A System for Situated Tangible Programming of Robot Skills

Yasaman S. Sefidgar[1], Sarah Elliott[1], Maya Cakmak[1]

***Abstract*— Challenges in robot programming pose a major barrier to robotic automation in small and medium sized enterprises. Skill-based programming interfaces can lower this barrier by encapsulating low-level functionality into high-level programming blocks, with parameters that non-experts can easily provide. We propose *situated tangible robot programming* within this paradigm to create robot programs by placing specially designed *tangible* blocks in the robot's workspace. Users create programs by sequencing action blocks that correspond to different high-level actions, such as pick and place, and instantiate the parameters of those actions using selection blocks that specify objects, locations, or regions in the robot's workspace. The robot detects the blocks and objects and compiles them into executable programs. We present a proof of concept implementation of this technique within pick-and-place task domain.**

I. INTRODUCTION

Customization marks a fundamental shift in manufacturing in recent years, where highly varied products are produced in small batches over shorter production life cycles. Traditional automation is too inflexible and costly to address the demands. In contrary, robots have the potential to serve the high mix and low volume requirements above, with programmability as their key advantage [1]. However, robot programming is notoriously complex and time-consuming. Despite its benefits, robotic automation is not yet affordable for many small and medium sized businesses.

Easy to learn and use robot programming tools hold the promise for large impact by lowering the barrier to robot programming, thus allowing non-experts to deploy robots for various tasks with short down times. As a result, robotics companies now place more emphasis on end-user programmability. For instance, ReThink's robots, Baxter and Sawyer, are advertised as being "simple to train, flexible, and re-deployable"; Franka Emika's upcoming robot, dubbed "everybody's robot," offers "visually intuitive programming."

Task-level programming enables a user-centered methodology to robot programming by hiding low-level complexity in high-level blocks [1]. Each block, typically referred to as a *skill*, performs an action to impact the robot's environment. Skills have parameters that can be defined through user interactions [2]. Different interaction techniques for various tasks have been explored in literature, e.g. kinesthetic operation for picking [3], natural language for assembly [4], or augmented-reality for welding [5]. In most cases, robotics experts carefully develop skills, and except for instantiating the parameters, end-users have no control over a skill.

[1] Allen School of Computer Science and Engineering, University of Washington, 185 Stevens Way, Seattle, WA 98195 USA {einsian, sksellio, mcakmak}@cs.washington.edu

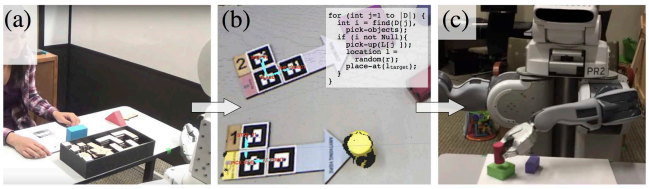

Fig. 1. *(a) Situated tangible programming* involves programming a robot by combining and placing specially designed tangible blocks in the robot's workspace to select objects, locations, or regions, and to specify actions and their ordering. (b) Blocks and objects in the workspace are detected by the robot and compiled into a robot program. (c) The robot can perform the instructed task in new environments and in the absence of blocks by executing this compiled program.

We propose situated tangible programming to allow end-users to create robot programs by sequencing existing skills and intuitively instantiating their parameters (Fig. 1). These skills take locations, regions, or objects as parameters and instruct the robot to perform actions relative to those parameters. We evaluated the intuitiveness and learnability of this technique for pick-and-place tasks in [6]. Here, we briefly review the specifics of situated tangible programming and then present a proof-of-concept implementation.

II. SITUATED TANGIBLE PROGRAMMING

We propose programming robots by placing tangible blocks in the robot's physical workspace to specify skills and their ordering and to instantiate skill parameters. Below, we present the programming language underlying this approach and describe the situated tangible programming process.

A. Skill-based Programming Language

A *program* is a sequence of skills. We focus on skills used for pick-and-place tasks in industrial settings. We consider fixed locations or regions, as well as single or multiple specific objects for pick or place as these are commonly featured in industrial manufacturing. We assume the robot can detect the surfaces, where locations, regions, and objects of interest can be found. The system supports the following four skills that take a *location*, a *region*, an *object*, or a *list of objects* as parameters: `pick-up-from-top`, `pick-up-from-side`, `place-at`, and `drop`. A *location* is a 3-dimensional point on the robot's workspace and a *region* is a convex hull. Objects are represented with a *descriptor* that allows them to be localized in the workspace. This language can be extended to additional skills (e.g. drilling or welding) and parameters (e.g. duration) without any changes in semantics.

B. From Blocks to Skills

A skill is formed by the following three types of blocks:

Selection blocks: These blocks *select* a location, a region, a single object, or multiple objects (Fig. 2). When placed in the environment within a skill, selection blocks specify a parameter of the skill (Fig. 3). A location on a surface is indicated by the tip of an arrow placed on the surface. A region on a surface is identified by two L-shaped corner brackets. A single object is selected by an arrow placed on the surface pointing towards the object. A group of objects is specified by enclosing them between two corner brackets.

Action blocks: Action blocks identify the type of skill (Sec. II-A, Fig. 2) and are always attached to a selection block that identifies the skill parameter (Fig. 3). If a `pick` or `place/drop` action block is attached to a location selector (Fig. 3(a,b)), the action is performed at that location. A pick action requires that an object be found at that location at execution time, whereas the place/drop action will adjust the height of the action depending on whether the location is empty or already has an object.

If a `pick` action block is attached to a region selector (Fig. 3(c)), the robot will pick up all objects found in that region at execution time. If a `place` action block is attached to a region selector (Fig. 3(d)), the robot will place the currently held object at a random location in that region.

If a `pick` or `place/drop` action block is attached to a single object selector (Fig. 3(a,c)), then the descriptor of that object is stored as the skill parameter. The program will locate this object in the new scene and perform the action at its location.

If a `pick` action block is attached to a multiple object selector (Fig. 3(d)), the robot will pick all instances of objects that are the same as an object in the selected set. If a `place/drop` action block is attached to a multiple object selector (Fig. 3(b)), the robot will find an object in the current scene that matches any of the selected objects, and will place/drop the currently held object on it.

Ordering blocks: Ordering blocks have positive integers and always attach to action blocks. They indicate the order of skills in the program. To compile a complete program from a set of blocks, our system uses the ordering constraints specified by these blocks, together with the constraint that each pick skill should be paired with a place/drop skill (Fig. 3).

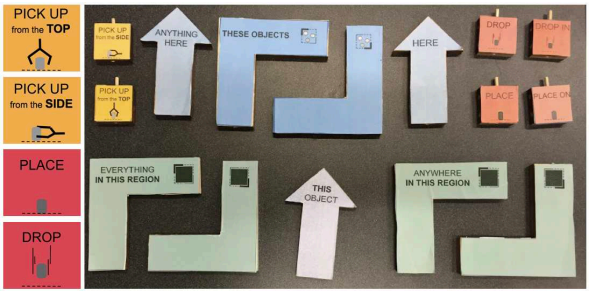

Fig. 2. Selection (arrows and paired brackets) and action (shown at scale on left for better viewing).

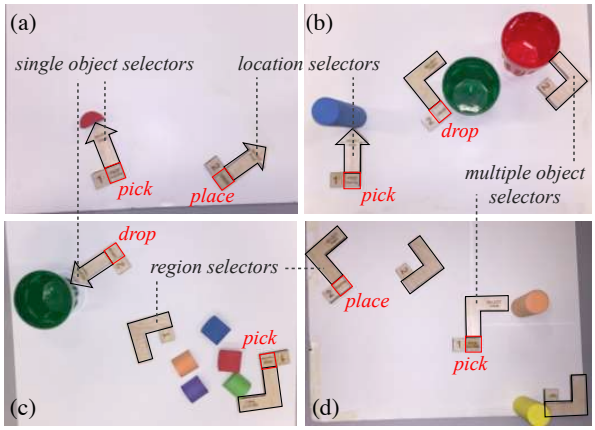

Fig. 3. Example situated tangible programs. Programs vary in the type of selection blocks used as part of the pick and place/drop skills.

III. SYSTEM IMPLEMENTATION

Situated tangible robot programming can be implemented on robot manipulator platforms with visual perception and basic contact control capabilities. We used a PR2 robot for our proof-of-concept implementation detailed below.

A. Hardware and Software

PR2 is a 14-DOF dual-arm robot (7 DOF's per arm) on an omnidirectional base. The 1-DOF grippers can open to a maximum of 9cm. We used the right arm only but the implementation can be extended to both arms. We assumed the robot is fixed in its workspace and provide no calibration procedure.

We implemented the system in ROS framework and used off-the-self libraries for basic perception (Alvar AR Tag tracking[1] and Point Cloud Library table top segmentation[2]) and motion planning (MoveIt![3]).

B. Architecture

Our system to realize situated tangible programming has three modes of operation: program creation, program execution, and idle. In program creation mode, the robot analyzes the scene to find objects and blocks to then form skills and instantiate their parameters. The skills are saved as a program to be run later in execution mode. When in neither creation or execution modes, the system is idle. There are a number of ways to control the mode of operation. We created *Edit* and *Run* tangible blocks that when placed in robot's view would activate program creation and program execution modes respectively. The system will enter idle mode if neither *Edit* nor *Run* block is present in the scene. Below, we describe implementation specifics of the two major modes of operation. Fig. 4 depicts the overall system architecture.

[1] `http://wiki.ros.org/ar_track_alvar`
[2] `http://pointclouds.org/`
[3] `http://moveit.ros.org/`

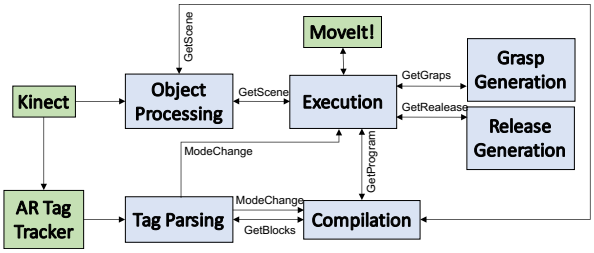

Fig. 4. Situated tangible programming system architecture. Blue boxes represent system nodes while green boxes represent functionality available by external packages. Single head arrows indicate publisher-subscriber connections with the arrow pointing to the subscriber. Double head arrows represent client-service relations. The Object Processing node processes point clouds from Kinect and provides a service for surface and object information. The Tag Parsing node publishes mode changes and provides a service for the semantic meaning of blocks. The Compilation node compiles tangible blocks and objects into skills that form a program and provides a service to return the compiled programs. The Execution node combines current scene information with compiled program, obtains grasp/release information for the object to pick/release by calling the relevant services, and turn this information to MoveIt! calls.

Program Creation: The Compilation node handles program creation. It first obtains a list of tangible blocks, objects, and surfaces supporting those objects from the Object Processing and Tag Parsing nodes. Perception of blocks is simplified using AR tags that are incorporated into the block design and allow us to determine both location and orientation of blocks. Each type of block is given a unique identifier. PCL table top segmentation returns the dominant surface in the scene and a list of segmented objects on it. In the simplest representation, objects are modeled by the size of their bounding box. Other more sophisticated object representations can additionally account for shape, color, and category of objects.

Given a list of tangible blocks, the Compilation node first forms skills by grouping selection, action, and ordering blocks based on their relative placement and distance. We have devised a peg/hole attachment mechanism on blocks to naturally enforce valid formation of skills. Next, the Compilation node instantiates the parameters of skills based on the type of their selection block. For *location* and *region* selectors, the specific location and region values define the parameter. For each *single object* selector, the Compilation node iterates over the list of objects to determine which object is in the direction of the selector arrow on the surface within a certain distance. For each *multiple object* selector, it iterates over the list of objects to determine which objects are within the convex hull created by the selectors on the surface. Selected objects define the parameter values.

While in program creation mode, the Compilation node constantly analyzes the scene to obtain the most recent version of the program. Upon mode change, the program is saved and can be retrieved later on in execution mode.

Program Execution: The Execution node manages program execution. It first obtains the program to execute by a service call to the Compilation node. It then loops through the program until the system exits the execution mode. The execution is resumed from where it was left off when operation mode changes to idle and then back to execution.

To execute each skill of a program, the Execution node first obtains scene information through Object Processing service. It then examines the objects against skill parameters and uses that information to generate appropriate motion by calling Grasp or Release Generation services. These services were written to generate simple grasps and releases by analyzing the scene geometry. In the future, such systems could incorporate external grasp or release generation software.

IV. EXAMPLE APPLICATIONS

Despite its simplicity, our proposed system for situated tangible programming allows the robot to complete important tasks. Fig. 5 shows two programs and how the robot executes each in a different scene. The one on top instructs the robot to pick a specific object (the green cylinder) and place it at the specified point (marked with a blue cross for clarity). When executing this program in a scene with several green cylinders, the robot stacks up the objects at the specified point. The program shown at the bottom instructs the robot to pick items from one region and place them in a different region (e.g. when removing items from a conveyor belt into a bin). Only the items within the region are manipulated.

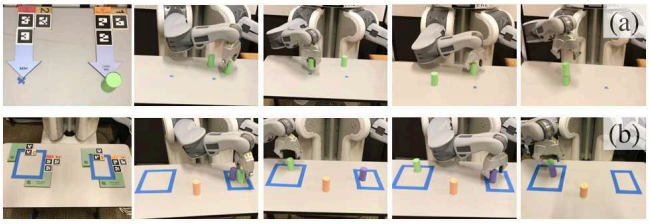

Fig. 5. Sample situated tangible programs and their executions. (a) The program instructs the robot to pick a specific object and place it at a certain point. The blue crosses in the images mark the tips of the arrows for clarity but are not used by our system. By repeatedly executing this program we can stack all green cylinders in the workspace. (b) The program instructs the robot to collect all objects in the right region and place them in the left region (e.g. when removing items from a conveyor belt into a bin). The borders of the regions are marked with blue lines for clarity only.

REFERENCES

[1] T. Lozano-Perez, "Robot programming," *Proceedings of the IEEE*, vol. 71, no. 7, pp. 821–841, 1983.

[2] M. R. Pedersen, L. Nalpantidis, R. S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen, "Robot skills for manufacturing: From concept to industrial deployment," *Robotics and Computer-Integrated Manufacturing*, vol. 37, pp. 282–291, 2016.

[3] R. S. Andersen, L. Nalpantidis, V. Krüger, O. Madsen, and T. B. Moeslund, "Using robot skills for flexible reprogramming of pick operations in industrial scenarios," in *Computer Vision Theory and Applications (VISAPP), 2014 International Conference on*, vol. 3. IEEE, 2014, pp. 678–685.

[4] M. Stenmark and J. Malec, "Describing constraint-based assembly tasks in unstructured natural language," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3056–3061, 2014.

[5] R. S. Andersen, T. B. Moeslund, O. Madsen *et al.*, "Intuitive task programming of stud welding robots for ship construction," in *Industrial Technology (ICIT), 2015 IEEE International Conference on*. IEEE, 2015, pp. 3302–3307.

[6] Y. S. Sefidgar, P. Agarwal, and M. Cakmak, "Situated tangible robot programming," in *Proceedings of the 2017 ACM/IEEE International Conference on Human-Robot Interaction*. ACM, 2017, pp. 473–482.