RoboFlow: A Flow-based Visual Programming Language for Mobile Manipulation Tasks

Sonya Alexandrova, Zachary Tatlock and Maya Cakmak

Abstract— **General-purpose robots can perform a range of useful tasks in human environments; however, programming them to robustly function in all possible environments that they might encounter is unfeasible. Instead, our research aims to develop robots that can be programmed by its end-users in their context of use, so that the robot needs to robustly function in only one particular environment. This requires intuitive ways in which end-users can program their robot. To that end, this paper contributes a flow-based visual programming language, called RoboFlow, that allows programming of generalizable mobile manipulation tasks. RoboFlow is designed to (i) ensure a robust low-level implementation of program procedures on a mobile manipulator, and (ii) restrict the high-level programming as much as possible to avoid user errors while enabling expressive programs that involve branching, looping, and nesting. We present an implementation of RoboFlow on a PR2 mobile manipulator and demonstrate the generalizability and error handling properties of RoboFlow programs on everyday mobile manipulation tasks in human environments.**

I. INTRODUCTION

Robots that can assist humans in everyday tasks have the potential to bring independence to persons with physical disabilities, enable older adults to age in place, and improve the quality of our lives. A key challenge in realizing such robots is to program them to robustly function in the end-users' unique environments. Useful robotic capabilities for mobile manipulators such as fetching items[1], baking cookies[7] or setting up the table[2], have previously been demonstrated; however, the way that these demonstrations are realized is not scalable. The reason is two-fold: (i) they only work in the particular environment they are developed for, and (ii) they require highly skilled developers experienced in robotics to program them. Most robotics research targets the first problem by aiming to develop universal or adaptive capabilities that will work in all possible scenarios. This is extremely challenging and has had limited practical success so far. Instead, we aim to address the second problem. Our goal is to develop robots that can be programmed by the end-users after they are deployed in their context of use.

To that end, we seek to apply techniques from the field of End-User Programming [5] (EUP) to robot programming. Although there are various techniques in EUP that are relevant for robot programming, the technique that has been most popular in robotics is Programming by Demonstration [5] (PbD). Our previous work explored the use of *program visualization* [26] in conjunction with PbD, to improve the users' mental model of what the robot learns from provided demonstrations [2]. In this paper, we explore the use of another powerful EUP technique called *visual programming* [10]. We develop a visual programming language for mobile manipulation tasks and propose a programming paradigm that involves graphical interactions to edit program structure and physical demonstrations to instantiate program procedures. We demonstrate the expressivity of the language in creating flexible programs that generalize across different scenarios on a set of real-world mobile manipulation tasks on a PR2 robot. We also evaluate program comprehension, creation, and debugging though a small scale user study that confirms the intuitiveness of the language.

The authors are with the Computer Science and Engineering Department, University of Washington, Seattle, WA 98195, USA.

[1]Beer me Robot: `http://youtu.be/c3Cq0sy4TBs`

[2]PR2 Sushi challenge: `http://youtu.be/NnfJUPz6__M`

II. RELATED WORK

End-User Programming (EUP) is an active research area in human-computer interaction that aims to enable everyday people, who are not professional software developers, to create custom programs that meet their particular needs [24], [21]. Popular examples of EUP include spreadsheets [27] and webpage development [34]. Research in EUP has produced many techniques such as domain-specific languages (DSLs) [25], programming by example [23] or model-based development [30]. This paper focuses on one such method called *visual programming* [26], [18], [10] which has had the most success in making programming accessible to non-technical users [16], [31]. Previously, the EUP technique that has been most popular in robotics is Programming by Demonstration (PbD) [4]. PbD allows users to program new capabilities on a robot by demonstrating the desired behavior [5], [3]. Most work in this area focuses on learning control-level skills represented by cost functions [1] or policies/controllers that map a state to an appropriate action [15], [8].

One line of work, motivated by the use of robots in programming education, produced a set of visual programming tools for toy robots [32], [33], [20]. The simplicity of the robots used by these tools allows programming at a low-level where individual sensory inputs can be tied directly to actuators. This impedes their applicability to general-purpose mobile manipulators like PR2. Another set of tools have been developed for animating articulated robots, such as the Aldebaran Nao[3] or the MIT Media lab magician robot [29]. These closely resemble a class of visual programming languages, but they are intended for open loop robot motions rather than mobile manipulation tasks that involve interacting with the environment.

[3]`http://www.aldebaran.com/en`

Another related line of work is has been focused on the use of software engineering methods, particularly around a series of workshops on DSLs for robotics. Some example languages from this community include Robot Scene Graphs [6] or a DSL for pick-and-place [9]. Work by Kress-Gazit et al. uses formal verification techniques for low-level robot programs [22]. Finally, most related to our system in terms of application and purpose, is a system developed by Nguyen et al. for the PR2 robot called RCommander [28]. While the functionalities provided by RoboFlow and RCommander are similar, RoboFlow has a simpler and much more restricted programming interface and is formalized as a programming language.

III. Visual programming for mobile manipulation

In visual programming users create or modify programs by manipulating a graphical representation of the program. A visual programming language (VPL) is a programming language in which visual expressions (*e.g.* spatial relationships between tokens on a 2D screen) have significance in the meaning of the program [10]. There are several types of VPLs, differing based on how they exploit visual expression, such as *form-based* [12], *flow-based* [19], or *rule-based* [16] VPLs. Existing VPL research greatly informs its application to new problem domains such as robotics. However, in developing a VPL for robot programming, we face the challenge of specifying a language that appropriately balances intuitiveness, scalability, and robust implementation on a robot. The approach we take with RoboFlow is to maximally constrain the language to keep it as simple as possible at the high level, to ensure intuitiveness. Nonetheless, the language needs to be expressive enough to capture useful tasks. We focus on a rich but structurally constrained task domain, explained in the following.

A. Task domain

RoboFlow targets tasks that involve configuring everyday objects within a known environment. A large set of organizing tasks in human environments (often specified with verbs such as *straighten*, *pick up*, *put away*, *organize*, *tidy*, or *clear out*) simply involve reconfiguring objects within the environment [13]. For example, the task *straighten counters* involves placing dirty dishes in the dishwasher, perishable food in the fridge, and clean dishes, tools, and condiments in their respective cupboards or drawers. Such tasks require a small number of low-level capabilities on a robot including identification, localization, and manipulation of objects and autonomous navigation. Nonetheless, there are many different such tasks and each task has a unique instantiation in every home. We exploit the common structure of these tasks to specify a compact and extensible VPL that allows programming unique programs tailored to a particular environment.

B. Language specification

Programming languages are specified by their *syntax* (form of the language) and *semantics* (meaning of the language). A visual programming language is a language whose semantically-significant syntax includes visual expressions [10]. VPLs exploit familiar visual representations to ensure intuitiveness; for example, arrows are often used to indicate flow of information in one direction. The VPL used in this paper is a *flow-based* VPL [19], similar to popularly known *flow diagrams* or *control flow graphs*. Based on the task structure that the language is intended to support (Sec. III-A), we propose the following syntax to keep the language as simple as possible.

- We use a box-line representation. Boxes are procedures or functions with inputs and outputs. Lines represent flow of data from the output of a box to the input of another box.
- We use a pure data flow model, with no control flow constructs such as *while* or *repeat*. We allow iterations/loops through cycles in the flow graph.
- We only allow selector functions, *i.e.* functions that have one input and multiple outputs.
- Outputs/inputs (*i.e. lines*) do not carry semantic information (they are on or off); instead semantics are embedded in the box structure.

The formal syntax and semantics of RoboFlow is shown and explained in Fig. 1.

An important decision in the design of the VPL is the choice of procedures (*i.e. boxes*) available to users. These need to ensure a robust implementation on the robot, while being intuitive for users. Based on the capabilities required for our task domain (Sec. III-A), we constrain available procedures to three types robot actions that independently control different groups of actuators on the robot: (i) manipulation, (ii) navigation, and (iii) active perception (*i.e.* head movements). Each procedure type has a fixed structure. The core is the *operation*, which is a low-level subroutine that interacts with the actuators. The operation may have *pre-conditions* to be checked before it is executed and *post-conditions* to be checked upon completion of the operation. This is a common action representation for robots in the planning literature [17]. The three types of procedures are as follows (illustrated in Fig. 2(a-c)).

1) Manipulation procedures: These actuate the robot's arms to interact with objects in the environment. We represent manipulation actions as a sparse sequence of end-effector poses relative to landmarks (objects or detectable markers). These actions are programmed by demonstration. Our previous work has demonstrated this simple representation captures a wide range of manipulation actions, from a simple pick-up-place, to complex bi-manual or constrained grasps and non-prehensile manipulation actions [2]. The pre-conditions for a manipulation procedure is that the landmarks involved in the action are present in the robot's view and that any pose relative to these landmarks is reachable be the robot. The post-condition checks whether the manipulation has succeeded. Failures can happen if the arm get stuck due to an obstacle or if the objects slip during manipulation. Both failures are detectable.

```
P ::= nil | gid { G } P
G ::= nil | nid -> N :: G

N ::= Op(O, nids)
    | Call(gid, nid, nid)
    | Success | Failure

O ::= Manipulate(params)
    | Navigate(params)
    | Look-at(params)
```

(a)

$$\frac{G[n] = Op(o, ns) \qquad [\![o]\!](\sigma, ns) = (\sigma', n')}{(n, (G, n_s, n_f) :: \Gamma, \sigma) \rightarrow (n', (G, n_s, n_f) :: \Gamma, \sigma')} \qquad \frac{G[n] = Success}{(n, (G, n_s, n_f) :: \Gamma, \sigma) \rightarrow (n_s, \Gamma, \sigma)}$$

$$\frac{G[n] = Call(g, n'_s, n'_f) \qquad P[g] = G' \qquad entry(G') = n'}{(n, (G, n_s, n_f) :: \Gamma, \sigma) \rightarrow (n', (G', n'_s, n'_f) :: (G, n_s, n_f) :: \Gamma, \sigma)} \qquad \frac{G[n] = Failure}{(n, (G, n_s, n_f) :: \Gamma, \sigma) \rightarrow (n_f, \Gamma, \sigma)}$$

(b)

Fig. 1: RoboFlow Formal Syntax and Semantics. The Backus-Naur Form grammar in (a) specifies RoboFlow programs as a list of graphs, each labeled by a graph identifier, *gid*. Each graph in turn is specified as a list of nodes labeled by a node identifier, *nid*. There are four types of nodes: *Op, Call, Success, Failure*. Each node includes the node identifiers of its potential successors. The small step operational semantics in (b) provides the meaning of RoboFlow programs as a set of logical inference rules specifying when one *state* can step to another. Each state comprises the current *nid*, call stack, and *configuration* which represents the state of the world and is denoted by σ. Note the use of an operation's denotation $[\![o]\!]$ in the *Op* case which both returns the updated configuration and successor node. This design choice modularizes RoboFlow with respect to the available procedures which keeps our semantics simple and provides extensibility for adding new procedures.

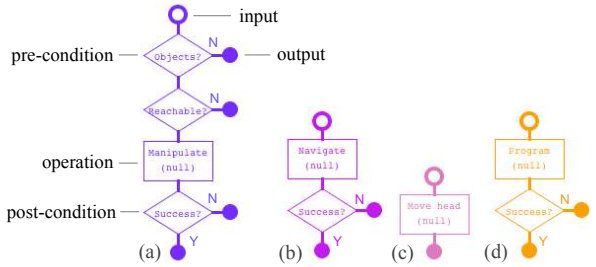

Fig. 2: Illustration of VPL tokens and structure of (a) manipulation, (b) navigation, and (c) active perception procedures; and (d) procedures abstracted from other programs.

2) Navigation procedures: These actuate the robot's base to make it move about its environment. They are parametrized by a target location. They have no pre-conditions but they have a post-condition that checks whether the destination was reached. Failures can happen due to obstacles that block the robot's path to its destination.

3) Active perception (look-at) procedures: These actuate the robot's pan-tilt head to direct its sensors towards different parts of the environment. They are parametrized by the pan and tilt angles and they have no pre-conditions. They have no post-conditions either, as they are guaranteed to succeed.

RoboFlow embeds conditions within a procedure. This is in contrast with general-purpose flow diagrams where the user directly manipulates conditions and operations and can connect them in arbitrary ways. Nonetheless, we chose to visualize the control flow within a procedure (with diamonds for conditions and rectangles for operations) to partially communicate the semantics of a procedure to the user.

C. Program definition and procedural abstraction

In addition to the procedures, a complete program requires a start terminal (a unit with one output only) and end terminals (a unit with one input only). We allow two types of end terminals: *success* and *failure*. The flow of a program is specified by edges between procedures and terminals, represented with arrows from an output an input. Note that only one edge can start at an output, but multiple edges can end at an input.

Parametrization of the procedures improves the scalability of our VPL [11]; however it necessitates giving users the ability to *instantiate* each parameter. The different types of procedures are instantiated through different interfaces described in Sec. III-D.

Given our language specifications, a valid program is a program that has (i) one start terminal, (ii) at least one success end-terminal, (iii) at least one procedure, (iv) for every output to have one outgoing edge, and (v) every input to have at least one incoming edge. This program definition allows *nesting* programs in other programs; a concept known as *procedural abstraction*. A program can be abstracted as a procedure simply by considering its start as input and its two types of terminals as its two outputs. Thus, a program is equivalent to a procedure with two outputs (Fig. 2(d)).

D. Programming interactions

VPLs naturally support creation and modification of programs through a graphical interface. However, a purely graphical interface does not exploit the physicality and situatedness of the robot. Our VPL nicely separates the portion of the programming process for which graphical interaction is crucial; that is, the specification of program structure. On the other hand, for the instantiation of procedures physical interaction with the robot can be more effective than a graphical interface. We propose two alternative programming interaction paradigms that aim to combine graphical and physical interactions in different ways.

1) Top-down programming: The first proposed approach is to have users start creating a program from scratch using a graphical interface. This interaction technique is akin to existing VPL development tools. Users add different procedures and terminals to a program pad, and create edges to specify program structure. At any time during this process,

users can instantiate a procedure that is in the program. The instantiation process is different for each type of procedure. For manipulation the user will demonstrate the sequence of poses (possibly relative to landmarks) by physically guiding the robot's arms and giving simple commands [2]. For navigation, the user drives the robot to the desired location using a joystick. For active perception, users specify the pan-tilt angle of the head by clicking on the robot's camera view to center target objects.

2) Bottom-up programming: The second approach involves the user first demonstrating one full execution of the program. This is used to automatically create a *default* program. The user then switches to the graphical program development environment, and edits the program structure. The default program is one that has all demonstrated procedures in a sequence and fails if any of the conditions are not met and succeeds when the last procedure completes successfully. A single demonstration can only create a linear program. Therefore, the provided demonstration should trace all parts of the program at least once. Structural edits can then change the program to repeat or skip certain parts of the default program. This approach becomes problematic if a program has two branches that cannot be traced in the same execution. In that case, users need to exploit procedural abstraction by creating separate programs for each branch and then nesting them into the main program.

Although users do not have control on the condition checking structure, on manipulation procedures, they can change the similarity threshold for matching objects in the scene to the object with which the manipulation action was programmed with. A higher threshold results in more object to meet the criteria, and hence return true when the condition is checked. This functionality is explored in Sec. IV-B.2.

E. Implementation

We implement RoboFlow for the PR2 research robot. PR2 (Personal Robot 2) is a mobile manipulator with an omni-directional base and two 7 degree-of-freedom (DoF) arms with 1-DoF under-actuated grippers. PR2's manipulators are naturally gravity compensated through a passive balance mechanism. This allows safe and comfortable kinesthetic interactions with the robot, during demonstrations of manipulation procedures. The implementation of manipulation procedures is based on the open-source PR2 Programming by Demonstration package[4]. Similarly, navigation procedures use existing autonomous navigation software[5]. The front-end editor of RoboFlow is implemented as a Java applet (Fig. 3).

IV. EVALUATION

Next, we present sample programs and executions of these programs in different environments to demonstrate the expressivity of RoboFlow and the generalizability of programs created with it. Then, in Sec. IV-C we present findings from a small scale user study investigating the usability and intuitiveness of RoboFlow.

[4]http://ros.org/wiki/pr2_pbd
[5]http://wiki.ros.org/pr2_navigation

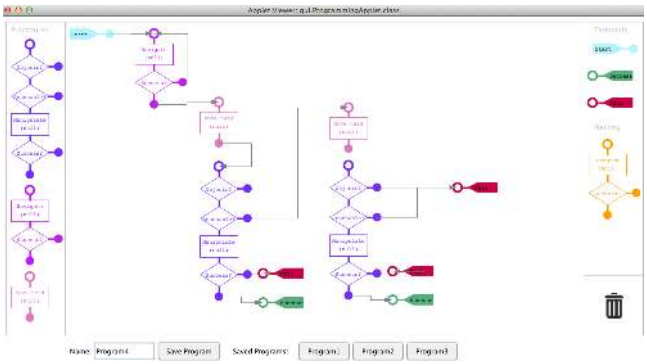

Fig. 3: Overview of the RoboFlow editor.

For the systematic evaluation we used the bottom-up programming approach (Sec. III-D.2). All tasks were programmed by one of the authors and tested in 3 to 5 different scenarios chosen to illustrate different traces of the program. For user studies, we assumed a top-down programming approach but the participants did not actually instantiate the procedures on the real robot. Rather, they chose procedures from a set of pre-specified alternatives that were described to them.

A. Analysis of expressivity

We first present RoboFlow programs that illustrate its expressivity over simple sequential programs that are common in robot PbD systems (*e.g.* [2]).

1) Looping: To demonstrate looping, we programmed two tasks. The first consisted of putting toy building blocks in a box. The demonstration used one block only (Fig. 4(a)), and the execution was tested with different numbers of blocks (Fig. 4(b-c)). The RoboFlow program for the task is shown in Fig. 8(a). The program starts with moving the robot head down, then looking for objects similar to the object used in the demonstration. If no such objects were detected, the program ends successfully. If the objects were detected, but are unreachable, the program fails. Otherwise, the robot manipulates the object as in the demonstration (in this case, puts the block in the box). If manipulation fails, the program fails too. Otherwise, the program loops back, *i.e.* the robot looks for objects again, an so on.

The second task consisted of stacking paper cups. The demonstration used two cups (Fig. 5(a)), the execution was again tested with different numbers of cups (Fig. 5(b-c)). The program is structurally exactly the same as for the previous one (Fig. 8(a)), the only difference is that now the robot looks for two objects that are similar to cups, instead of the small block and the large box.

2) Procedural abstraction: We illustrate the procedural abstraction with two examples. The first one involves stacking cups and putting them in a box. Since the first part was programmed before, this program could use that program as a procedure. The demonstration for this task consist of calling that procedure and then continuing on to demonstrate the second part of the task by picking up the the stacked cups and placing them into the box. This program is shown

Fig. 4: (a) Demonstration of the manipulation procedure for picking up a small block and placing it in a large box. (b-c) Execution of the looping program in the original scene it was demonstrated and a new scene with more objects. (d-e) Execution of the lopping program with different object matching condition thresholds.

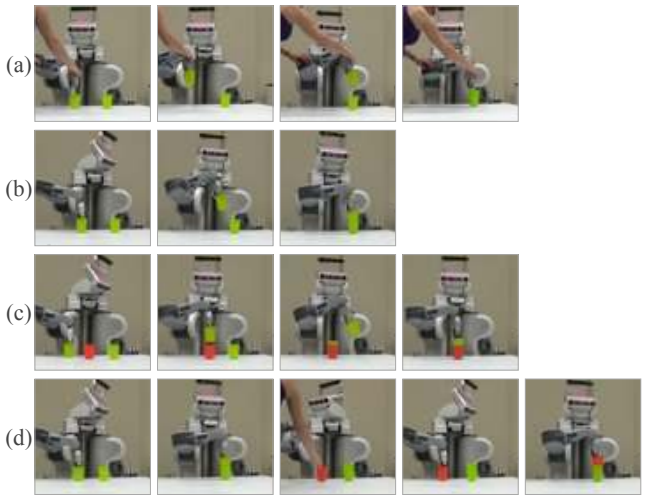

Fig. 5: (a) Demonstration of the manipulation procedure for stacking a cup on another one. (b-c) Execution of the corresponding looping program in the original scene and a new scene with three cups. (d) Execution in a dynamically changing environment (new cup inserted by user during execution).

Fig. 6: Two programs that re-use abstracted procedures (opening the drawer and closing the drawer) (a) putting an object into the drawer, and (b) taking an object out of the drawer.

in Fig. 8(b). For the second example we programmed two smaller tasks first: opening and closing a drawer. We then used those tasks as operations in two new tasks: putting a block in a drawer and taking a block out of a drawer, both starting and ending with a closed drawer. The execution of these two programs is shown in Fig. 6. We envision users creating a library of such reusable tasks, such as opening or closing drawers, cabinets and doors or pick up and placement of specific objects. These could later be reused in different programs that involve those objects.

3) Branching: A type of branching is already seen in the previous examples: if an object is found, perform manipulation, otherwise, end the program. We further explore the branching capabilities by programming a trash clean-up task. For this task, the robot navigates to a table and looks for paper cups. If one is found, the robot picks it up and navigates to the trash can, where it throws the cup away. After that, the robot proceeds to the next table. If a cup is not found or if it cannot be reached, the robot proceeds to the next table immediately, skipping several steps that require the presence of the cup. An execution of this task is shown in Fig. 7(a).

Another use case for branching is a *supported right grasp*, where the robot picks up a bottle if it is reachable with its right gripper but, otherwise, pushes it towards the right arm using its left arm. This program is also an example of a loop: as long as the left gripper can reach the bottle and the right gripper cannot, the robot pushes the bottle towards the right. The corresponding program is shown on Fig. 8(c) and different executions of the program are shown in (Fig. 7(b-c)). Yet another example of branching is the task of picking up a cup with different grasps (Fig. 7(d)). At first the robot attempts to pick up the cup from the side. If that fails, the robot tries to pick it up from the top. If the first grasp succeeds, the second manipulation procedure is not used.

B. Analysis of generalization

The structure of RoboFlow programs allows them to work robustly across different situations. In this section we reiterate the dimensions in which generalizability is supported by the capabilities demonstrated in Sec. IV-A.

1) Number of objects: Loops allow for generalization across different number of objects, as illustrated by the block and cup examples discussed earlier (Sec. IV-A.1).

2) Types of objects: Flexible conditions allow for generalization of actions programmed for one object to different objects. For instance, after programming the robot to pick up toy blocks, we edited the threshold for object similarity, allowing the robot to perform the same action on larger toy

Fig. 7: Programs that further illustrate branching: (a) picking up recycling from all tables, (b-c) *supported right grasp* where the robot uses the left arm to push the object towards the right arm if it is not directly reachable with the right arm, (d) adaptive grasp that tries an alternative grasp if the first tried one fails.

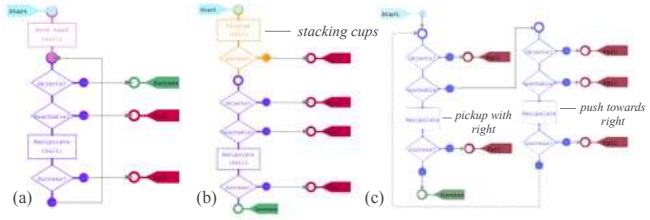

Fig. 8: Sample programs in RoboFlow: (a) picking up objects and placing them into a box in a loop, (b) a program that uses another program as a procedure (c) a program that picks up an object with the right arm, but is the object is not reachable, pushes it towards the right using the left arm within a loop.

blocks ((Fig. 4(d)), and even less similar objects such as a whiteboard eraser and a sponge ((Fig. 4(e)). Note that the robot does not alter its grasp, but the task is still performed successfully, because the grasp on all roughly rectangular objects of similar size is essentially the same.

3) Configurations of objects: RobotFlow has generalizability in different object configuration due to its object-centered representation of manipulation actions, based on [2]. RoboFlow further improves this generalizability by allowing alternative grasps (Fig. 7(d)) and by making an object graspable when it is not (Fig. 7(b)). Our system also accommodates the objects being present or absent at different locations. For example, we can have a program that searches for an object until it is found, or a program that goes through all locations and operates on the same objects in any location they occur (*e.g.* pick up and throw in trash).

4) Dynamic changes in the environment: The branching and looping allow to handle dynamic changes in the environment. For instance, in the stacking cups example, cups can be added to the scene in the middle of the task, and the robot will stack the new cups as well (Fig. 4(d)). Another example would be setting the table in collaboration with a human: human places the placemats, then robot places the plates and cups, then human places silverware, then robot pours water into the cups, and so on.

5) Error handling: The described capabilities of RoboFlow allow for error detection and recovery. Since each operation has a success/fail execution status, the following operations can use that in conjunction with branching to recover from errors. For example, if navigation fails because the planner could not find the route to the destination, the robot can be programmed to move a small distance to localize, reset the planner and try again. In manipulation, if a grasp fails, another grasp can be tried (Fig. 7(d)).

C. Analysis of intuitiveness

Lastly we verify the usability of RoboFlow as a programing language for robot tasks.

1) Procedure: We evaluate the front-end of RoboFlow with a common protocol for VPL evaluation [10], separately testing program *comprehension*, *debugging*, and *creation*. Participants are first introduced to the RoboFlow GUI through an example. The experimenter demonstrates adding procedures to the program and one by one introduces the three types of procedures explaining their operation. Participants do not actually instantiate procedures in the study but instead they choose one from the existing procedure instantiations (*e.g.* "pick up small object with left arm" or "look at the table"). The experimenter then demonstrates creating and editing links, and continues to create a complete sample program for the task shown in Fig. 7(d).

After answering question by participants, we move on to the comprehension task. In this part participants are shown a program (Fig. 8(a)) and are asked to describe how the program would behave.

The next task is debugging, where we give participants a program that is automatically created from a single demonstration in the bottom up programming approach. The desired program behavior is described and participants are told to modify the program so it would behave as intended. The desired program is the one shown in Fig. 8(c). The default program created by demonstration does not have the backwards edge that results in repeating the pushing procedure until the object is reachable by the right arm. So the participant needs to correct that edge so it loops back.

Finally, participants create a program from scratch for a described behavior. For this the task is to search for a particular object around the lab and pick is up when it is found. We record an audio and screen capture of the whole session for later analysis. At the end, we conduct a semi-structured interview with the participants to get their feedback on RobotFlow.

2) Findings: We conducted our user study with 9 participants (5 male and 4 female, ages 24-28). Three were roboticists who are proficient programmers, three were programming languages experts, and the last three were non-

TABLE I: Task metrics in the user study. The left entry in each column includes completion time (seconds), while the right entry includes the number of errors made in the task.

Participant	Comprehension		Debugging		Creation	
	time	*#err*	*time*	*#err*	*time*	*#err*
P1 (Robo)	110	0	130	0	235	1
P2 (Robo)	255	0	245	0	245	1
P3 (Robo)	115	0	170	0	505	0
P4 (PL)	95	0	135	0	240	0
P5 (PL)	160	0	235	0	190	0
P6 (PL)	130	1	205	0	205	0
P7 (EU)	315	0	675	0	888	2
P8 (EU)	120	0	252	1	364	0
P9 (EU)	116	0	126	0	468	0
Average	*157*		*241*		*371*	
St.dev.	*76*		*170*		*225*	

programmers. Although RoboFlow is intended mainly for non-programmers with diverse backgrounds, this initial evaluation included the other two extreme users types, as their insights can be valuable in improving the language before a larger scale usability analysis. Table I summarizes the measurements from the user study in all three tasks. We make the following observations.

a) Overall performance: After only a three to four minute tutorial on RoboFlow, all participants were able to complete each of the tasks in a few minutes, while making very few mistakes. The mistakes that occurred were as follows. One of the programming languages (PL) experts misunderstood the relationship between looking (changing head pan and tilt) and object detection for manipulation during the comprehension test. Two of the robotics experts (Robo) made small mistakes during the creation test where their program instructed the PR2 to continue searching for an object even after the specified task required failure. One of the non-expert programmers neglected the look-at procedures in parts of the program, while the other included a redundant pushing action in their program in the case where the object was already reachable at the start.

b) Difference between tasks: As expected, the Comprehension, Debugging, and Creation tasks were ordered by amount of time required to complete the task. The fact that the debugging task took less time than the creation task could be taken as evidence in favor of the bottom up programming approach (Sec. III-D.2). However, this comparison does not take into account the difference between the two programs in each task. Therefore it is not conclusive. It should also be noted that the creation task involves more mechanical steps (dragging procedures and terminals into the program and connecting them) whereas debugging involves just a few edits on an existing program. Most of the debugging task is actually comprehension. Hence, improvements on the editor that streamline mechanical tasks could reduce the gap between the two tasks.

c) Differences among user types: While the task metrics show fairly similar quantitative performance for the three groups, their approach to solving the various tasks was distinct. The robotics experts tended to describe their reasoning in real world terms as they worked the tasks, using phrases like "When the robot moves between tables, if it encounters an obstacle, then the whole task should fail". In contrast, the programming language experts often focused on invariants arising from the path constraints required to reach a certain procedure in the program graph, *e.g.* "At this point I know the object cannot be present since all paths to this node establish and maintain that invariant."

Even though the robotics and programming languages experts were good at programming with RoboFlow and found it intuitive, they were not in favor of using a visual programming language. During the interview, all six of them stated that they would prefer to write the same programs in a general-purpose programming language such as Python. In contrast, one of our non-expert participants (P8) who had taken a 10-week Python class, stated that she "very much prefers [RoboFlow] to Python." Her reasoning cited the usefulness of having a "visual analogue for code" in allowing her to assess whether the program was complete, *i.e.* all the conditions were handled.

V. DISCUSSION

One of the key limitations of RobotFlow is the absence of manipulable program state. Because the programmer cannot record state transitions, certain abstractions are difficult to implement. In particular, once control flow merges at a node, no later transitions will be able to distinguish between the various possible execution paths that could have led to that node. To overcome this limitation, users sometimes must copy entire parts of the graph and tweak only a few small parameters in leaf nodes. Such code duplication makes scaling programs up and maintaining changes more challenging. Furthermore, the lack of state means that each time a `Call` operation is executed, the caller must rely on the callees to maintain all crucial invariants, since the caller is unable to save necessary state that may be useful once the callee has returned.

An extension of RoboFlow with simple condition constructs would allow such state checks. However, instantiation of such conditions is an open challenge. In the case of precondition for manipulation, the conditional statements are based on the objects that are involved in the manipulation. In other words, manipulation demonstrations are one way to refer to objects detected by the robot. Allowing arbitrary conditionals that check whether an object exists or not would require another way for the user to indicate which of the objects currently visible by the robot is actually the one the robot needs to check in the future. A clickable visualization of the robot's view would be an intuitive option. Another extension for conditionals would be the ability to compare individual object properties rather than compare objects based on an overall similarity metric. This would allow the application of manipulation procedures to object

that are similar in relevant dimensions (*e.g.* object of similar height) rather than object that are globally similar.

Our user study yielded promising results; however, it is crucial that we evaluate RoboFlow with a larger population that has no programming experience. In addition, our user study did not involve the full system; it focused on the high-level program structure and left out the instantiation of procedures in the program. While our previous user studies [2], [14] have demonstrated the usability of PbD-based procedure instantiation, an end-user evaluation of the complete system would be valuable. Finally, we are interested in comparing the bottom-up and top-down programming approaches proposed in Sec. III-D as part of such a study involving interactions with the full system.

VI. CONCLUSION

This paper contributes RoboFlow: a flow-based visual programming language for mobile manipulation tasks. We describe the design of this language, in all aspects ranging from the choice of procedures to interaction modes, and we present an implementation on the PR2 robot. We demonstrate that generalizable RoboFlow programs can be created for a diverse set of mobile manipulation tasks, simply be demonstrating a trace of the program and modifying its structure in the RoboFlow editor. A preliminary evaluation with three different user groups demonstrates that RoboFlow can be quickly learned by people with diverse backgrounds, allowing them to quickly complete common robotics programming tasks with a low error rate.

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. Alexamdrova, M. Cakmak, K. Hsaio, and L. Takayama. Robot programming by demonstration with interactive action visualizations. In *Robotics: Science and Systems (RSS)*, 2014.

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

[4] C.G. Atkeson and S. Schaal. Robot learning from demonstration. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 12–20. Morgan Kaufmann, 1997.

[5] A. Billard, S. Calinon, R. Dillmann, and S. Schaal. *Robot Programming by Demonstration*, chapter 59. Springer, December 2008.

[6] Sebastian Blumenthal, Herman Bruyninckx, Walter Nowak, and Erwin Prassler. A scene graph based shared 3d world model for robotic applications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 453–460. IEEE, 2013.

[7] M. Bollini, J. Barry, and D. Rus. Bakebot: Baking cookies with the pr2. In *The PR2 Workshop: Results, Challenges and Lessons Learned in Advancing Robots with a Common Platform, IROS*, 2011.

[8] C. Breazeal and A. L. Thomaz. Learning from human teachers with socially guided exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2008.

[9] Thomas Buchmann, Johannes Baumgartl, Dominik Henrich, and Bernhard Westfechtel. Towards a domain-specific language for pick-and-place applications. *arXiv preprint arXiv:1401.1376*, 2014.

[10] M.M. Burnett. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 1999.

[11] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, 1995.

[12] M.M. Burnett and H.J. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):1–33, 1998.

[13] M. Cakmak and L. Takayama. Towards a comprehensive chore list for domestic robots. In *Proceedings of the 8th ACM/IEEE international conference on Human-robot interaction*, pages 93–94. IEEE Press, 2013.

[14] M. Cakmak and L. Takayama. Teaching people how to teach robots: The effect of instructional materials and dialog design. In *Proceedings of the International Conference on Human-Robot Interaction (HRI)*, 2014.

[15] S. Chernova and M. Veloso. Confidence-based policy learning from demonstration using gaussian mixture models. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2007.

[16] A. Cypher and D.C. Smith. Kidsim: end user programming of simulations. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 27–34. ACM Press/Addison-Wesley Publishing Co., 1995.

[17] R. E Fikes, P.E. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

[18] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[19] D.D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[20] S.H. Kim and J.W. Jeon. Programming lego mindstorms nxt with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pages 2468–2472. IEEE, 2007.

[21] A.J. Ko, B.A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 199–206. IEEE, 2004.

[22] H. Kress-Gazit, G.E. Fainekos, and G. J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[23] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[24] H. Lieberman, F. Paterno, M. Klann, and V. Wulf. *End-user development: An emerging paradigm*. Springer, 2006.

[25] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[26] B.A. Myers. Visual programming, programming by example, and program visualization: a taxonomy. In *ACM SIGCHI Bulletin*, volume 17, pages 59–66. ACM, 1986.

[27] B.A. Nardi and J.R. Miller. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories, 1990.

[28] H. Nguyen, M. Ciocarlie, and K. Hsiao. Ros commander (rosco): Behavior creation for home robots. In *IEEE Intl. Conference on Robotics and Automation*, 2013.

[29] D. Nunez, M. Tempest, E. Viola, and C. Breazeal. An initial discussion of timing considerations raised during development of a magician-robot interaction. In *HRI 2014 Workshop on Timing in Human-Robot Interaction*, 2014.

[30] F. Paterno. *Model-based design and evaluation of interactive applications*. Springer, 2000.

[31] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[32] F. Riedo, M. Chevalier, S. Magnenat, and F. Mondada. Thymio ii, a robot that grows wiser with children. In *Advanced Robotics and its Social Impacts (ARSO), 2013 IEEE Workshop on*, pages 187–193. IEEE, 2013.

[33] J.B. Weinberg and X. Yu. Robotics in education: Low-cost platforms for teaching integrated systems. *Robotics & Automation Magazine, IEEE*, 10(2):4–6, 2003.

[34] J. Wong and J.I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1435–1444. ACM, 2007.