2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation

An IDE for Easy Programming of Simple Robotics Tasks

David Shepherd[1], Patrick Francis[1], David Weintrop[2], Diana Franklin[3], Boyang Li[1], and Afsoon Afzal[4]

[1]ABB Corporate Research
[2]University of Maryland
[3]University of Chicago
[4]Carnegie Mellon University

***Abstract*—Many robotic tasks in small manufacturing sites are quite simple. For example, a pick and place task requires only a few common commands. Unfortunately, the standard languages and programming environments for industrial robots are complex, making even these simple tasks nearly impossible for novices. To enable novices to program simple tasks we created a block-based programming language and environment focused on usability, learnability, and understandability and embedded its programming environment in a state-of-the-art robot simulator. By using this high-fidelity prototype over the course of a year in a case study, a user study, and for countless demonstrations we have gained many concrete insights. In this paper we discuss the details of the language, the design of its programming environment, and concrete insights gained via longitudinal usage.**

***Index Terms*—**

I. Introduction

In the past five years the landscape of the most common industrial robots, one-armed stationary robots (see Figure 1, has completely changed. The cost of robots has plummeted, smaller robots are being produced, and new safety measures have removed the need for protective fencing [1]. Due to these changes, many smaller labs and factories previously unable to consider robots have begun buying and integrating robots into simple, small-scale automation tasks. They often buy a single robot and program (and re-program) it to perform many different tasks in their context [2].

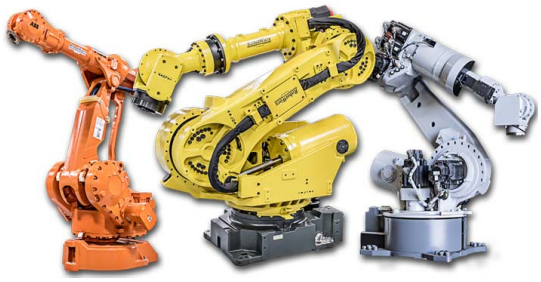

Fig. 1. Traditional industrial one-armed robots

Unfortunately these users are forced to use the same complex robot programming tools that robot programming experts utilize. Consider the relatively user-friendly industrial robot programming interface shown in Figure 2. In this example a single pick and place task has been implemented. For novices the combination of a low-level language, commands with obtuse names, and a confusing array of buttons makes this interface hard, if not impossible, to use without extensive training [3]. For instance, it would require weeks of ABB's own robot training courses to learn to program basic tasks.

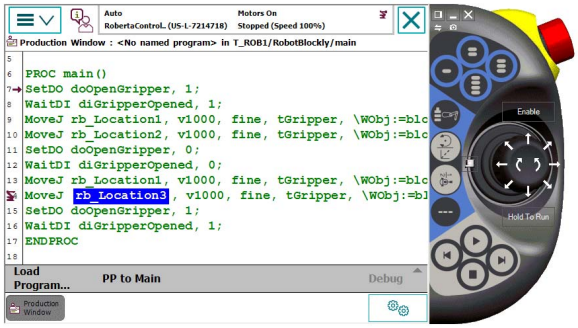

Fig. 2. An example industrial robot programming interface

While industrial users are struggling with traditional programming interfaces educational users have found great success with new programming interfaces based on blocks. As first introduced by MIT's Scratch language [4]. These languages are easy-to-read, easy-to-learn, and easy-to-modify. Many children learn to program using these languages and, increasingly, many educational robots (i.e., mobile robots without arms) can be programmed with them [5]. Building on the insights of educators, we decided to create a block-based programming language to program industrial one-armed robots. This allows for great simplification of robot programs–Figure 2 is shown in its block-based form in Figure 3–but it remains an open question if these block-based languages are powerful enough to implement real world robotics tasks.

In order to determine whether a block-based language could be used to automate realistic industrial tasks we implemented CoBlox, a plugin to RobotStudio, a professional grade integrated development environment for ABB's industrial robots. While we have previously introduced the concepts behind this approach [6] and reported on a user study with this approach [3] we have not, due to paper length constraints,

2470-6892/18/$31.00 ©2018 IEEE
DOI 10.1109/SCAM.2018.00032

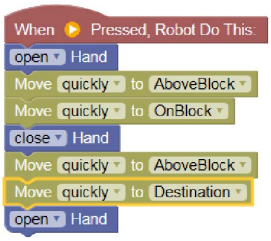

Fig. 3. A simple program in CoBlox

published the necessary conceptual and implementation details behind this approach that would be necessary for a full replication. In this paper we fully flesh out the approach. In Section 2 we provide an example usage. In Section 3 we provide an overview of the approach. In Section 4 we introduce all robot-specific commands, how they are used in CoBlox, and the generated RAPID (i.e., low-level) code. In Section 5 we discuss the aspects of CoBlox appreciated by users and we detail remaining challenges. In Section 6 we discuss the related work.

II. Tasks

Traditionally industrial robots have been used to perform precise tasks quickly and at scale. For instance, a typical application of ABB's industrial robots would be to perform precise welding on an automobile's frame as it moved down the assembly line. Traditional tasks like this are not well suited for novice programmers because they require a great deal of precision and because their cycle time, or time to complete a single task, must be absolutely minimized as thousands of parts are manufactured per year.

However, there are other tasks that are well-suited for smaller robots that can be programmed by novices. These tasks include pick and place and machine tending tasks, among others. A pick and place task is a simple as it sounds, where an object such as a raw material is, for instance, picked up and placed onto a conveyor belt, or a palette of objects are placed on a conveyor belt one at a time. These tasks are common in a manufacturing setting [7].

Another task that is simple but often necessary is that of machine tending [8]. Like a pick and place task, this task begins by picking up a piece of raw material, and is only different in that the material is placed inside a machine (e.g., a CNC machine), often to shape the part. After the machine has processed the material the part is retrieved and placed in the finished bin. This task often has subtasks such as opening the machine's door and pressing the start button, which do not add much complexity to the overall job.

III. Demonstration

Figure 6 shows an overview of the CoBlox programming environment. The program editor is on the left and the simulation environment is on the right. Here we detail the steps to program a pick and place task. For a video tutorial

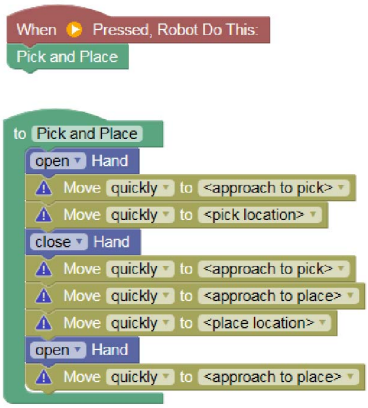

Fig. 4. A pick and place recipe prior to definition

of the environment see https://youtu.be/yQoxvIg1B-Q, for an example program run see https://youtu.be/HPak4tFeo38.

When using CoBlox to perform a pick and place task, the interface guides the user through the process. The user starts off by selecting a pick and place recipe from the Recipes drawer and dragging it into the program editor canvas. Once the user does this the canvas contains a recipe block as in Figure 4. Notice that at this point each Move command has a warning icon on its left, indicating that each location is undefined. To complete this pick and place task the user must define each location. To do this the user triggers the new location dialog by selecting New location from the Move command's dropdown, which is showing <approach to pick> in Figure 4. The user uses the positioning arrows shown in Figure 7 to move the robot arm to the correct position. Then the user chooses to name the location or uses the default name. In this case the user chooses to name his location AbovePick, referring to the fact that it was located above the object to be picked. The user then continues defining locations in his recipe in this way.

Once the user defines the second location notice that the next command is to close the gripper (i.e., the fourth command in the recipe), thus picking the block up. As part of the programming process the user presses the close gripper button in the simulation view, picking up the block. The user then continues defining locations. When the user reaches the location where he wants to place the block it becomes important that he has the block in his gripper. If the block were not in the gripper, then it would be difficult to gauge how high the gripper would need to be from the ground. With the block in the gripper, the user defines the location exactly as the object touches the floor. The user presses the open gripper button to release the block there.

Once defining all of the locations for the pick and place task the user simply presses play. Even though the block is now not at its original starting point, the play button triggers a reset of the workspace, which moves the block back and begins playing the program. Halfway through the program the

user decides that the AbovePlace location is not high enough off of the floor. He stops the run, uses the jump to location button to move to that location, and then modifies the location to be higher. Upon pressing play again the program works as expected, moving the block from its original location to the destination.

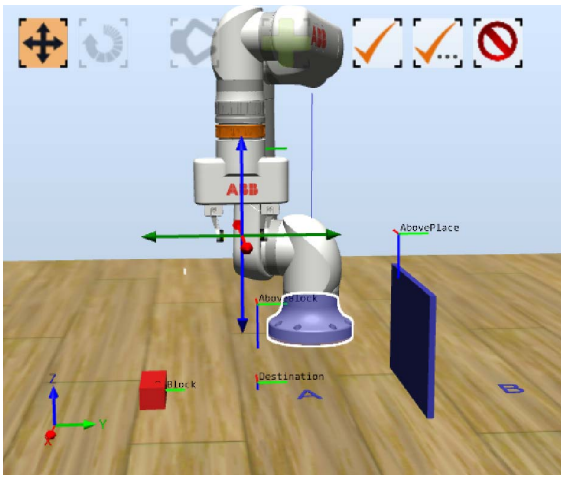

Fig. 5. Defining a location in the Simulation pane

IV. Overview

CoBlox is a block-based programming approach that allows users to program a one-arm, industrial robot. It hides complexity from the user by automatically translating high-level, block-based commands into low-level, robot-specific commands. As previously mentioned, an overview of the environment is shown in Figure 6. On the left is the code canvas and on the right is the simulation view.

A. Drag-and-Drop Commands

Following common block-based programming conventions CoBlox allows users to choose their commands from a variety of drawers, as shown on the left of Figure 6. In the Move drawer all commands related to moving the robot arm, in the Grip drawer are the commands related to opening and closing its gripper, and in the Loop commands are the commands for a loop. When clicking on a particular drawer, for instance the Loop drawer as shown in Figure 7, the drawer opens and shows all of the available commands in that drawer, in this case all three types of loops. The user can drag any command from this drawer directly to the main program and snap it into place.

There are several advantages to this approach. First, users cannot create syntax errors, as they are not entering text. Given these guardrails, users feel much more free to explore, often adding commands to their program simply to see what they do. Second, users do not have to memorize or even remember commands, they are laid out in categories simply to be dragged and dropped into place.

B. Simulation Management

In order to use CoBlox users need to be able to manipulate the simulation view. For instance, once they have their virtual world set up, with whatever objects they intend to interact with, they should use the Take Snapshot button to record their locations. Then, after moving these objects around during a test run, they can either Restore Snapshot or return all objects to their original locations. Note that resetting object locations is automatically managed when using the Play button.

In addition to managing the location of objects, users need to define locations for the robot arm. This can be triggered either via a Move command or in the simulation pane directly. Once triggered, the simulation pane will become in focus and users are required to position the robot arm using the positioning arrows as shown in Figure 5. Once positioned the user can either use the default name or they can customize the location's name. Note that once a location is defined users can easily jump to them using the Jump to Location button, which is useful for modifying a position or for defining a nearby position.

Another functionality that is required for the Simulation view is the open/close gripper button. This allows the user to close the gripper without running a program. This is useful when creating a series of commands to pick up an object and set it down, as the position of the object in the gripper becomes important when setting it down.

Note that all of these commands together give the programming environment a sense of **liveness**. Programmers can see exactly where locations are in the simulated world, pick up objects, and move them. This always on, live feedback, allows them to test their code as they create it.

V. Commands

While a block-based programming approach itself helps to simplify the industrial robot programming experience, the choice of available commands and their semantics has a large affect on the usability. For instance, creating a one-to-one mapping with all low-level RAPID commands would not simplify the experience. In this section we discuss each robot-specific block we introduced and its mapping in low-level code.

The most basic command we introduce is the Move command, as shown in Figure 8. This command moves the robot's arm to the designated position. While this command does have a nearly one-to-one mapping from CoBlox to RAPID, the CoBlox command hides complexity and eliminates unnecessary choices. There are about twenty different move commands in RAPID, and so providing the appropriate choice (MoveJ) and the appropriate parameters, of which there are five, dramatically improves usability. The single move command in CoBlox maps to the following code in RAPID.

```
CONST robtarget rb_Location2:=[[408.246120787,
    -137.507100021, 270.131300256],
    [0.000292855000009545, -0.0097450920031381,
    -0.99994149832215, -0.00468480000150642], [-1,
```

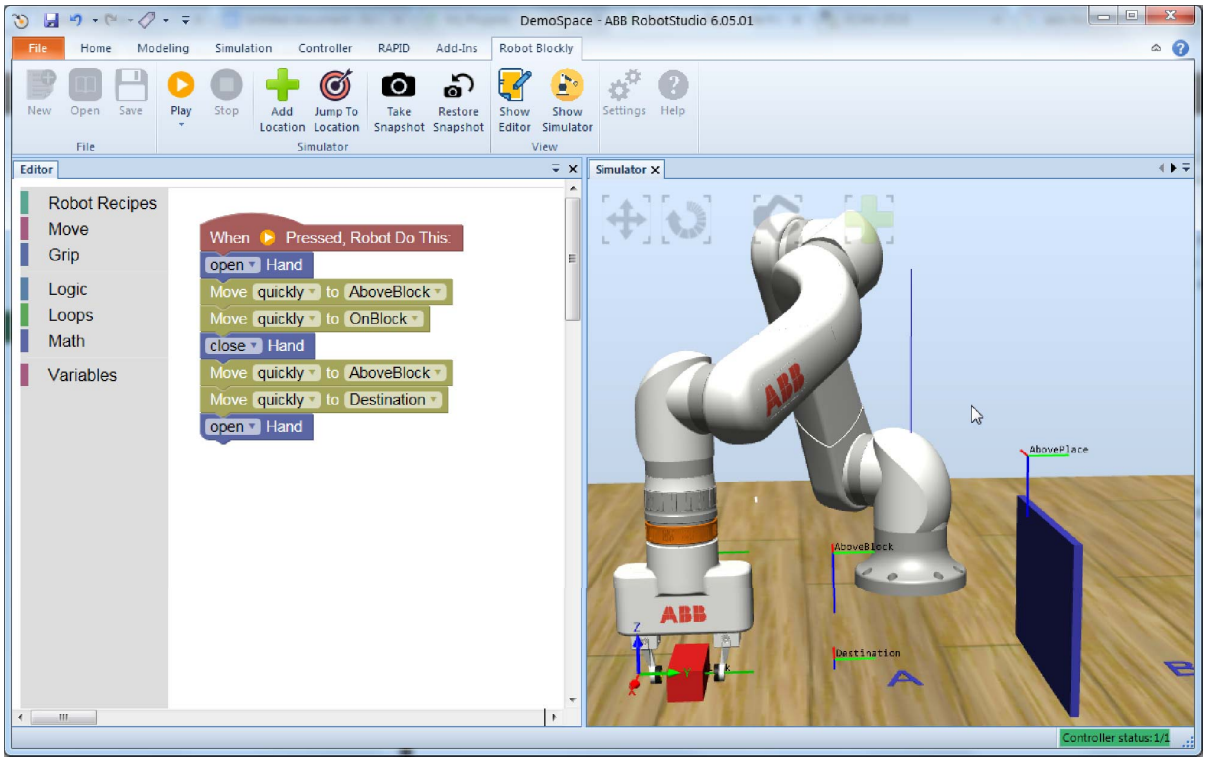

Fig. 6. The CoBlox Programming Environment

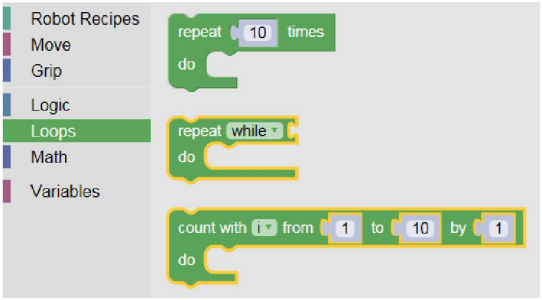

Fig. 7. The Loop drawer when opened

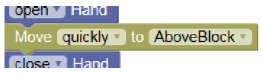

Fig. 8. A Move Command

```
    0, -1, 0], [9000000000, 9000000000, 9000000000,
    9000000000, 9000000000, 9000000000]];
MoveJ rb_Location2, v1000, fine, tGripper, \WObj:=
    blocklyWobj_4;
```

Another command we introduce is the open/close hand command. This command, shown in Figure 9 opens or closes the robot's gripper, allowing it to pick up or place down objects. Depending on the gripper being used this command will be translated down to one line or many lines of low-level code. While this code changes dramatically depending on the grippers used, and thus it is impossible to provide a single mapping, we provide an example mapping (for our virtual-world grippers) in the listing below.

```
SetDO doOpenGripper, 1;
WaitDI diGripperOpened, 1;
```

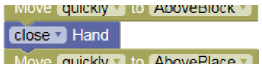

Fig. 9. A Close Command

Another command type we introduce are Recipes. These predefined, method-like structures allow us to bake domain knowledge from robot programs into the template. For instance, in Figure 10 the steps in a normal pick and place task have been added to the recipe. To use this command the user simply defines the locations specified in the steps. This helps users avoid common novice errors, such as not picking up an object before moving it to its new location. These recipes are well-understood by users and abstract away the potential complexity of defining methods in a traditional text language. Below we show the code generated from a pick and place recipe (note: location definitions omitted for brevity).

```
PROC main()
 Pick_and_Place;
ENDPROC

PROC Pick_and_Place()
```

```
    SetDO doOpenGripper, 1;
    WaitDI diGripperOpened, 1;
    MoveJ rb_Location1, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
    MoveJ rb_Location2, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
    SetDO doOpenGripper, 0;
    WaitDI diGripperOpened, 0;
    MoveJ rb_Location2, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
    MoveJ rb_Location3, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
    MoveJ rb_Location4, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
    SetDO doOpenGripper, 1;
    WaitDI diGripperOpened, 1;
    MoveJ rb_Location5, v1000, fine, tGripper, \WObj
        :=blocklyWobj_4;
ENDPROC
```

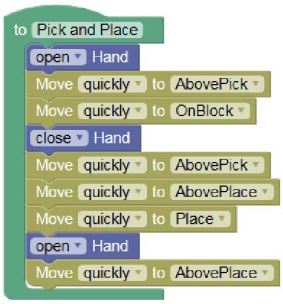

Fig. 10. A Recipe Command

VI. Discussion

Over the course of our user studies, pilots, and extensive internal usage users have given feedback on the positive and negative aspects of this approach. First, we briefly discuss the positive aspects. Then, with an eye towards researchers building upon our work we discuss the many open issues discovered once a working prototype was available for realistic testing.

A. Positives

The positives of this approach are often a direct result of the program representation. The programming environment creates programs that are easy-to-read, the environment itself is easy-to-use and easy-to-learn, and users are satisfied with it [3]. We found that users especially appreciate the decisions they no longer have to make (e.g., which of the 20+ move commands to use) and the simulation controls such as opening and closing the gripper help them interact with the virtual world easily when programming. Note that the significant positives are discussed in detail in earlier work [3], [6].

B. Challenges

In light of the many positives reported by users the challenges that remain open become even more interesting; if the few remaining issues can be solved perhaps this approach could be used in a real world setting. Because we currently do not have a sense of which issues are the most pressing these issues are discussed in random order.

One of the most appreciated features is location naming. While locations do not have to be named (i.e., they will automatically be named Location1, Location2, ..), users that do name their locations create much more readable programs, adding to a key advantage of our approach. However, naming locations can be challenging. While distinct locations such as "above the box" or "around the handle" work well, locations on a long path are difficult to name (e.g., Weld-Path-Point-20?), as are locations with no clear reference point. More thought needs to be put into how to name variables that represent physical locations in a systematic manner.

Related to this point, the number of locations in a given program can quickly become quite large; a common pick and place task requires a minimum of four locations. Because of the unscoped nature of the blockly language, even a few tasks in a program can lead to major usability challenges. For instance, when more than fifteen locations are defined the dropdown menu in the Move command used to select locations often scrolls off the canvas. Thus, the environment will need to be updated to include additional scoping or an improved user interaction to avoid this issue.

While naming locations is beneficial the user experience of naming locations is awkward. When using CoBlox users almost entirely use a mouse (or a finger, if using a tablet). However, each time they name a location they are forced to return to the keyboard or use a virtual keyboard. While current usage is often on a laptop, minimizing this issue, we envision this being used primarily on a tablet in the future and thus there will be even more of a need to avoid keyboard input.

One feature that has worked well for users is chunking via recipes. That is, for long tasks that involve multiple pick and place tasks the users who implemented them via recipes were generally more successful. However, using several recipes (similar to methods) quickly led to layout challenges. Where should the recipe definitions go? Should they be laid out in a grid-like manner or not? Once several recipes were on the canvas in any layout lots of scrolling was required to view the entire program. This evidence suggests that a more systematic and predictable way of laying out this code would be useful.

A related issue that has challenged users is that there is no generic recipe (or method) available to them. As many users found it useful to chunk their program into subtasks they removed the contents of the provided recipe and added their own content, effectively creating generic methods for the language. While this direction appears promising we will need to be careful not to introduce too much complexity for novices who may not understand the concept of methods.

Another related issue that has challenged users and stretched the implementation is that of scale. If too many blocks are used in a single program users have trouble keeping all relevant information on the canvas. At the same time the implementation of blockly, written in Javascript, struggles to render large numbers of blocks, especially as they are being dragged into place. Blockly was simply not designed to handle

large programs. In the future, it may be necessary to split the program into pieces and only render part of it at a time to avoid rendering delays.

A final challenge involves the decision to add (or not) more complex commands. While in some cases additional commands, like a MoveL which moves the robot arm in a straight line from point A to B, would be extremely useful. However, adding functionality adds complexity, taking away from the ease-of-use advantages of the current language. Only extensive testing with real world tasks can determine whether these additional commands are truly needed.

VII. Related Work

Our approach was initially inspired by the foundational work of the MIT Scratch team, who introduced the block-based programming paradigm [4]. However, it was also inspired by the countless toy vendors who utilized block-based programming to drive their toy robots, Wonder Workshop's Dash and Dot robots being prime examples of this [5]. Our work differs from Scratch in that it is trying to stretch the applicability of block-based programming from relatively trivial tasks to realistic tasks. Our work differs from toy vendors' offerings in that we are programming an industrial, one-armed robot instead of mobile robots, which these vendors favor.

There have been many attempts at making industrial robots easier to program via graphical or end-user languages over the years. MORPHA used a control-flow like representation with icons to communicate each nodes' meaning [9]. Our approach differs because it relies on natural language instead of iconography, as iconography quickly becomes confusing when trying to represent robot movements. Commercial companies like Rethink Robotics have also been active in this area. Rethink offers a programming environment based on behavior trees, which has been shown to be useful for video game creators to specify bot behavior [10]. Our approach is different because its representation is meant to be readable without any prior knowledge; their approach requires an understanding of behavior trees and their semantics, which can be challenging for novice users.

In addition to other approaches this work builds upon our own previous work on the topic. In a recent workshop paper we provided an initial description and small case study of CoBlox [6]. This paper differs in its description of the environment in that it provides implementation-centric details (e.g., the mapping between CoBlox commands and RAPID commands) and that this paper's lessons learned are from over a year of user studies and self-usage instead of a single user study. In a recent conference paper we published a large user study that compared CoBlox against state-of-the-practice approaches [3]. This paper differs from that one in that it provides a much more detailed description of the environment and it focuses on improvements that could be made to CoBlox instead of a performance comparison against different approaches.

VIII. Conclusion

Industrial robot programming is currently, in practice, a task only for experts. We have created a block-based programming environment called CoBlox that dramatically lowers the barrier to entry. While we have previously discussed the use of this prototype in the context of a case study and user study we add to this our own (and others') longitudinal experience. We discuss many new identified challenges which were only possible to find because of the hands-on experience that this robust prototype afforded us.

References

[1] P. Hollinger. (2015, May) Meet the cobots: humans and robots together on the factory floor. [Online]. Available: https://www.ft.com/content/6d5d609e-02e2-11e6-af1d-c47326021344

[2] R. Harbour and J. Schmidt. (2018) Make way for cobots, the collaborative robots. [Online]. Available: http://www.brinknews.com/make-way-for-cobots-the-collaborative-robots/

[3] D. Weintrop, A. Afzal, J. Salac, P. Francis, B. Li, D. C. Shepherd, and D. Franklin, "Evaluating coblox: A comparative study of robotics programming environments for adult novices," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, p. 366.

[4] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[5] W. W. Inc. (2018) Wonder workshop educational materials. [Online]. Available: https://education.makewonder.com/

[6] D. Weintrop, D. Shepherd, P. Francis, D. Franklin, L. Gusukuma, D. Kafura, A. C. Bart, R. Holwerda, F. Hermans, D. Rough *et al.*, "Blocks in new domains."

[7] A. Skoglund, B. Iliev, B. Kadmiry, and R. Palm, "Programming by demonstration of pick-and-place tasks for industrial manipulators using task primitives," in *Computational Intelligence in Robotics and Automation, 2007. CIRA 2007. International Symposium on*. IEEE, 2007, pp. 368–373.

[8] B. Rooks, "Machine tending in the modern age," *Industrial Robot: An International Journal*, vol. 30, no. 4, pp. 313–318, 2003.

[9] R. Bischoff, A. Kazi, and M. Seyfarth, "The morpha style guide for icon-based programming," in *Robot and Human Interactive Communication, 2002. Proceedings. 11th IEEE International Workshop on*. IEEE, 2002, pp. 482–487.

[10] M. Colledanchise and P. Ögren, "Behavior trees in robotics and AI: an introduction," *CoRR*, vol. abs/1709.00084, 2017. [Online]. Available: http://arxiv.org/abs/1709.00084