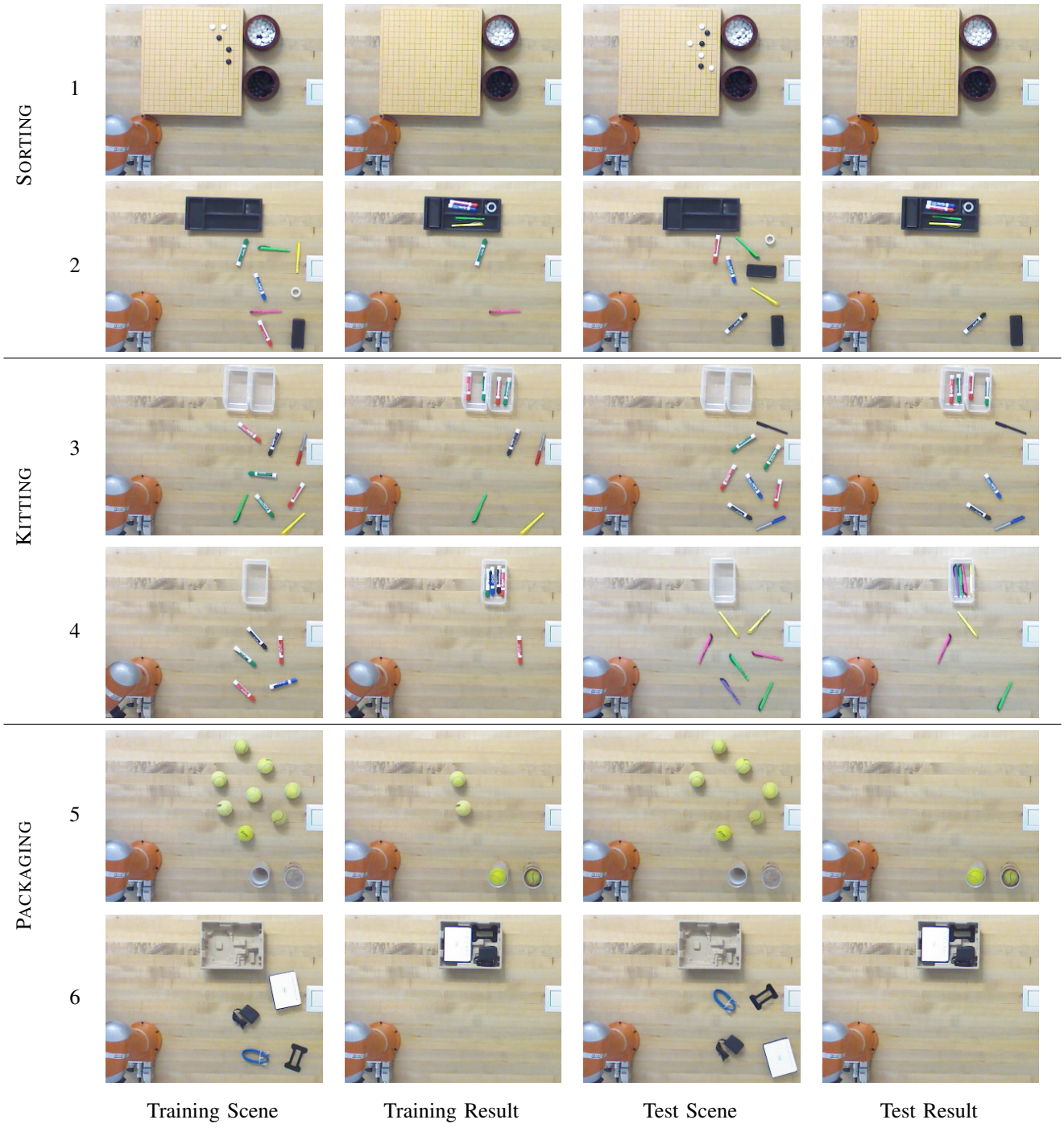


Fig. 6: Snapshots of the six experiments. For each experiment we show the initial and final conditions of the first demonstration, along with the initial and final conditions of a run in which the synthesized program was played on a scene that had never been seen. Several experiments required a single 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

TABLE I: Synthesized programs of the six experiments. Snapshots of the experiments are shown in Figure 6.

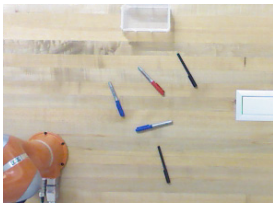
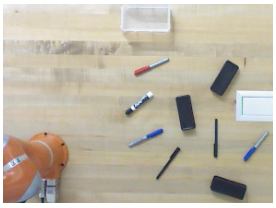
#	Demonstration	Program
1		**Synthesized Program:** things [’color @ ’black =] filter 590.50 272.50 3.14 moveall **Automatic translation:** “Move all things where dominant color is black to location (590.5, 272.5, 3.14).”
2		**Synthesized Program:** things [[’color @ ’black =] [’aspect @ 0.11 0.52 between not] bi and] filter 588.20 270.00 3.14 moveall **Automatic translation:** “Move all things where dominant color is black and aspect ratio is not between values 0.11 and 0.52 to location (588.2, 270.0, 3.14).”

Fig. 7: Progression of the program to move all black pens, as more demonstrations are given. Only the color was considered after the first demonstration, while the second demonstration forced both color and aspect ratio to be considered, resulting in the correct program. Also shown are automatically-generated human-readable English translations of the programs.

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, regardless of the initial positions of the objects.

E. Progression of Program Synthesis on Novel Scenes

A program is synthesized with one or more training scenes, and the correctness of the synthesized program is validated based on these same scenes. However, it is often the case that the first training scene does not include all the potential objects that could be present in the test scenes. Thus, although the synthesized program is consistent with the given demonstration, the intent of the user may not be correctly captured by the program. In such a circumstance, it is necessary to provide further demonstrations to train the robot such that the actual intent of the human teacher can be completely discovered.

Figure 7 shows the initial scene of two consecutive demonstrations, and the synthesized programs after each demonstration. The actual task was to separate solid black pens from other office supplies. In the first training scene, only two solid black pens and three markers with different colors were present. Therefore, when the user demonstrated by moving the black pens only, the system learned to move black objects to the bin. Although this program performs successfully on this particular training scene, it is not 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 together into the bin with other solid black pens, because it only sorted based on color. However, with a second set of demonstrations on the second scene, the correct program was learned, namely to move only the pens. Also shown in the figure are the nearly grammatically-correct English translations automatically generated by the system.

V. Conclusion

We have presented an approach to synthesizing human-readable computer programs for object repositioning tasks based on human demonstrations. We first introduced a stack-based domain specific language (DSL) for object repositioning tasks that is flexible and concise. We then described a learning algorithm which synthesizes a program of this DSL for an object repositioning task. Several experiments

were performed, showing that the resulting system is able to learn programs for a variety of object repositioning tasks (including sorting, kitting, and packaging tasks) from one or more demonstrations. We also introduced a novel tablet interface that rapidly speeds up the training and verification process, and we have shown that nearly grammatically-correct English sentence descriptions of the programs can be generated automatically.

It is easy to imagine ways to expand the capability of our system to represent a greater diversity of tasks. Beyond repositioning tasks, our work can be extended to more complicated and generic assembly tasks, such as assembling a toy car from individual parts. In assembly, not only the relative spatial relationships between parts need to be learned, but also how they come into those relationships (e.g., the mating procedure). The timing of actions is also important, so that by adding temporal dependence between actions, multi-step tasks can be modeled and learned. The system could also be extended to handle disturbances or non-determinisms in the world that prevent the manipulator from grasping objects in the simplistic manner shown here.

To scale our system to a wider range of objects, more advanced perception techniques are needed, including more robust feature extraction and object recognition. Distinctive and robust 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. Moreover, 3D perception and modeling would enable the system to perform on a wider array of scenarios.

VI. Acknowledgments

The authors wish to thank Harsha Kikkeri and Sumit Gulwani for initial discussions.

References

[1] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, pp. 469–483, May 2009.

[2] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *International Conference on Machine Learning (ICML)*, pp. 11–73, 1997.

[3] 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*, vol. 358, no. 1431, pp. 537–547, 2003.

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

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

[6] 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.

[7] 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.

[8] 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)*, pp. 1–6, Aug. 2011.

[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*, vol. 37, pp. 286–298, Apr. 2007.

[10] G. Konidaris, S. Kuindersma, R. Grupen, and A. Barto, "Robot learning from demonstration by constructing skill trees," *International Journal of Robotics Research*, vol. 31, pp. 360–375, Mar. 2012.

[11] 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.

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

[13] M. Cakmak, C. Chao, and A. L. Thomaz, "Designing interactions for robot active learners," *IEEE Transactions on Autonomous Mental Development*, vol. 2, no. 2, pp. 108–118, 2010.

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

[15] 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*, vol. 5, no. 4, pp. 477–490, 2013.

[16] 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)*, pp. 255–262, Mar. 2007.

[17] L. Montesano, M. Lopes, A. Bernardino, and J. Santos-Victor, "Learning object affordances: From sensory-motor coordination to imitation," *IEEE Transactions on Robotics*, vol. 24, no. 1, pp. 15–26, 2007.

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

[19] 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)*, pp. 685–691, Nov. 2013.

[20] 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)*, pp. 317–328, 2011.

[21] 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)*, pp. 317–330, 2011.

[22] 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)*, pp. 237–242, 2012.

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

[24] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[25] A. Hanbury, "Constructing cylindrical coordinate colour spaces," *Pattern Recognition Letters*, vol. 29, pp. 494–500, Mar. 2008.