Synthesizing Environment-Aware Activities via Activity Sketches

Yuan-Hong Liao[1,2*], Xavier Puig[3*], Marko Boben[4,5], Antonio Torralba[3], Sanja Fidler[1,2]

[1] University of Toronto, [2] Vector Institute, [3] MIT, [4] University of Ljubljana, [5] Abelium d.o.o

{andrew, fidler}@cs.toronto.edu, {xavierpuig,torralba}@csail.mit.edu, marko.boben@fri.uni-lj.si

Abstract

In order to learn to perform activities from demonstrations or descriptions, agents need to distill what the essence of the given activity is, and how it can be adapted to new environments. In this work, we address the problem of environment-aware program generation. Given a visual demonstration or a description of an activity, we generate program sketches representing the essential instructions and propose a model to transform these into full programs representing the actions needed to perform the activity under the presented environmental constraints. To this end, we build upon VirtualHome [15] to create a new dataset VirtualHome-Env, where we collect program sketches to represent activities and match programs with environments that can afford them. Furthermore, we construct a knowledge base to sample realistic environments and another knowledge base to seek out the programs under the sampled environments. Finally, we propose ResActGraph, a network that generates a program from a given sketch and an environment graph and tracks the changes in the environment induced by the program.
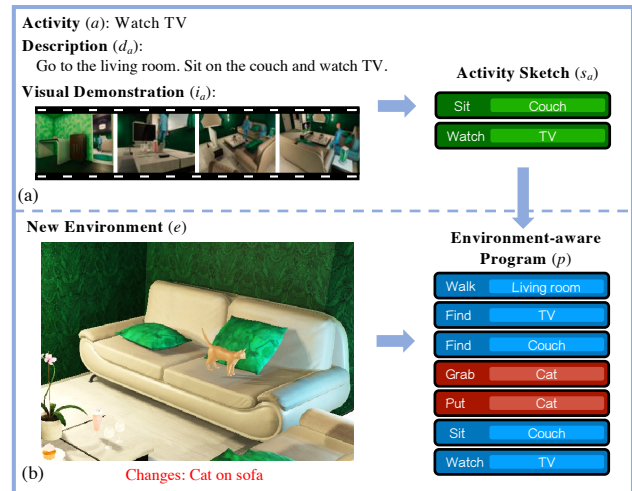
Figure 1. Overview of the *environment-aware program generation*. Our goal is (a) generating a sketch s_a distilling the essential steps of the given demonstration i_a or description d_a and (b) given a new environment e, generating a program p, adapting the sketch to e. The program contains the instructions to perform the activity (blue blocks) as well as instructions to deal with the environment (red blocks, grabbing the cat to sit).

1. Introduction

We want agents to be able to perform everyday tasks (such as setting up the table, preparing coffee, or even sit on the couch and watch TV). An agent should learn to perform these tasks from high-level descriptions or visual demonstrations. The challenge is on how to generalize the acquired knowledge to new environments. For instance, if we want to learn to make coffee, an agent could first watch a video of someone making coffee in order to extract the sequence of steps (program) that need to be executed. However, when trying to make coffee in an environment that differs from the environment in which the demonstration took place, the agent has to adjust the program so that it can be executed. For instance, the agent could find that the coffee machine is unplugged, or that it is in the living room instead

of in the kitchen, or that someone else is currently using it and the agent needs to wait. Being able to perform these adjustments requires the access to a common-sense database of knowledge that allows the agent to decide which steps in the demonstration are essential in the task definition, and how the program needs to be modified (by adding/removing steps) in order to accomplish the task in a new environment.

To address the generalization problem we represent activities with sketches, representations inspired from work in programming languages [16, 13]. Sketches are high-level representations of the steps needed to perform a task but leaving holes that need to be completed for the sketch to become executable in a particular environment.

Fig. 1 illustrates our goal. Given a visual demonstration or a description of someone going to watch TV, we want to extract the sketch of the activity (fig. 1a). In this example, the sketch consists of two steps: Sit Couch, Watch TV. The fact that the demonstration had the person going to the

---

living-room is not important. Watching TV requires us to go to the room that contains the TV. Executing this activity in a new environment requires expanding the sketch into a complete program (fig. 1b). The sketch expansion depends on the state of the environment. In this example, it turns out that there is a cat on the couch. To sit on the couch, we need to push away the cat first. We show the automatically generated program that includes all the steps needed to complete the task and to deal with the cat.

To expand sketches into programs, it is necessary to have access to commonsense knowledge about the world and how to deal with typical situations. This knowledge provides ways to solve situations that can be reused in a multitude of tasks. Since no such information, with diverse programs and environments, exists to date, we collect a new dataset, built upon VirtualHome [15], containing around 3k home activities. We extend the VirtualHome dataset to include more actions and over 30k programs and collect sketches of the activities by crowdsourcing. Each environment is represented as a graph with 300 objects and 4000 spatial relations on average.

We then propose a model to generate programs by selecting, for every step, a node in the environment graph representing an object of interaction. To do so, we exploit Graph Neural Networks (GNNs) to reason about the states and relations between the objects in the environments. Furthermore, we propose ResActGraph, a model that reasons about the changes in the graph induced by the agents previous steps to generate the goal program.

The main contributions of this work are: introducing sketches as environment-independent representations of an activity, a database of commonsense knowledge of activities, sketches and how to deal with a variety of situations, and a method to generate programs from sketches that accomplish the task in a new environment. We show results in VirtualHome [15].

2. Related Work

Learning from demonstrations. Learning from visual demonstrations or language descriptions has been of increasing interest in both robotics and computer vision. However, this has been mostly focused on learning low-level tasks rather than the high semantic level activities that we tackle in this work. For example, [18, 11, 1] focus on learning to navigate in environments or manipulate objects, while in visual imitation learning, multiple works have used videos to learn to manipulate objects under low supervision regimes [14, 7] or imitate kinetic human behaviors [12]. Our work is more closely related to semantic planning [9, 20, 3], which focus on modeling sequences of composite and semantically loaded actions. Learning those requires inferring and modeling the sub-goals of a given task. [9] represents such goals as a graph, with nodes be-

ing actions and edges being precondition states, while we propose program representations. Furthermore, to perform such tasks, it is necessary to know what the constraints of the given environment where it will be executed are. Similarly to [3], we propose to encode knowledge of the environments and how they can constrain the activities to be performed.

Program Synthesis by Sketching. Our approach for environment-aware program generation is partially inspired by performing program synthesis by sketching. In [16], a sketch expresses the high-level structure of an implementation but leaves holes in place of low-level details, which corresponds to our model that derives the details based on the environmental constraints so as to execute the programs smoothly.A recent body of work has developed neural approaches to program generation using user-provided examples [8, 4], visual demonstrations [17], and descriptions [6]. The work [13] is the most related to ours. They learn a model that predicts sketches of programs relevant to a label and the predicted sketches are concretized into code using combinatorial techniques. The main difference between our work and theirs is not only that our sketches are inferred from visual or textual data, but that we focus on how to incorporate the environmental constraints in program generation.

3. Problem formulation

The goal of environment-aware program generation is to predict, given a demonstration or description of an activity and an environment, a program that can execute the activity in such environment. We define this task and the corresponding notation in this section.

Let A and E be the universe of activities and environments. An activity a (e.g. watch TV) can be represented as a set of programs P_a containing a sequence of instructions (e.g. `TurnOn TV`, `Sit Sofa`, `Watch TV`), which vary depending on how the activity is performed. Let $\mathbb{1}(p, e)$ be an indicator function determining if p can be executed in e (e.g. an agent can not grab cups inside a closed cabinet). Given an activity $a \in A$ and an environment $e \in E$, our goal is to learn a model that generates p such that

$$p \in P_a, \mathbb{1}(p, e) = 1 \qquad (1)$$

We use the corresponding visual demonstrations $i_a \in I_a$ or descriptions $d_a \in D_a$ to specify a. However, i_a and d_a are implicitly conditioned on certain environments which might differ from the current one e. Therefore, directly inferring $\hat{p}$ does not ensure that $\hat{p}$ satisfies eq. 1 given e.

Inspired by [16], we introduce program sketches $s_a \in S$ as *environment-independent* representations of the activities. We thus change the constraint in eq. 1 to be:
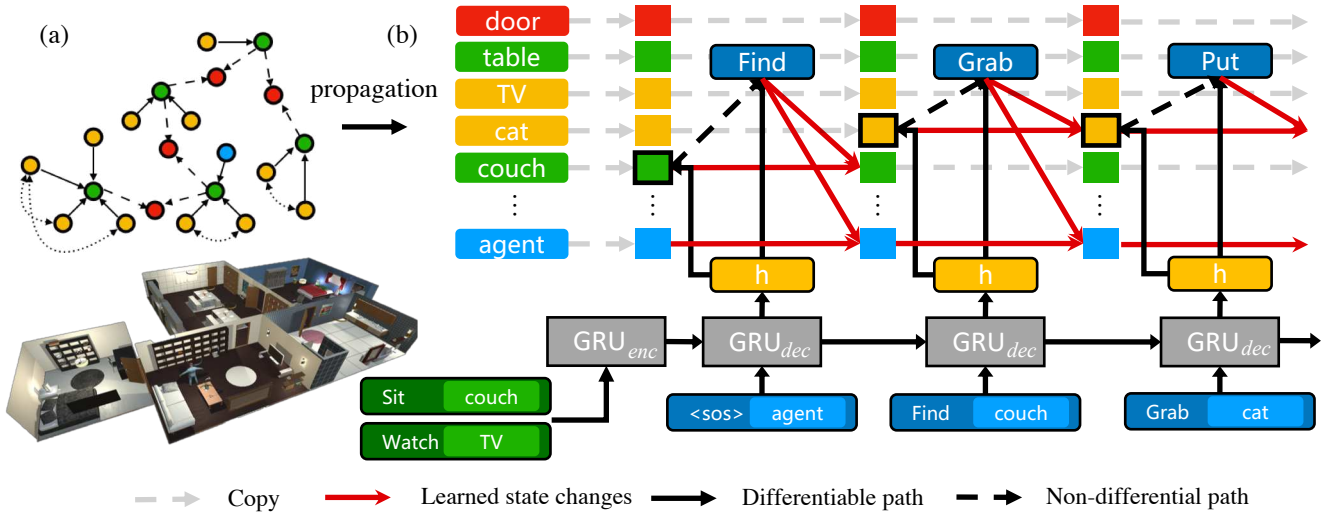
Figure 2. (a) We extract the ground truth environment graph from VirtualHome and perform message passing on the graph. (b) At every time step, the decoder perform sequential classification over the hidden states of the graphs (top row). The selected nodes are shown in bold border blocks. We also model the environment changes induced by the generated programs (solid red arrows).

$$p \in P_{s_a}, \mathbb{1}(p, e) = 1 \qquad (2)$$

With the program sketches as proxy representations, we can divide the task into two sub-problems: a model that predicts a sketch $\hat{s}_a$ from a demonstration i_a or a description d_a and another model that predicts a program $\hat{p}$ given the predicted sketch $\hat{s}_a$ and an environment e:

$$\hat{p} = f_{sketch2prog}(\hat{s}_a, e), \text{ where} \\ \hat{s}_a = f_{demo2sketch}(i_a) \text{ or } \hat{s}_a = f_{desc2sketch}(d_a) \qquad (3)$$

Here, p and s_a are a sequence of instructions. Each instruction is represented by an action and up to two arguments representing objects of interaction $(\alpha, \beta^1, \beta^2)$.[1]

4. Model

In this section, we present our approach to the *environment-aware program generation* task. First, we introduce *ResActGraph* to generate programs from sketches and target environments. Later on, we describe how we predict sketches from demonstrations or descriptions.

4.1. Program generation from sketches and graphs

We frame the program generation task as a seq2seq problem, where an encoder encodes the input sketch and the decoder generates the target program one instruction at a time, composed by an action and object arguments. Given that

the program must be grounded in a target environment, instead of predicting the objects from a fixed taxonomy, the model predicts for each instruction object instances that are present in the environment. This has two benefits: (1) It avoids referring to object instances that do not exist in the environment. (2) It allows the model to use information of each instance within the environment, such as its state or relations with other objects, to predict the appropriate instruction.

To do that, we encode the scene as a graph $G = (\mathcal{V}, \mathcal{R})$ modeling the dependencies of the object instances. The node $v \in \mathcal{V}$ indicates the object instance and each node has a label, including the object class c_v, its states l_v, and properties $prop_v$. Note that $\mathcal{V}$ includes a node for the agent itself. The edge $r \in \mathcal{R}$ encodes the spatial relations, including ON, IN_OBJ, IN_ROOM, CLOSE_TO, and FACE_AT, between every two object instances.

The node labels and relations are used to obtain vector embeddings for each instance which are used by the decoder to predict the environment-aware program, as we describe in the following section.

4.2. ResActGraph

We adopt the GGNN [10] framework to obtain the hidden states of the nodes and capture the object relations in the environment graph. The hidden states of each node v are initialized by its label $(c_v, l_v, prop_v)$:

$$h_v^0 = tanh(g_{init}([W_c c_v, W_l l_v, W_{prop} prop_v])) \qquad (4)$$

We apply one-hot encoding to the label and set

---

[1]The number of arguments depends on the type of the action.

W_c, W_l, W_{prop} as learnable weights. g_{init} is a network composed of fully connected layers that combine all the information.

At propagation step k, each node's incoming information x_v^k is determined by aggregating the hidden states of its neighbors $v' \in \mathcal{N}(v)$ at the previous step $k-1$:

$$x_v^k = \sum_{j \in L(\mathcal{R})} \sum_{v' \in \mathcal{N}_j(v)} W_{p_j} h_{v'}^{k-1} + b_{p_j} \quad (5)$$

$L(\mathcal{R})$ denotes the set of edge labels and the linear layer W_{p_j} and bias b_{p_j} are shared across all nodes.

After aggregating the information, the hidden states of the nodes are updated through a gating mechanism similar to Gated Recurrent Unit (GRU) [5] as follows:

$$\begin{aligned}
z_v^k &= \rho(W_z x_v^k + U_z h_v^{k-1} + b_z), \\
r_v^k &= \rho(W_r x_v^k + U_r h_v^{k-1} + b_r), \\
\hat{h}_v^k &= tanh(W_h x_v^k + U_h(r_v^k \odot h_v^{k-1}) + b_h), \\
h_v^k &= (1 - z_v^k) \odot h_v^{k-1} + z_v^k \odot \hat{h}_v^k
\end{aligned} \quad (6)$$

This results in a vector embedding for each object h_v^k, with information about its state and relationship with the environment.

We use one GRU to encode the sketches and another one to generate the program one instruction at a time. For time t, let $feat_{s_a} = enc(s_a)$ be the output of the sketch encoder, and h_{dec}^t the hidden states of the decoder. To predict the program instruction, $(\hat{\alpha}_t, \hat{\beta}_t^1, \hat{\beta}_t^2)$, we predict the first object argument β_t^1 over the graph nodes, use it to predict the action $\hat{\alpha}_t$ and combine this information to predict the second argument $\hat{\beta}_t^2$:

$$\begin{aligned}
\hat{\beta}_t^1 &= \arg\max_{v \in \mathcal{V}} \sigma(g_{\beta^1}(h_{dec}^t, h_v^K, feat_{s_a})) \\
\hat{\alpha}_t &= \arg\max_{\alpha \in \mathcal{A}} \sigma(g_\alpha(h_{dec}^t, h_{\hat{\beta}_t^1}^K, feat_{s_a})) \\
\hat{\beta}_t^2 &= \arg\max_{v \in \mathcal{V}} \sigma(g_{\beta^2}(h_{dec}^t, h_v^K, feat_{s_a}, h_{\hat{\beta}_t^1}^K, \hat{\alpha}_t))
\end{aligned} \quad (7)$$

where $\mathcal{A}$ is all the possible actions and σ denotes the `softmax` function.

Note that so far the hidden states of the nodes h_v^K are constant over t, but we would like them to change according to the program being executed. To do that, at time t, we use the previously generated instructions $(\alpha_{<t}, \hat{\beta}_{<t}^1, \hat{\beta}_{<t}^2)$ to update the hidden states of the nodes. We set the initial state of each node as $h_v^{K_0} = h_v^K$ and update the state $h_v^{K_t}$ at time t if v is interacted by the agent at the previous time step or is the agent itself. For example, if the instruction at the previous time step is `grab mug`, we use the action embeddings of `grab` to change the hidden states of `agent`

and `mug`. Let $\hat{v}$ correspond to the agent node or one of the previous arguments $\hat{\beta}_{t-1}^1, \hat{\beta}_{t-1}^2$. The state $h_{\hat{v}}^{K_{t-1}}$ is updated as follows:

$$\begin{aligned}
h_{\hat{v}}^{K_t} &= h_{\hat{v}}^{K_{t-1}} + r \\
r &= tanh(g_{res}(h_{\hat{v}}^{K_{t-1}} \odot g_e(Emb(\hat{\alpha_{t-1}}), m_{t-1})))
\end{aligned} \quad (8)$$

where $Emb(\hat{\alpha_{t-1}})$ is the embedding of $\hat{\alpha_{t-1}}$ and m_{t-1} is a one-hot encoding denoting if v is the subject or the object of $\hat{\alpha_{t-1}}$. g_e and g_{res} consider the change of the $h_{\hat{v}}^{K_{t-1}}$ and predict the residuals. Note that since the node `agent` is involved at every time step, it tracks the progress through the generation. The model overview is shown in Fig. 2.

Learning. We use the cross-entropy loss function for program prediction. The GGNN and GRUs are then trained with the back-propagation through time (BPTT).

4.3. Inferring sketches

Activities specified by demonstrations. We use key frames $i = [i_n]_{n=1:N_{demo}}$ as the representations of the demonstrations, where N_{demo} is the length of the key frames. To be specific, we take the bird-eye view of each i_n. Besides, we also use the ground truth semantic segmentation map $i_{seg} = [i_{seg_n}]_{n=1:N_{demo}}$ as input. Two CNNs are used to extract the features separately, and we apply late fusion to extract the nth visual features $feat_n$ as follows:

$$feat_n = g_{fuse}([CNN_i(i_n), CNN_{seg}(i_{seg_n})]) \quad (9)$$

where $[,]$ denotes concatenation. Later on, we max-pool the features over the different steps time steps and apply a GRU to decode the sketches.

Activities specified by descriptions. We adopt the seq2seq model with a GRU encoding each word in the description and a GRU to decode each of the sketch instructions.

5. Dataset: VirtualHome-Env

Our goal is to generate a sketch s_a from a demonstration i_a or description d_a and induce a program p from s_a and a target environment e. In this section, we describe how we obtain the dataset containing (a, i_a, d_a, s_a, e, p). We briefly introduce VirtualHome as the playground for the dataset collection, we describe how we extend the dataset with activity sketches and finally describe how we pair the programs with arbitrary environments where they can be performed, motivating the creation of a common sense knowledge base of activities[2].

Background. VirtualHome is a dataset and simulator to represent human household activities. The dataset was collected by asking annotators to come up with various activities and provide a description for them. In a second

---

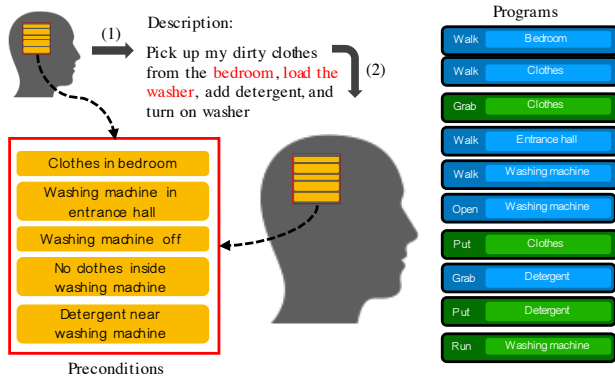[2]The details of the collected knowledge base are described in **??**

Figure 3. The annotators label the descriptions and programs with certain environments in mind (bottom left), resulting in the *environment-dependent* descriptions and programs. The blocks colored in blue are considered as environment-dependent components.

stage, annotators were shown a description of an activity and were asked to write a program from it. With activities represented as programs, a Unity simulator executes them in some predefined apartments and renders the programs as videos. This allows us to obtain activities and programs together with descriptions and demonstrations (a, i_a, d_a, p).

5.1. Collecting sketches

When collecting each activity in VirtualHome, annotators imagine an environment where the activity could take place and provide a description according to it. As a result, the description and subsequent program is specific to a given environment, but may not be doable when presented with new environments or constraints (see Fig. 3). Therefore, we need a more abstract representation of an activity, which can be consistent with multiple environments and the information (a, i_a, d_a, p).

Inspired by [16], we collect the sketches of the activities to abstract out the components that are environment-dependent and informally define the sketches as the *environment independent representations*. Different from programming languages, it is highly non-trivial to define the sketches of the activities since they depend on the common-sense of each individual. Therefore, we manually collect the sketches and get the information (a, i_a, d_a, s_a, p).

5.2. Pairing programs with environments

We finally need to add an environment that pairs with the activity programs. Since we do not know the environment that each annotator had in mind, we need to infer it from the program. We first extract the preconditions of the programs and use them to sample feasible environments. We define preconditions of a program as the conditions that have to be true in the environment in order to execute the program in it.

For example, in order to execute `watch TV`, the tv should be on. We construct a function Φ that infers the preconditions from p and sample e from the set of environments that satisfy such preconditions $E_{\Phi(p)}$:

$$e \sim E_{\Phi(p)} \subset E, \text{ s.t. } \forall j \in \Phi(p), e \text{ satisfies } j \qquad (10)$$

The above is a weak constraint since it does not inform about objects that are not specified in $\Phi(p)$. To get realistic environments, these objects should follow some priors. For example, couches can be occupied, but they can not be cold. Apples can be stored inside fridges, but they are seldom found in bathtubs. We build collect these rules in a knowledge base, *KB-RealEnv* and use them to build the environment, starting from the environments provided in VirtualHome.

5.3. Extending programs to diverse environments

We now have the information (a, i_a, d_a, s_a, e, p). However, relying solely on the original programs results in a limited set of preconditions and thus environments. One possible reason is that when describing activities, annotators tend to assume the simplest setting to perform the given activity. For example, when thinking of doing the laundry, it is common to imagine that the washing machine is idle or empty. To address that, we build a simulator Ψ that takes a program p and environment graph e and outputs the graph corresponding to the environment after executing the program, or raises an exception if the program is not executable at a certain step. Given a program with preconditions $\Phi(p)$, we start by randomly perturbing them into $\Phi(p)'$ and use eq. 10 to obtain e' as an environment satisfying $\Phi(p)'$. Then, we execute p in the simulator with the environment e'. Given that the environment and preconditions have changed but the program is still the same, as the program is executed, some exception will be raised from the simulator. Then, a subroutine is called to modify the program p into p', by inserting or removing instructions to correct the exception, obtaining the extended program.

For example, when we executing `Sit Sofa`, if `Sofa` is occupied, the subroutine is expected to perform actions to remove things on the `Sofa` until there is enough space to `Sit`. We manually compose the subroutines based on different types of exception, forming the knowledge base, *KB-ExceptionHandler*. We show more details of how we augment the programs in the supplementary materials.

This way, we augment over 30k tuples of sketches, environments, and programs (s_a, e', p'). Note that the sketches s_a are environment-independent, so there is no need to change them after applying the subroutines.
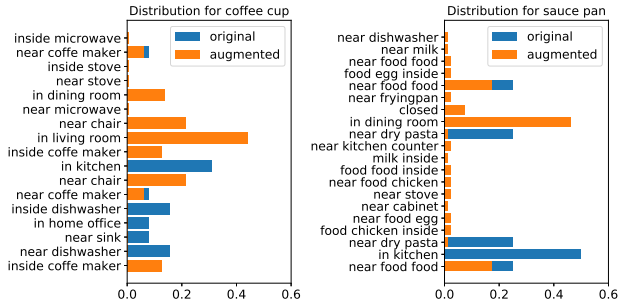
Figure 4. The effect of the dataset augmentation: changes in the distribution of preconditions for the objects in the environment.

5.4. Dataset Analysis

From the original 2807 programs in VirtualHome, we trim out the programs that can not be executed in the simulator with the environment sampled via preconditions, obtaining 1387 executable programs. Using the process described in Sec. 5.3, we extend these programs to our final dataset with around 30k programs. The significant increase in the program length is induced by the modified preconditions. For example, the agent needs to open containers to reach objects or make space to sit on a sofa. In Table **??**, we show the statistics of the new dataset. Fig. 4 shows the change in the distribution of preconditions of the cup and the sauce pan. The programs after augmentation show a less skewed distribution of preconditions and therefore more diverse environments. Finally, we use the simulator to generate snapshots of the environment after executing each instruction of a program, as show in fig. **??**. Note that some of the objects in the program do not have a model in the simulator, so we generate frames for a subset of 8421 programs.

6. Experiments

We split the dataset into train and test set in terms of different types of activities with ratio 7:3 and leave one apartment for the test set. We aim to test the capability of our model with novel *activities* and *environments*. We follow the same split for sketch prediction, where we only keep the original programs for the *desc2sketch*, since they contain the collected descriptions, and use the available frames for the demo2sketch task.

In this section, we describe the evaluation metrics, baselines. Next, we show the extensive experiment results of *ResActGraph*. Finally, we analyze the extent to which the proposed method is *environment-aware*. We will describe the implementation details in supplementary materials.

6.1. Evaluation Metrics

We analyze the performance of sketch prediction and program generation by measuring the normalized longest common subsequence (LCS) between the generated and ground truth sequences. LCS is sensitive to the order of the sequences and allows gaps in between. To further measure if the generated programs achieve the specified activities, we compute the differences between the final environment graphs $\hat{G} = \Psi(\hat{p}, e)$ and $G = \Psi(p, e)$ using $\mathbb{F}_1$ scores[3]. In particular, we only compared the sub-graph containing the object instances mentioned in p and $\hat{p}$. We describe the details of $\mathbb{F}_1(\hat{G}, G)$ in the supplementary materials. Inspired by [2], we also compute $\mathbb{F}_1$-state and $\mathbb{F}_1$-relation.

Furthermore, inspired by program synthesis, we care whether the generated programs are "compilable" as well. We evaluate if the generated programs can be parsed (parsibility) and executed (executability) by the simulators. We will describe the detailed definition of them in the supplementary materials.

6.2. Baselines

We implement five different baselines to compare with the proposed *ResActGraph*.

Nearest Neighbors: For every example in the testing set, we retrieve the training sample that has a sketch with the highest LCS. In case of a tie, we pick the one with the most similar initial graph.

Unaries: We set $K = 0$ in Eq. 7. This model does not consider the relations of the objects. We use it to showcase the benefits of modeling object relations.

Graph: This model does not consider the change of graphs induced by programs (Eq. 8).

FCActGraph: This model uses a FC layer to model the graph changes. Specifically, it takes the $[h_v^{K_{t-1}}, Emb(\hat{\alpha_{t-1}}), m_{t-1}]$ as inputs and outputs $h_v^{K_t}$.

GRUActGraph: This model treats the graph changes as another sequence and uses a GRU to ingest $[Emb(\hat{\alpha_{t-1}}), m_{t-1}]$ as inputs and considers $h_v^{K_{t-1}}$ as hidden state to output $h_v^{K_t}$.

6.3. Results

We show the results of *ResActGraph* quantitatively and qualitatively. Next, we show the ablation study of the number of the graph propagation steps. Finally, we show the prediction results of the whole system.

Program generation from sketches and graphs. The results are shown in Table 1. By comparing the Graph and Unaries, we show that aggregating information from neighboring nodes increases performances in nearly all metrics.

The three bottom rows of Table 1 show the results of models that consider graph changes induced by programs. The $\mathbb{F}_1$ scores and executability benefit the most, which is expected. For example, suppose there is a glass near an opened cabinet in the environment and the model predicts (Grab Glass, Put Glass Cabinet, Close

---
[3]If the generated programs cannot be parsed or cannot be executed, the $\mathbb{F}_1$ is set to be 0.

	LCS	F_1-relation	F_1-state	F_1	Executability	Parsability
Nearest Neighbors	0.463	0.537	0.051	0.100	-	-
Unaries	0.39	0.188	0.179	0.188	22.93%	79.16%
Graph	**0.526**	0.4	0.397	0.4	46.79%	78.9%
FCActGraph	0.515	0.423	0.4	0.4	48.23%	84.44%
GRUActGraph	0.517	0.426	0.428	0.429	50.28%	**84.74%**
ResActGraph	0.519	**0.432**	**0.436**	**0.436**	**51.1%**	83.66%

Table 1. Induce program from ground truth sketches and ground truth graphs. ($K = 2$)

Sketch	GT Program	Generated Program
[Open] <washing machine> [Put] <basket> <washing machine> [Put] <soap> <washing machine> [SwitchOn] <washing machine>	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket> (1000) [Grab] <basket> (1000) [Walk] <bathroom> (1) [Walk] <washing machine> (1001) [Find] <washing machine> (1001) [Open] <washing machine> (1001) [Put] <basket> (1000) <washing machine> (1001) [Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing machine> (1001) [Find] <washing machine> (1001) [Close] <washing machine> (1001) [PlugIn] <washing machine> (1001) [SwitchOn] <washing machine> (1001)	[Walk] <bedroom> (273) [Walk] <basket> (1000) [Find] <basket > (1000) [Grab] <basket > (1000) [Find] <washing machine> (1001) [Open] <washing machine> (1001) [Put] <basket> (1000) <washing machine> (1001) [Find] <soap> (1002) [Grab] <soap> (1002) [Put] <soap> (1002) <washing machine> (1001) [Close] <washing machine> (1001) [Plugin] <washing machine> (1001) [SwitchOn] <washing machine> (1001)
Environment		
Washing machine (1001) is closed Washing machine (1001) is off Washing machine (1001) is unplugged Washing machine (1001) in bathroom (1) Soap (1002) inside Washing machine (1001)		

Figure 5. An example of the prediction of ResActGraph. We colore the LCS between the prediction and ground truth in light green. Note that the sketch is environment agnostic, so it does not specify the 'id' (the number in the parentheses) of the object instances.

	LCS	F_1-relation	F_1-state	F_1
Unaries (K=0)	0.39	0.18	0.17	0.18
Graph (K=1)	0.48	0.34	0.36	0.35
Graph (K=2)	0.526	0.4	0.397	0.4
Graph (K=3)	0.527	0.39	0.39	0.39
ResActGraph (K=1)	0.49	0.38	0.39	0.39
ResActGraph (K=2)	0.519	**0.432**	**0.436**	**0.436**
ResActGraph (K=3)	**0.536**	0.415	0.418	0.419

Table 2. Ablation study of the propagation steps K.

Cabinet) at the first three steps, if the model wants to grab other things from the cabinet without opening it after $t \geq 4$, it fails since the cabinet is closed at $t = 3$.

Among the three bottom rows of Table 1, the proposed ResActGraph performs the best in F_1. The reason is that using the residual architecture is easier for the model to learn the state "changes" compared to using FC. Using GRU to encode the state changes is also an alternative, but we observe that it converges slower since it has more number of parameters to learn and does not perform better.

In Fig. 5, we show the qualitative results of ResActGraph. Even though the generated program does not exactly

match the ground truth, it reaches nearly the same environment state. In Fig. 6, we show the results with the same sketch, but different initial environment states. Note that we only show the states and relations related to the programs. The model correctly induces correct actions w.r.t. the environment changes and the two generated programs nearly reach the same environment states.

Ablation studies. We show the effect of the number of propagation steps K in Table 2. Both the baseline and the proposed model benefit as K increases, and the proposed model performs better than the baseline regardless of different K. We found that the performance saturates when $K = 2$, so we fixed it for all other experiments.

Combining predicted sketches with ResActGraph. The LCS of $f_{demo2sketch}$ and $f_{desc2sketch}$ are 0.16 and 0.33 respectively. The reason why the LCS is low is that they are significantly shorter (on average 2.4 instructions) than the programs (on average 18.79). This makes the sketch prediction a quite challenging task where LCS is highly penalized even under small errors.

With the trained model, we can directly generate the programs from the demonstrations or descriptions. Note that we do not re-train the program generation model. In Ta-

Sketch	[Sit] <sofa> [Read] <book>		
Environment 1	Generated Program 1	Environment 2	Generated Program 2
Book (263) in bedroom (23) Bookmark (27) near book (263) Bookmark (27) in bedroom (23) Sofa (101) in bedroom (23) Sofa (101) near bookmark (275) Sofa (101) is free Book (263) near sofa (1001)	[Walk] <bedroom> (23) [Walk] <book> (263) [Find] <book> (263) [Grab] <book> (263) [Find] <sofa> (101) [Sit] <sofa> (101) [Find] <bookmark> (27) [Read] <book> (263)	Sofa (101) in bedroom (23) Book (263) in bedroom (23) Bookmark (27) in bedroom (23) Sofa (101) occupied Phone (75) on sofa (101) Cards (261) on sofa (101) Game (231) on sofa (101) Check (213) on sofa (101)	[Walk] <bedroom> (23) [Grab] <game> (231) [Walk] <sofa> (101) [Release] <game> (231) [Find] <sofa> (101) [Find] <check> (213) [Find] <phone> (75) [Grab] <check> (213) [Grab] <phone> (75) [Release] <check> (213) [Release] <phone> (75) [Sit] <sofa> (101) [Find] <cards> (261) [Standup] [Grab] <cards> (261) [Walk] <book> (263) [Release] <cards> (261) [Find] <book> (263) [Find] <game> (231) [Grab] <book> (263) [Read] <book> (263)

Figure 6. An example of the prediction of ResActGraph with the same sketch, but different environments. We highlight the difference between two environments with orange and color the LCS between two predictions in light green.

Demonstration	Environment	Generated Sketch	Generated Program	
	Phone (10) in living room (1)	[Grab] <phone>	[Walk] <living room> (1) [Walk] <phone> (10) [Find] <phone> (10)	[Walk] <phone> (10) [TurnTo] <phone> (10) [LookAt] <phone> (10) [Grab] <phone> (10)
Description	Environment	Generated Sketch	Generated Program	
Walk into the home office. Walk up to the chair, sit down in the chair. Type with the keyboard.	Chair (29) close to keyboard (2) Chair (29) is free Keyboard (2) close to computer Computer (31) in living room (1)	[Sit] <chair> [Type] <keyboard>	[Walk] <living room> (1) [Walk] <desk> (137) [Find] <chair> (29)	[Sit] <chair> (29) [Find] <keyboard> (2) [Type] <keyboard> (2)

Figure 7. Predictions from the ResActGraph given sketches from descriptions and demonstrations.

	LCS	$\mathbb{F}_1$-relation	$\mathbb{F}_1$-state	$\mathbb{F}_1$	Executability	Parsability
Unaries	0.25	0.16	0.16	0.16	32.05%	**86.71%**
Graph	0.4	**0.33**	0.32	0.33	43.69%	82.5%
ResActGraph	0.4	0.32	**0.34**	0.33	**47.22%**	80.94%

Table 3. Induce program from sketches predicted from decsriptions and ground truth graphs. ($K = 2$)

	LCS	$\mathbb{F}_1$-relation	$\mathbb{F}_1$-state	$\mathbb{F}_1$	Executability	Parsability
Unaries	0.23	0.24	0.25	0.24	97.77%	68.88%
Graph	**0.4**	**0.39**	**0.45**	**0.4**	77.77%	60%
ResActGraph	0.3	0.36	0.39	0.35	**93.33%**	66.66%

Table 4. Inducing program from sketches predicted from demonstrations and ground truth graphs. ($K = 2$)

ble 3 and Table 4, we show the results of the program generation with the sketches predicted from descriptions and demonstrations respectively. The performance gap between the proposed model and the baselines becomes small. The reason is that the model is confused when the non-perfect sketches are given, resulting in similar performance. Note that all models still perform better than Unaries. Qualitative results are shown in Fig. 7, showing that the model predicts the plausible sketch and *ResActGraph* generates plausible programs w.r.t to the sketch and the graph.

7. Conclusion

In this work, we propose the *environment-aware program generation* task. We introduce sketches as environment-independent activity representations and address the problem in two steps: generating sketches from demonstrations or descriptions and generating programs from sketches and graphs. To this end, we propose a novel model, *ResActGraph* and create a dataset *VirtualHome-Env*, with sketches, environments, and programs.

The *environment-aware program generation* is far from being solved and opens exciting research directions. While we access the truth state of the environment graph, one natural extension would be to predict it from environment observations [19], but this would still require an oracle providing images of the interior of closed objects or unexplored areas. Additionally, though the *ResActGraph* updates the hidden states of nodes at each time step, the graph structure is not changed, which impedes from performing message passing at each time step. Finally, our setting assumes a fully observable environment. Working with partially observable environments opens up exciting directions, such as dealing with environment changes due to factors external to the agent or modelling the agent's theory of mind.

References

[1] M. Alomari, P. Duckworth, M. Hawasly, D. C. Hogg, and A. G. Cohn. Natural language grounding and grammar induction for robotic manipulation commands. In *Proceedings of the First Workshop on Language Grounding for Robotics*, pages 35–43, 2017. 2

[2] P. Anderson, B. Fernando, M. Johnson, and S. Gould. Spice: Semantic propositional image caption evaluation. In *European Conference on Computer Vision*, pages 382–398. Springer, 2016. 6

[3] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth. Robotic roommates making pancakes. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 529–536. IEEE, 2011. 2

[4] M. Bošnjak, T. Rocktäschel, J. Naradowsky, and S. Riedel. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016. 2

[5] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014. 4

[6] L. Dong and M. Lapata. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018. 2

[7] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine. One-shot visual imitation learning via meta-learning. In *Conference on Robot Learning*, pages 357–368, 2017. 2

[8] A. L. Gaunt, M. Brockschmidt, R. Singh, N. Kushman, P. Kohli, J. Taylor, and D. Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428*, 2016. 2

[9] D.-A. Huang, S. Nair, D. Xu, Y. Zhu, A. Garg, L. Fei-Fei, S. Savarese, and J. C. Niebles. Neural task graphs: Generalizing to unseen tasks from a single video demonstration, 2018. 2

[10] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. 3

[11] H. Mei, M. Bansal, and M. R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences, 2015. 2

[12] J. Merel, Y. Tassa, S. Srinivasan, J. Lemmon, Z. Wang, G. Wayne, and N. Heess. Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*, 2017. 2

[13] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017. 1, 2

[14] A. Nair, D. Chen, P. Agrawal, P. Isola, P. Abbeel, J. Malik, and S. Levine. Combining self-supervised learning and imitation for vision-based rope manipulation. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pages 2146–2153. IEEE, 2017. 2

[15] X. Puig, K. Ra, M. Boben, J. Li, T. Wang, S. Fidler, and A. Torralba. Virtualhome: Simulating household activities via programs. *arXiv preprint arXiv:1806.07011*, 2018. 1, 2

[16] A. Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008. 1, 2, 5

[17] S.-H. Sun, H. Noh, S. Somasundaram, and J. Lim. Neural program synthesis from diverse demonstration videos. In *Proceedings of the 35th International Conference on Machine Learning*, 2018. 2

[18] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, pages 1507–1514. AAAI Press, 2011. 2

[19] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene graph generation by iterative message passing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5410–5419, 2017. 8

[20] Y. Zhu, D. Gordon, E. Kolve, D. Fox, L. Fei-Fei, A. Gupta, R. Mottaghi, and A. Farhadi. Visual semantic planning using deep successor representations. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 483–492, 2017. 2