Design and Evaluation of a Rapid Programming System for Service Robots

Justin Huang
Computer Science & Engineering
University of Washington, Seattle, WA
Email: jstn@cs.washington.edu

Tessa Lau
Savioke, Inc.
Santa Clara, CA
Email: tlau@savioke.com

Maya Cakmak
Computer Science & Engineering
University of Washington, Seattle, WA
Email: mcakmak@cs.washington.edu

***Abstract*—This paper introduces *CustomPrograms*, a rapid programming system for mobile service robots. With *CustomPrograms*, roboticists can quickly create new behaviors and try unexplored use cases for commercialization. In our system, the robot has a set of primitive capabilities, such as navigating to a location or interacting with users on a touch screen. Users can then compose these primitives with general-purpose programming language constructs like variables, loops, conditionals, and functions. The programming language is wrapped in a graphical interface. This allows inexperienced or novice programmers to benefit from the system as well.**

We describe the design and implementation of *CustomPrograms* on a Savioke Relay robot in detail. Based on interviews conducted with Savioke roboticists, designers, and business people, we learned of several potential new use cases for the robot. We characterize our system's ability to fulfill these use cases. Additionally, we conducted a user study of the interface with Savioke employees and outside programmers. We found that experienced programmers could learn to use the interface and create 3 real-world programs during the 90 minute study. Inexperienced programmers were less likely to create complex programs correctly. We provide an analysis of the errors made during the study, and highlight the most common pieces of feedback we received. Two case studies show how the system was used internally at Savioke and at a major trade show.

I. Introduction

CustomPrograms is a programming system for mobile service robots, designed to enable roboticists and end-users alike to rapidly prototype and customize applications for the robot. Using our system, a programmer can make the robot perform a wide range of applications. Our goals for the system were 1) to be expressive enough for users to make new behaviors they wanted, and 2) to be easy to use for inexperienced programmers. We designed and implemented a simple API for the robot's capabilities, which we refer to as *primitives*. To make programming easier for inexperienced users, we provide a graphical interface for making programs, shown in Figure 1. However, users have access to standard programming language features, giving them control over the logic and flow of their programs. This interface is web-based and hosted on a cloud web server, allowing users to create, edit, and run programs without special software.

Our system can be thought of as a compromise between simple interfaces designed for end-users, and programming in a fully expressive, general-purpose programming language. Our design was motivated by the fact that for commercial

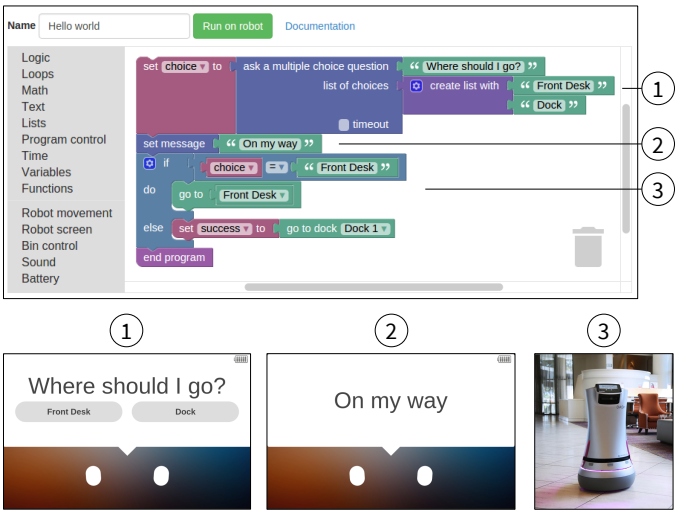

Fig. 1: A sample execution of *CustomPrograms*. Steps 1 and 2 affect the robot's touch screen, as shown. The Relay robot is shown driving in step 3.

service robots, there are many types of users, with a wide range of technical ability. On one end of the spectrum, there are staff members who might not have any programming experience, but who want an easy way to customize how the robot interacts with their guests. On the other end of the spectrum, there are experienced robot programmers. Our system allows roboticists to quickly prototype new applications for commercial use, without needing to write, build, or deploy code. In the middle, there are users with some programming experience, such as designers, sales people, and hotel technicians. We designed our system with this diverse range of users in mind.

Having a tool like *CustomPrograms* will be important as more service robots are developed and deployed. This paper contributes the design and evaluation of the *CustomPrograms* system and our choice of primitives, as well as our experiences deploying it on the Savioke Relay, a service robot. We present results from interviewing Savioke employees about new use cases for their robot, both internally and for their customers. We also conducted a user study with people new to the interface. In our evaluation, we report results on the system's ease of use and expressiveness. Finally, we present two case studies on how *CustomPrograms* was used in practice.

II. Related work

A. End-user programming

Researchers are actively studying end-user programming (EUP) [1], [2], with the goal of enabling ordinary people with no software development experience to write programs. Although our system is not primarily for end-users, many of the methods and design guidelines [3], [4], [5] for EUP could apply to creating a rapid programming system.

One such method is to represent the flow of the program or of the input data as a flowchart diagram [6]. Some commercially used examples of this include LabView and Simulink for engineering analysis, RapidMiner for data analysis, and Salesforce Process Builder for corporate processes. Flowchart-like languages have also been applied to programming robots [7], [8]. However, we chose not to use similar flowchart metaphors because we wanted the full expressivity of a general-purpose language. For example, it would be difficult to store high-dimensional state in a typical flowchart, which grows exponentially with the size of the state space.

Other forms of end-user programming include programming by demonstration, in which the program is inferred from examples [9], [10]; trigger-action programming, which trades off expressivity for high ease of use [11], [12]; and domain-specific languages, which are specifically designed languages for a limited set of tasks [13], [14].

B. Visual programming

Our system is most closely related to visual programming languages for robots. While our system runs on a subset of JavaScript, the syntax is wrapped in a visual editor called Blockly [15], shown in Figure 1. Blockly has been used in several educational applications, such as App Inventor, Hour of Code, and Made with Code[1].

Visual programming interfaces have been used for programming robots in educational settings, most notably with Lego Mindstorms [16], [17]. Of these educational interfaces, our system bears closest resemblance to that designed by Wonder Workshop for its line of Dash and Dot robots[2]. Our system, however, is not for educational purposes, and includes programming language features like functions and lists that are eschewed by these interfaces. Visual programming is also used for applications outside of robotics. Scratch [18] and Alice [19] are both examples of using visual programming to make animations and interactive applications.

C. Primitives and robot architecture

This paper presents the choice of primitives and the implementation of the runtime system for *CustomPrograms*. In previous work, researchers have attempted to organize robot capabilities into primitives, such as in CARMEN [20]. However, the open-source robotics community has since moved towards other robot-agnostic middleware such as Player/Stage [21] and ROS [22]. The primitives presented in this paper were not designed to work for all robots, but developing such an implementation is an idea for future work (Section VII-B).

[1]https://developers.google.com/blockly/about/showcase

[2]https://www.makewonder.com/apps/blockly

III. System overview

A. The Relay robot

We designed *CustomPrograms* for the Savioke Relay robot, shown in Figure 1. However, our system could be applied to any mobile robot with autonomous navigation and a touch screen. The Relay robot is approximately 3 feet tall and weighs 100 pounds. It can autonomously navigate indoor environments, using a LIDAR, 3D sensors, and several sonar sensors. The front of the robot features a 7-inch touchscreen display, which can be used to show information and receive user input. The robot has a bin with 0.75 cubic feet of storage, covered by a lid which locks when closed. It can connect to the internet to call elevators and phones. The robot also has a docking station, where it can go to charge its battery.

The Relay's main commercial use case is delivering items to rooms in hotels. When the hotel staff receive a delivery request from a guest, they enter a PIN number on the robot. They then enter the room number to deliver to, load the bin, and send the delivery. The robot travels to the guest room, riding an elevator if needed. At the guest door, the robot calls the room phone. When the guest answers the door, the robot opens the bin. The robot also asks guests how their stay is. Finally, the robot leaves and goes back to its docking station.

B. Primitives

Each primitive robot behavior is encapsulated in a function call. The primitives are described below.

1) Robot movement: Table I shows a list of primitives related to indoor navigation, a core capability for service robots like the Relay.

Name and arguments	Returns
goTo(string location)	void
shimmy()	void
move(number forward, number right)	void
turn(number degrees)	void
distanceTravelled()	number

TABLE I: Primitives related to robot movement.

The **goTo** primitive makes the robot navigate to a location named with a string ID. These IDs are human-friendly names, like "Front Desk" or "Room 201." We assume that the robot has an existing mechanism for building maps and labelling locations with names. Users can provide the location ID manually, generate it programmatically, or use a helper block that lists all the location IDs the robot knows about.

The **shimmy** primitive is a short side-to-side swaying, which gives the robot a way to convey happiness. The **move** and **turn** primitives move the robot relative to its current position, allowing it to go to unnamed locations on the map.

2) User interaction: Another important capability of the robot is the ability to interact with people. Our user interaction primitives are shown in Table II. We include primitives not just for displaying messages on screen, but also for receiving user input, e.g., by asking a multiple choice question. Asking for

Name and arguments	Returns
displayMessage(string text, number timeout=0)	void
askMultipleChoice(string text, string[] choices, number timeout=0)	string
askPasscode(string text, string passcode, number timeout=0)	bool
askNumber(string text, number timeout=0)	number
askRating(string text, number numStars, number timeout=0)	number
askScale(string text, number min, number max, string minText, string maxText, number timeout=0)	number
playSound(string sound)	void

TABLE II: Primitives related to user interaction.

a PIN number enables applications that require user authentication. The robot can also ask survey questions like star or scale ratings, which makes the robot capable of information-gathering applications.

The robot does not talk, but plays beeping and whistling sounds with the **playSound** primitive. This helps to manage expectations of the robot [23]. This was confirmed by Savioke designers in personal communications with the authors.

3) Battery and bin: The last set of primitives, shown below, relate to the robot's battery and bin.

Name and arguments	Returns
goToDock(string dock)	bool
batteryPercentage()	number
isCharging()	bool
raiseLid()	void
lowerLid()	void
isLidOpen()	bool

TABLE III: Other primitives for the robot.

We included the **goToDock** block for making the robot go to a docking station autonomously. When combined with the battery percentage primitive, this enables programs to run the robot for a long period of time, charging its battery as needed.

C. Error handling

Error handling is an important topic when designing APIs. However, we wanted users to be able to focus on their programs, and assume that the primitives work as described. The only primitive that we offered error handling for was **goToDock**, which returns false if docking failed. For other robots, we could have more primitives return a boolean success value, depending on the robustness of the implementation.

D. Graphical interface

To facilitate programming for users without formal programming experience, we used a graphical interface called Blockly [15]. Blockly is not a programming language itself, but a framework for building visual programming languages. In Blockly, program elements such as constant values, binary operators, while loops, or function calls are represented as blocks shaped like puzzle pieces. These blocks can be connected by stacking them, attaching values to inputs, or nesting blocks inside of other blocks. Blockly allows custom blocks to be made by defining a block's appearance, inputs, and outputs.

We designed custom blocks for each primitive, with inputs and outputs as shown in Section III-B. We also provided

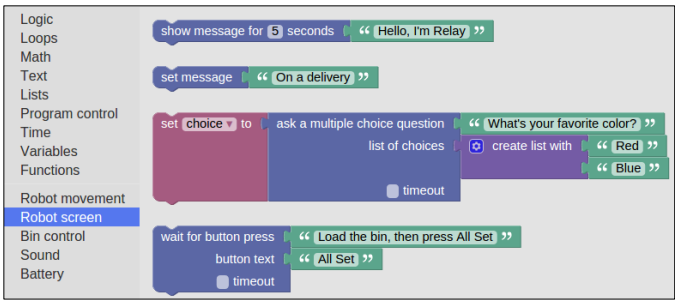

Fig. 2: Screenshot of the interface, showing how blocks can be pre-assembled for convenience.

standard programming language constructs such as loops, conditionals, variables, functions, strings, lists, math utilities, and logical operators. A full list of all the standard blocks can be seen on Blockly's website[3].

Blockly does a shallow form of type checking, which can eliminate some errors. For example, a block representing a string value can't be attached to a block expecting a number. However, it does not guarantee the correctness of the types used, and the type constraints can be circumvented.

A useful feature of Blockly is the ability to organize blocks in an arbitrary hierarchy of menus. We organized the primitives into categories as shown in Tables I, II, and III. The blocks were color-coded so that blocks in the same category shared the same color. Additionally, snippets of code, represented as pre-assembled collections of blocks, can be added to the menus. Figure 2 shows how we used this. The *ask a multiple choice question* involves 4 different kinds of blocks, which we pre-assemble in a commonly used configuration.

E. Compilation and runtime

Each block generates JavaScript code, by first recursively generating code for its input blocks. The code for the block itself is then assembled along with the inputs appropriately. The code for the program as a whole is done by generating code for the blocks at the top level of the program.

When the user starts the program, the generated JavaScript code is sent to the robot. The robot runs a sandboxed JavaScript interpreter[4] on a Node.js server. The interpreter parses and executes the code, including primitives. In our implementation, the interpreter uses roslibjs [24] to call ROS services and actions that execute the primitives.

IV. Use cases

Before developing *CustomPrograms*, we interviewed 2 employees at Savioke about the possible uses for the Relay robot, outside of deliveries in hotels. Additionally, we received informal feedback from employees at Savioke acting as points of contact for hotel customers. The use cases that turned up included both internal uses and customer-facing programs.

[3]https://blockly-demo.appspot.com/static/demos/code/index.html
[4]https://github.com/NeilFraser/JS-Interpreter

A. Employee interviews

We conducted separate semi-structured interviews with 1 designer and 1 business person at Savioke. For question 3 below, we showed the employees the default version of Blockly, from the Blockly website. The probe questions were:

1) When or where might you want to create a custom program for the robot?
2) Are there examples of programs you've wanted to make?
3) What do you think of Blockly?
4) How much time do you want to spend learning to program?
5) How much time do you want to spend making a demo program for a conference?

The business person said they would want to spend around 20 minutes making a program if they had to make it on the spot (e.g., at an off-site demo). The designer said that it would be fine to spend more time creating programs, as long as they were reusable. Both said an hour would be a reasonable time to learn to use the system.

B. Use cases

1) Customizing deliveries: One use case, mentioned by the designer, would be for hotel staff to customize the robot's interaction with the guest. For example, the robot could show "Happy birthday!" on screen, if applicable. Having this capability would also make it easy to pick up items like laundry from a room, instead of doing a delivery. The robot would just have to visit the room with an empty bin, and say something different at the door. The designer also mentioned that hotels have asked for multi-destination deliveries to staff members in the hotel. Instead of returning to the front desk after a single delivery, it could visit multiple floors in one trip.

2) Mingling with guests: Both employees mentioned mingling with hotel guests as a use case. The business person described a kiosk mode, where the robot displays information about the hotel while not in use, and has lightweight interactions with guests. One hotel had expressed interest in having the robot do something similar in the breakfast area in the morning. Another hotel asked if the robot could go to different locations in the lobby, displaying information at each location. In these examples, the robot is also advertising itself, potentially leading customers to ask for a delivery later.

3) Information gathering: A third type of capability that was asked for revolved around information gathering. For example, some Savioke employees discussed the idea of detecting food trays in the hallways, and reporting the locations to the hotel staff. Another example was to ask people in the breakfast area to rate their stays. Reporting bad stays would be useful for hotels, to avoid getting a bad review online.

4) Other uses: Savioke employees informally discussed internal programs they wanted the robot to run. These included running programs to test new software, and creating custom behaviors for sales demos. In Section VI-D, we give two case studies of our system being used for these purposes.

Additionally, researchers studying the use of mobile robots in other settings such as retirement communities could use the system. Such research requires a rapid prototyping system to create a large space of candidate designs [25], before the designs are narrowed down and detailed.

V. User study

We evaluated our system by conducting an observational study with non-software engineering employees at Savioke, and experienced programmers outside of Savioke. The goal of the study was to understand how well each group could use our system to make real-world programs. We also wanted to see whether users thought the system was expressive enough for the uses they had in mind.

A. Procedure

Participants were seated at a computer in view of the robot they would program. The participants were first introduced to the robot with a paragraph-length description and short video. Then, they completed a 45-minute tutorial on how to use *CustomPrograms*. In the tutorial, the users made an application that sent the robot to a room, and a program that asked whether to go to the "Front Desk" or to the dock (Figure 1).

The participants then created three programs with *CustomPrograms*, one program at a time. These programs, shown in Table IV, were based on the real-world use cases discussed in Section IV. The robot was configured with locations including room numbers, a "Front Desk", and some poses in the nearby area representing the lobby. Some participants (those in Group 2, see Section V-C) were allowed to optionally create a fourth program, of their own choosing, if time allowed. All participants were asked additional questions at the end of the study. Finally, we also collected demographics such as gender, age, and prior programming experience.

B. Measures

1) Ease of use: An objective measure of the system's ease of use was whether or not users were able to create the three programs correctly. For each program, we determined if the program correctly matched the program description we gave. Because some errors are more serious than others, we labeled the errors participants made, and classified them as either major or minor. Although the labels could be subjective, in practice, most of the errors made were easy to spot and distinguishable from one another. We considered minor errors to be easy to fix errors such as a missing statement or errors that could be fixed by changing a constant value. An example of a minor error might be displaying the wrong text on screen. An example of a major error might be using an if statement to repeatedly check a condition, instead of a *while* loop.

We also gathered subjective measures for ease of use. After completing each program, users were asked to rate the easiness of making the program on a 7 point Likert scale, and to describe what the most challenging part of making it was.

2) Expressiveness: We considered two measures of expressiveness for this study. After being introduced to the robot, but before being exposed to the programming interface, users were asked to think of a specific task the robot could do other than

#	Prompt
1	**Goal**: The robot drives around the lobby, stopping at certain places to explain to guests what it does. **Program behavior**: The robot should go to "Lobby 1" and "Lobby 2" over and over again. At each place it visits, it should stop and show these three messages for 10 seconds each: "I'm Relay, a delivery robot", "Need something? Call the front desk and I'll bring it up!", "Have a great day!" While the robot is travelling, it should say, "Excuse me, on my way".
2	**Goal**: Pick up items from a guest room. **Program behavior**: When the program starts, the robot should go to the front desk, and ask which room number to go to. The robot should be able to go to any room number it knows about. The robot should go to the room and open its bin. It should say "Please load your items" and present a "Done" button. Once the guest touches the "Done" button, the robot should close its bin and go back to the front desk. After that, the program should end. When the robot is out on a pickup it should say "On a pickup" on its screen. When it's returning to the front desk, it should say "Returning home" on its screen.
3	**Goal**: Stand in the lobby, going back to dock if it needs to charge. **Program behavior**: The robot should go to "Lobby 1" and say "Welcome to the hotel!" for 60 seconds on its screen. After showing the message for 60 seconds, it should check if its battery level is at least 50%. If so, it should show the message for 60 seconds again. If the battery is below 50%, the robot should go to the dock. While in the dock, it should say "Welcome to the hotel!" for 30 seconds, and "I'm charging up" for 30 seconds, until its battery is full. Once the battery is full, it should go back to the "Lobby 1" and repeat.

TABLE IV: The prompts describing each of the 3 programs to make in the user study.

hotel delivery. They were asked to describe these programs in a step-by-step fashion. We analyzed these programs to determine whether *CustomPrograms* could be used to make them. Additionally, after making all three programs with the interface, we asked users to rate their agreement with the statement, "Any task the robot could possibly do, without any hardware modifications, can be programmed using this interface" on a 5 point Likert scale.

C. Participants

We conducted the study with two groups of participants. The first group of participants, referred to as Group 1, were employees at Savioke who were not in software engineering roles. This group represented an important group of potential users–designers, sales people, and engineers who would want to prototype programs for hotels or for internal uses. To study the system with programmers outside of Savioke, we recruited an additional group, Group 2, from amateur robotics email lists targeted at the San Francisco Bay Area. This group represents another group of early users–researchers or other programmers who might use our system outside of hotels. Participants in the second group were required to have at least 2 years of programming experience, and were offered a $50 Amazon.com gift card for their participation.

There were 18 participants, 9 in each group. We asked all participants to rate their prior programming experience on a scale of 1 to 5, with 1 being no prior experience and 5 being professional level of experience. All of the programmers in Group 2 rated their prior experience a 5, while Group 1 had less experience (M = 2.67, SD = 1), ranging from 1 to 4. We consider *experienced* programmers to be any participant who rated their prior experience a 4 or above. Based on this definition, 2 of the users in Group 1 were experienced, and 7 were not. The Group 2 programmers were asked to approximate their years of programming experience, which ranged from 6 - 50 years (M = 20.3, SD = 13.6). Group 1 included 7 males and 2 females with ages ranging from 23 to 64 (M = 36.1, SD = 13.5); the Group 2 programmers were all male with ages ranging from 24 to 73 (M = 48.7, SD = 16.4).

VI. Evaluation

There are three parts to the evaluation of *CustomPrograms*. First, we examine its ease of use, using the data from our user study. Second, we analyze its expressivity, based on use cases from Section IV and from the user study. Finally, we present two case studies showing how the system was used in practice.

A. Ease of use

In the user study, Group 1 made 46% of the programs with only minor errors, compared to 77% by Group 2, shown in Table V. The subjective ease of making each program is shown in Table VI.

Program	All users	Group 1	Group 2
All programs	0.62	0.46	0.77
Program 1	0.71	0.56	0.88
Program 2	0.5	0.33	0.67
Program 3	0.65	0.5	0.78

TABLE V: The rate of programs made with only minor errors.

Program	All users	Group 1	Group 2
All programs	5.14	4.73	5.58
Program 1	5.71	5.44	6.00
Program 2	4.76	4.22	5.38
Program 3	4.94	4.50	5.38

TABLE VI: The perceived ease of making each program, on a 7 point Likert scale.

When asked, two Group 2 programmers said they would definitely prefer a graphical programming interface, and one said they would definitely prefer a text programming interface. The remaining gave a mixed response, saying that the graphical interface would be better for people with less experience, or for one-off tasks, while a text interface would be better for complex programs or day-to-day use.

B. Error analysis

Across all three programs, we found that Group 1, which had less experienced programmers, had a harder time making programs than Group 2. The errors they made were mostly related to programming concepts like infinite loops, string concatenation, and programming logic.

Program 1 required an infinite loop to repeat the program forever. However, 3 programmers, all from Group 1, used a loop that repeated a finite number of times. One way to alleviate this issue could be to have a *repeat forever* block. In our system, programmers had to make infinite loops with *while-true*, which is not obvious for new programmers.

Program 1 errors	**Type**	**G1**	**G2**
Did not loop forever	Major	3	0
Did not show message when travelling	Major	1	1
Looped 5000 times, but not forever	Major	1	0
Did not show required message the first time the robot travels	Minor	0	3
Did not show correct message when travelling	Minor	1	1

TABLE VII: # of people making each error in Program 1.

In Program 2, the robot was supposed to ask for a room number, and go there using string concatenation, as in, **robot.goTo**("Room " + number). Although making the robot go to a room number was part of the tutorial, users may not have fully absorbed how to do it. Instead, users avoided that part by presenting a limited subset of rooms as a multiple choice question, or by going to lobby locations rather than rooms. Two Group 1 users who used string concatenation did so incorrectly, e.g., by omitting a space after the word "Room".

Program 2 errors	**Type**	**G1**	**G2**
Could only go to a subset of possible rooms	Major	2	1
Could only go to locations other than rooms	Major	2	1
Did not concatenate `Room` with room number correctly	Major	2	0
Did not go to front desk after pickup	Major	0	1
Did not show correct message when travelling	Minor	1	2
Did not show any message when travelling	Minor	2	0

TABLE VIII: # of people making each error in Program 2.

Program 3 required programmers to continuously monitor the robot's battery state using *while* loops. These loops needed to be nested inside another *while* loop, to repeat the program forever. 4 of the Group 1 users used an *if* statement at the top level to check the battery level, rather than a *while* loop (Figure 3). 4 users used an incorrect primitive to make the robot dock. We had a **goToDock** primitive specifically for this purpose; however, these users used the **goTo** primitive, and made the robot go to a location named "PreDockingPose," which was a location in front of the dock.

Program 3 errors	**Type**	**G1**	**G2**
Did not use correct block to go to dock	Major	2	2
Did not continuously check battery before docking	Major	3	0
Did not continuously check battery before undocking	Major	1	0
Did correct checks, but did not loop program forever	Major	1	0
Continuously checked for wrong battery level before undocking	Minor	1	3
Displayed message for wrong amount of time	Minor	1	2
Did not display correct message(s) on screen	Minor	1	1
Continuously checked for wrong battery level before docking	Minor	0	2

TABLE IX: # of people making each error in Program 3.

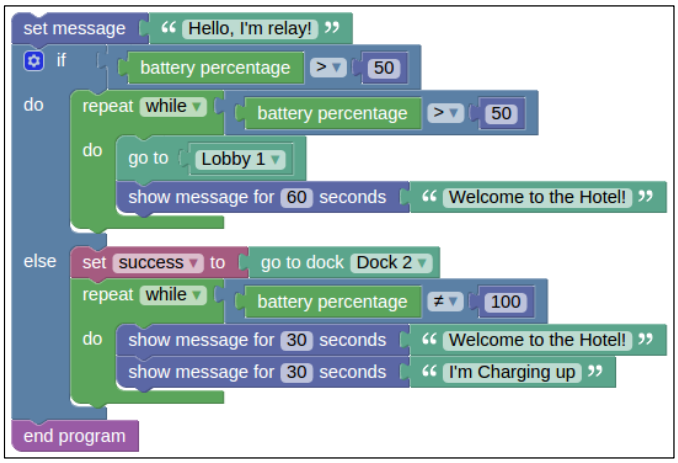

Fig. 3: A Group 1 participant's version of Program 3. The participant used *while* loops to wait for the battery to charge and discharge, but used an *if* statement at the top level instead of an infinite loop.

Overall, even the major errors made were not too serious, and could be avoided with just a small additional investment of development time or training.

C. Expressiveness

In this section, we characterize our system's ability to create applications envisioned by Savioke employees (Section IV) and by user study participants. We consider 4 categories of applications, based on how much effort it would take to implement: those that can be done, those that need minor software changes, those that need major new software capabilities, and those that require hardware modifications.

1) Savioke use cases: Our system implements all the primitives necessary to do a normal hotel delivery, so small wording changes can be made easily. Visiting multiple locations on one trip is also just a small modification of normal delivery.

Our system can display messages to guests while mingling. It can also interact with guests and gather information by asking for star ratings, ratings on a numeric scale, or multiple choice questions. The responses can be stored in variables or lists. However, the robot can only communicate the answers by displaying the information on screen. Other ways, such as email, would need minor software development.

One behavior our system does not support is scheduling the robot to run parts of the programs at certain times of day. For example, the hotel would not be able to have the robot automatically go to the breakfast area at 6 AM every day. We chose not to implement this because we wanted to keep hotel staff in control of the robot's availability. Making this feature available would require minor software development.

Our system does not include any perception-related primitives, such as a hotel tray detector. We consider this to be major development work, as many of these perception tasks are still active areas of research. However, as perception primitives are created, they can be added to the interface as blocks.

2) User study programs: Participants in the user study came up with new use cases for the robot in 11 unique settings, before they had seen the interface. The most common setting, envisioned by 7 participants, was in an office building. Other settings included homes, retirement communities, hospitals, warehouses, and restaurants. Most programs were variations on delivery, such as delivering medication in a retirement community on a schedule, or delivering packages in an office. A few applications were more novel, such as checking if the front door was locked, or using the robot as a shopping cart.

8 of the 18 applications were doable immediately with our system. 5 applications could be done with minor software development. 2 applications required major new software capabilities the robot did not have, such as using computer vision to check if the front door of a house was locked. 4 of the applications required hardware modifications to the robot, such as readers for office badges, RFID, and credit cards.

Of the 5 applications that needed short-term development, 2 of them needed the ability to input text through a software keyboard, 2 of them required the ability to schedule tasks for the future, and 1 required the ability to call phones.

For the optional 4th program, one experienced programmer tried to make the robot drive in a Fibonacci spiral, and said that it would be interesting to use the robot to teach math.

After trying the interface, users were asked to rate, on a scale of 1 to 5, whether *CustomPrograms* could program any task the robot could do (barring hardware modifications). Users gave a slightly positive response (M = 3.28, SD = 1.27) to this question.

D. Case studies

1) Intel Developer Forum 2015: One of the motivating use cases for *CustomPrograms* was to customize the robot's behaviors for trade shows and sales demos. In the summer of 2015, *CustomPrograms* was deployed to 5 Relay robots at the Intel Developer Forum, a large technology conference. The day before the conference, we developed and tested a demo application in about an hour.

In our initial designs, all user input primitives waited indefinitely for input. However, we realized that even at a crowded conference, there might not always be someone around to interact with the robot. As a result, we added an optional timeout to all user input primitives, and updated our program. If a timeout was set and no user input was provided, then the primitives returned `null`, which the program handled.

At the conference, the robot's bin was loaded with snacks and water bottles. The robots then drove around to several locations around its area, and offered a snack to anyone nearby. If no one asked for a snack, the robot timed out and went to a different location. The robot also told people to visit a booth at the conference. While the program performed well, one limitation was the inability to customize the navigation behavior. Groups of people often crowded around the robot, which made the robot stop moving. The robot could have made more progress if it slowly pushed its way through crowds.

2) Internal testing: When *CustomPrograms* was deployed to robots at Savioke, one technical business person wrote a program for stress testing the robot. The program drove the robot around the office continuously, stopping only to charge its battery. During the first few attempts, employees noticed that the robot behaved oddly after driving for several hours. After investigating the issue, they discovered that the robot was having difficulty storing log data after driving for a long period of time. This issue had not been encountered before in the field because the robot did not drive as frequently and as long in normal use. This experience helped guide the development of changes to fix this issue. Eventually, *CustomPrograms* was able to run on a robot for over 24 hours continuously.

VII. Discussion

We believe *CustomPrograms*, along with the design of our primitives, is a useful framework for rapid programming on the Relay robot, with some limitations.

Our user study results showed that users could make real-world programs for the Relay robot, with programming errors that are easy to correct. Most of the differences between the two groups can be explained by prior programming experience. The first group had less programming experience, and made basic programming errors like not properly concatenating a number to a string. Group 2 programmers had some issues with the idiosyncrasies of our interface. For example, in Program 3, just as many Group 2 users did not use the **goToDock** primitive to go to the dock as Group 1 users.

During the study, participants had limited time (approximately 90 minutes) to learn the system and create three programs. We envision that users interested in using the interface would spend more time learning and using the interface. As shown in Section VI-D, our system enables new public-facing applications to be developed and tested in a matter of hours.

We believe that the interface will scale to meet increasing needs over time. Because we use a general-purpose programming language, users will have expressive ways to manage complexity. For example, some of the experienced programmers in our study organized their code with functions, although we never explained how. Additionally, we can add more primitives to our system at any time, and organize them in an ever-growing standard library.

A. Revision implications

Additional development of the system would make it more usable. In Section IV, we discussed missing capabilities that could be useful, like scheduling a program to run in the future, perception primitives like a tray detector, and the ability to send emails. And in Section VI-C, we mentioned text input and the ability to call phones as missing primitives. We also would want to refine the blocks and user interface to address the issues we encountered in the user study.

B. Design implications

Blockly offers the full expressivity of a general-purpose programming language, so it is not surprising that experienced

programmers had an easier time with the interface than less experienced programmers. While it eliminates the need to remember syntax, some researchers have long argued that software engineering is inherently difficult [26]. As a result, our system may never be fully intuitive for non-technical end-users. However, an area for future research is to understand whether we could adopt a "copy and paste literacy" model of end-user programming [27]. In this model, technically skilled users would create and distribute programs, while less experienced users could modify and customize them [28].

It would not be difficult to implement our system on a different robot. Each primitive would be implemented through some commonly used middleware like ROS. For example, **robot.goTo**(...) could simply send a goal to a ROS action that navigates the robot, the exact implementation of which could be different for each robot.

C. Limitations

CustomPrograms is built on top of Blockly, which was designed with educational purposes in mind. In its current form, Blockly generates global variables for all variables. This naturally limits the maximum complexity of the program. The scale of a program is also limited by the fact that code cannot be shared between two programs.

In our study, we did not incentivize users to make correct programs. This may have led to programmers not testing their code or not carefully reading the program specifications. For example, in Program 1, one of the experienced programmers in Group 1 repeated the program 5000 times instead of making an infinite loop. Most likely, the programmer was familiar with infinite loops, but they either did not read the program specification carefully enough, or they did not want to spend the time to make the loop correctly.

VIII. Conclusion

We presented the design and evaluation of *CustomPrograms*, which allows users to rapidly experiment with new applications for the Relay robot. *CustomPrograms* supports many use cases such as deliveries, information gathering, and more. Savioke employees successfully used the system internally and at a large trade show. Our user study showed that programmers could use the system to make real-world programs. Our study also revealed common mistakes users made and helped us design future iterations of the system.

References

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

[2] B. Myers, "Visual programming, programming by example, and program visualization: a taxonomy," in *ACM SIGCHI Bulletin*, vol. 17, no. 4. ACM, 1986, pp. 59–66.

[3] G. Biggs and B. MacDonald, "A survey of robot programming systems," in *Proceedings of the Australasian conference on robotics and automation*, 2003, pp. 1–3.

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

[5] A. Repenning and A. Ioannidou, "What makes end-user development tick? 13 design guidelines," in *End User Development*. Springer, 2006, pp. 51–85.

[6] D. Hils, "Visual languages and computing survey: Data flow visual programming languages," *Journal of Visual Languages & Computing*, vol. 3, no. 1, pp. 69–101, 1992.

[7] S. Alexandrova, Z. Tatlock, and M. Cakmak, "RoboFlow: A flow-based visual programming language for mobile manipulation tasks," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on*. IEEE, 2015, pp. 5537–5544.

[8] 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.

[9] B. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[10] C. Atkeson and S. Schaal, "Robot learning from demonstration," in *International Conference on Machine Learning (ICML)*. Morgan Kaufmann, 1997, pp. 12–20.

[11] J. Huang and M. Cakmak, "Supporting mental model accuracy in trigger-action programming," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 215–225.

[12] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, "Practical trigger-action programming in the smart home," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2014, pp. 803–812.

[13] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[14] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages," *ACM computing surveys (CSUR)*, vol. 37, no. 4, pp. 316–344, 2005.

[15] N. Fraser *et al.*, "Blockly: A visual programming editor," 2013.

[16] D. C. Cliburn, "Experiences with the LEGO Mindstorms throughout the undergraduate computer science curriculum," in *Frontiers in Education Conference, 36th Annual*. IEEE, 2006, pp. 1–6.

[17] P. B. Lawhead, M. E. Duncan, C. G. Bland, M. Goldweber, M. Schep, D. J. Barnes, and R. G. Hollingsworth, "A road map for teaching introductory programming using LEGO Mindstorms robots," *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 191–201, 2003.

[18] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The Scratch programming language and environment," *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16, 2010.

[19] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling Alice motivates middle school girls to learn computer programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2007, pp. 1455–1464.

[20] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The Carnegie Mellon navigation (CARMEN) toolkit," in *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2003, pp. 2436–2441.

[21] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*, 2005, p. 145.

[22] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[23] S. Yamada and T. Komatsu, "Designing simple and effective expression of robot's primitive minds to a human," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE, 2006, pp. 2614–2619.

[24] R. Toris, J. Kammerl, D. V. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills, and S. Chernova, "Robot Web Tools: Efficient messaging for cloud robotics."

[25] B. Buxton, *Sketching user experiences: getting the design right and the right design*. Morgan Kaufmann, 2010.

[26] F. Brooks, *No silver bullet*. April, 1987.

[27] D. Perkel, "Copy and paste literacy? Literacy practices in the production of a MySpace profile," *xxxix. Informal Learning and Digital Media: Constructions, Contexts, Consequences, eds. Kirsten Drotner, Hans Siggard Jensen, and Kim Schroeder (Newcastle, UK: Cambridge Scholars Press, 2008)*, 2006.

[28] W. E. Mackay, "Users and customizable software: A co-adaptive phenomenon," Ph.D. dissertation, Citeseer, 1990.