ConCodeIt! A Comparison of Concurrency Interfaces in Block-Based Visual Robot Programming

Michael Jae-Yoon Chung, Mino Nakura, Sai Harshita Neti, Anthony Lu, Elana Hummel, Maya Cakmak

***Abstract*— Concurrency makes robot programming challenging even for professional programmers, yet it is essential for rich, interactive social robot behaviors. Visual programming aims to lower the barrier for robot programming but does not support rich concurrent behavior for meaningful robotics applications. In this paper, we explore extensions to block-based visual languages to enable programming of concurrent behavior with (1) asynchronous procedure calls, which encourage imperative programming, (2) callbacks, which encourage event-driven programming, and (3) promise, which also encourages imperative programming by providing event synchronization utilities. We compare these approaches through a systematic analysis of social robot programs with representative concurrency patterns, as well as a user study (N=23) in which participants authored such programs. Our work identifies characteristic differences between these approaches and demonstrates that the promise-based concurrency interface enables more concise programs with fewer errors.**

I. Introduction

Social robots are becoming increasingly ubiquitous across domains including entertainment, education, social-emotional learning, and mental health support, among others. Programming social robots to be robust, effective, and engaging for every unique use case and environment remains a bottleneck given the complex multi-modal, interactive nature of desired robot behaviors.

Research in end-user robot programming [1], [2], [3], [4] aims to create tools that let people with little or no experience with software development write robot programs on their own through intuitive interfaces. End-user programming often involves *abstractions* of programming concepts to simplify the programming task as well as *constraints* to avoid programmer errors. These simplifications often come at a cost of expressivity, i.e., robot behaviors obtainable using simplified languages are not as rich as ones created using general-purpose languages by expert programmers.

One concept that is particularly difficult to simplify, yet essential for robot behaviors, is *concurrency*–the execution of multiple threads in parallel. Interactive social robot behaviors require concurrency in many ways. For example, coordinating multiple expressive actions, like gesturing and making sounds, or handling human interruptions, such as poking or leaving during a robot action, require concurrency handling [1], [5]. However, creating programs with concurrent behavior is notoriously difficult even for professional programmers [6] because it involves complex concepts, such as callback registration and handling, in widely used robot programming frameworks like ROS[1].

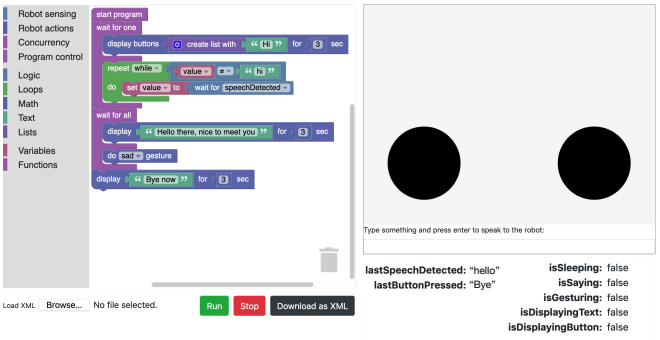


Fig. 1. ConCodeIt! web interface used for the study. (Left) Block-based visual program editor. (Right) The robot's face and texts showing its status.

Prior work on end-user robot programming provides at least one means of expressing concurrency [7], [8], [9], [5], [10]. However, to our knowledge, no prior work has investigated and compared alternative approaches for supporting concurrency in robot programming through simplified interfaces. Our paper aims to close this gap. To that end, we present Concurrent CodeIt! (ConCodeIt!), a block-based visual programming system that lets novice programmers create different concurrent behaviors on a robot (extending our previous work on CodeIt! [2], [11], [5]). We designed three alternative concurrency interfaces for ConCodeIt! with (1) **asynchronous procedure calls**, which encourage imperative programming, (2) **callbacks**, which encourage event-driven programming, and (3) **promises**, which also encourage imperative programming by providing event synchronization utilities. We first identified representative social robot programs that have common concurrency patterns, and we systematically analyzed programs written with each concurrency interface to characterize its expressivity. We then comparatively evaluated the three interfaces through a user study in which participants authored concurrent robot behaviors. Our results indicate that callback-based concurrency is most complex and challenging for lay programmers, while promise-based concurrency allows more concise and intuitive programs.

II. Related Work

Research in end-user robot programming aims to produce easily understood abstractions of robot programming languages and develop intuitive user interfaces for

Paul G. Allen School for Computer Science and Engineering, University of Washington. Seattle, WA 98195, USA. mjyc,nakuram,saihn,strato95,elana99,mcakmak@cs.washington.edu

[1] https://www.ros.org/

programming. Researchers have developed several visual programming languages based on abstractions including flowcharts [8], [9], [12], state machines [13], behavior trees [14], block-based imperative languages [2], [5], [11], and trigger-action rules [15]. Systems have been developed for programming mobile or tabletop manipulators [12], [2], [11], [16], [13], [17], humanoid robots [9], [1], [15], and social robots [18]. Others have investigated alternative modes of interaction, such as natural language [18], body-storming [4]. and tangible interfaces [19].

A subset of prior work presents ways to express concurrency within a robot programming system. Lourens and Barakova's textual language [7] and Chorepographe, a default programming system for the Nao humanoid robot [8], provide generic operators such as wait, parallel, and sequence for coordinating actions and sensor inputs. Interaction composer enables the handling of asynchronous events using the interrupt module, which executes the robot action defined within the module when a monitoring event is triggered [9], [1]. Robot programming systems have been built by taking a human-centered approach that provides specialized ways for expressing frequently desired concurrent actions [5], [10]. Some robot control systems that aim to make authoring socially intelligent and reactive and to simplify complex robot behaviors are based on a mathematical model or programming paradigm that emphasizes concurrency support, such as timed Petri net [20], behavior tree [21], and reactive programming paradigms [22]. Although the preceding list provides options for expressing concurrency, the work gives priority to developing a robot programming system. Our work further investigates and evaluates different ways of supporting programming interactive robots with concurrency.

III. System Overview

ConCodeIt! has three components: a general-purpose programming language with (1) custom robot action and sensing primitives and (2) custom concurrency constructs. It also includes (3) a graphical user interface.

A. Programming Language

ConCodeIt! exposes a small subset of JavaScript to users via Blockly, a block-based visual programming interface [23]. This subset includes JavaScript language constructs for expressing variables, loops, and conditionals but does not include concurrency functions such as `setTimeout`. ConCodeIt! supports concurrency with its custom functions for robot interfacing (Sec. III-B) and custom concurrent constructs (Sec. III-C).

B. Robot Primitives

1) Robot Platform: ConCodeIt! is not tied to a particular robot; however, we focus in this paper on using ConCodeIt! to program an idealized social robot, Meebo. Meebo consists of a "face;" a GeeekPi 7inch touchscreen for displaying messages and buttons; and a "neck," a 5-DoF Open MANIPULATOR-X robot arm for making head gestures such as nod and shake. It can speak and recognize speech via a speaker microphone attached to the connected computer. The control system for Meebo is a JavasScript library that exposes function calls for triggering robot actions, like displaying messages, and for returning (processed) sensor outputs, like the recognized speech texts of a user. We used Cycle.js to build a web application for displaying messages and buttons and the Web Speech API implementation for the Chrome browser for speech synthesis and recognition.[2] The web application was loaded on the touchscreen when the robot was turned on. For gesturing, we used the open_manipulator_controller ROS package and the roslib npm package for controlling the robot arm.[3] For the user study, we used a simulated version of Meebo, as shown in Fig. 1right, to conduct the study online (Sec. VI).

TABLE I
Robot Actions

Name and arguments	Asynchrony
say(string *text*)	blocking
gesture(string *expression*)	blocking
displayText(string *text*, number *duration*)	blocking
displayButton(string[] *choices*, number *duration*)	blocking
sleep(number *duration*)	blocking
startSaying(string *text*)	non-blocking
startGesturing(string *expression*)	non-blocking
startDisplayText(string *text*, number *duration*)	non-blocking
startDisplayButton(string[] *choices*, number *duration*)	non-blocking
startSleeping(number *duration*)	non-blocking

2) Robot Actions: Actions are robot control processes that can be started from a program and run until finished or preempted. Actions can be synchronous, i.e., block the process until the action is done, or asynchronous, i.e., start an action and move on to executing the next statement. For each action, we assume only one instance can run at a time regardless of an action's asynchrony. Available actions are shown as function definitions in Table I; action names are shown in bold, arguments are in italic, and variable types are in normal font. The asynchrony of available robot actions is shown in the right column of Table I.

We define the following robot actions. **say** causes the robot to say a phrase, and **gesture** makes the robot do one of a few pre-specified gestures, like a "happy" or "sad" gesture, by moving its neck and making a facial expression. Only one message can be said at a time, and only one gesture can be played at a time. **displayText** displays a text message for the user, and **displayButtons** displays a list of buttons as choices on the face screen. Only one message and only one set of buttons can be displayed at a time. **sleep** pauses program execution for the specified duration. Only one sleep can be used at a time.

3) Robot Events and States: Events indicate the occurrence of some change at a specific point in time. For ConCodeIt!, we defined two event categories: externally triggered

[2]https://cycle.js.org, https://wicg.github.io/speech-api

[3]http://wiki.ros.org/open_manipulator, https://npmjs.com/package/roslib

TABLE II

ROBOT EVENTS AND STATES

Event name	Last event state name	Value type
speechDetected	**lastDetectedSpeech**	string
buttonPressed	**lastPressedButton**	string

(a) Externally triggered events

Event name	Action status state name	Value type
sayDone	**isSaying**	boolean
gestureDone	**isGesturing**	boolean
displayTextDone	**isDisplayingText**	boolean
displayButtonDone	**isDisplayingButton**	boolean
sleepDone	**isSleeping**	boolean

(b) Action-triggered events

TABLE III

THREE CONCODEIT! INTERFACE SETUPS

Name	Blocking actions	Non-blocking actions	Action events	Action states	External events	External states
async	no	yes	no	yes	no	yes
callback	no	yes	yes	yes	yes	yes
waitfor	yes	no	no	yes	yes	yes

and action-triggered events. Externally triggered events are initiated by some entity and sensed and processed by the robot. For example, a user finishing speaking to the robot triggers a **speechDetected** event. Action-triggered events indicate the completion of running an action, e.g., a **sayDone** event occurs when the robot finishes saying a phrase. *States* are conditions whose value can be evaluated and accessed at any given time. Events can be converted to states by storing the most recently emitted value, for example, the **lastDetectedSpeech** state stores the most recently detected speech text as a string value.

Table II shows the complete list of the robot events and states. A **buttonPressed** event is triggered when a robot detects a human pressing one of the displayed buttons shown using **displayButtons**. The **lastPressedButton** state stores the value of the most recent **buttonPressed** event as a string. An ***actionName*Done** event occurs when the robot completes an action. Finally, the **is*ActionName*ing** states indicate the running status of the robot actions.

C. Concurrent Programming in ConCodeIt!

ConCodeIt! provides the following functions to help users work with concurrent robot events and states. The `when` function takes the `eventName` string and `callback` functions as input arguments and invokes the `callback` function when an `eventName` event occurs. The accessor functions, such as `getLast`*`EventName`* and `getIs`*`ActionName`*`ing`, return stored last event or latest state values. `wait` function takes `eventName` which blocks the process until one `eventName` event occurs, and returns the value of the occurred event. The `waitForAll` and `waitForOne` take a list of programs as input arguments; `waitForAll` blocks the process until all input programs are finished, and `waitForOne` blocks the process until one of the input programs is finished. The behavior of `waitForAll` and `waitForOne` is modeled after common promise-based synchronization utilities.[4] In fact, since ConCodeIt! is based on JavaScript, we leveraged JavaScript's features, such as event handlers and async/await promise, to implement the concurrency functions.

We present three ConCodeIt! interfaces–**async**, **callback**, and **waitfor**–that employ different combinations of the concurrency functions noted above (see Table III). Each implementation supports a different concurrent programming approach. **async** is designed to provide the traditional concurrent programming experience, e.g., as in socket programming, to users. Users have access to the asynchronous function calls for starting actions and the action state accessor functions. **callback** is inspired by JavaScript's event-driven programming and *Star Wars: Building a Galaxy With Code* exercises at Code.org[5]. Similar to **async**, users can launch the robot's actions using asynchronous function calls and check the action status and latest values of externally triggered events using the state accessor functions. Users also have access to the `when` function to register a callback function; the callback function is called for occurrences of the specified robot action or external event specified by the user. **waitfor** helps to easily express concurrency behaviors while staying in the imperative programming paradigm using promise,[6] more specifically the `wait`, `waitForOne`, `waitForAll` functions that make use of promise. Like the other two implementations, users have access to all state accessor functions; however, they can use blocking functions to run actions instead of using non-blocking trigger functions.

D. Graphical User Interface

The graphical user interface for ConCodeIt! uses Blockly, a library for creating block-based visual programming editors (Fig. 1left). Users drag and drop blocks from a toolbox of predetermined blocks onto a workspace to construct programs located on the left side of the editor. The blocks are organized into different categories – such as loops, logic, math, and ConCodeIt!-specific blocks – which can be linked through stacking or nesting or attached to variables that will be returned. Users can run or stop the program using the buttons located below the editor. Once running, the program controls the robot face displayed on the right of the interface and updates the robot status displayed below the face. Blockly enables code generation, which we use to generate the code for the ConCodeIt!-specific functions, as discussed in Sec. III-B and Sec. III-C. Blockly also supports exporting a visual block program as an XML file, which we use in the "Download as XML" button displayed below the editor.

[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race

[5]https://code.org/starwars

[6]https://en.wikipedia.org/wiki/Futures_and_promises

IV. Concurrency Patterns

We are interested in the common concurrency patterns involved in robot programming. To this end, we define the concurrency patterns with (1) the source of the robot events, and (2) how users want to handle multiple events. There are two sources of the robot events, robot action and external (Sec. III-B.3) and two ways to handle events, waiting for all events or waiting for a certain event to occur. Since there are three ways to combine the sources of events (e.g., Action-Action, Action-Event, Event-Event) and two waiting approaches, there are six concurrency patterns of interest.

Table IV shows example code for three selected concurrency patterns expressed using the three ConCodeIt! implementations. The three patterns are (1) running two actions in parallel and waiting for all of them to finish, (2) waiting for one of two alternate inputs, and (3) waiting for an input event or the end of a running action. At a high-level, **async** requires using a loop to check robot states; **callback** requires using the **when** functions for reacting to events; and **waitfor** requires using **waitForAll** or **waitForOne**.

V. Systematic Evaluation

To objectively investigate the similarities and differences among three concurrent programming approaches, we systematically evaluated the three ConCodeIt! interfaces: (1) imperative programming with asynchronous function calls, (2) event-driven programming with callbacks, and (3) imperative programming with event synchronization helper functions (Sec. III-C), as well as the common concurrency pattern in robot programming (Sec. IV).

A. Procedure

We first designed a set of unit concurrent robot behaviors, one for each concurrency pattern described in Sec. IV. We describe unit behaviors in Table V. We designed each behavior as a minimal robot behavior that expresses the associated concurrency pattern, and we implemented each behavior using each of the three ConCodeIt! interfaces. Where there were multiple ways to implement the desired behavior, we chose the approach requiring the least number of blocks. The resulting visual programs were stored as XML files to run our evaluation metrics.

B. Measures

We approximated the complexity of the unit behavior programs using the following measures.

1) Number of blocks: The total number of individual blocks used to implement a program. The three ConCodeit! interfaces have different total numbers of available block types because of the interface-specific blocks (Sec. III-C). For example, the `when` block is available only in **callback**, and `waitForAll` and `waitForOne` are available only in **waitfor**.

2) Number of functions: The total number of function definition blocks. This measure includes the `when` block counts because it is used to define a callback function.

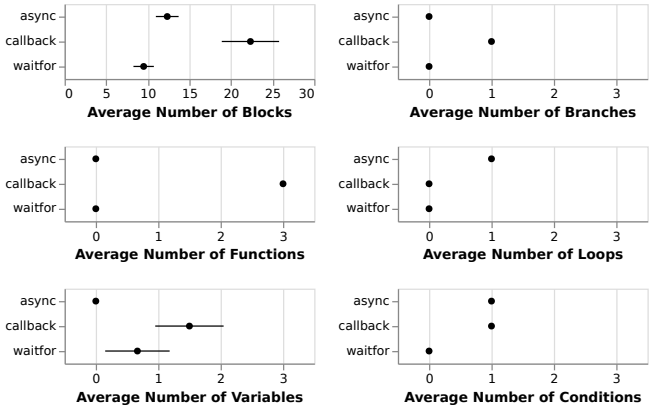


Fig. 2. Systematic evaluation results: interfaces vs. measures.

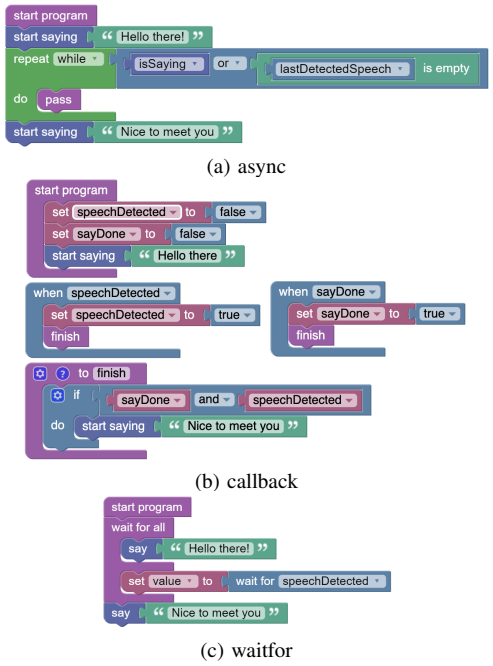


Fig. 3. Wait-for-All with Action-Event (WA-AE) programs implemented using three ConCodeIt! interfaces.

3) Number of variables: The total number of variable definition blocks.

4) Number of loops: The total number of loop statement blocks, such as `for` and `while` blocks.

5) Number of branches: The total number of if statement blocks, such as `if`, `else`, `else if`, and logical ternary blocks. Note that if an `if` statement contains multiple `else if`s, it is counted as one plus the number of `else if`s.

6) Number of conditions: The total number of logical operator blocks, such as negation, conjunction, and disjunction blocks, and arithmetic comparison operator blocks, such as greater than and equals.

C. Results

Fig. 2 shows the results of the systematic analysis. Overall, **callback** resulted in the most complex programs since it had the highest averages in most measures (5/6, i.e., all

TABLE IV

SELECTED CONCURRENCY PATTERNS IN THE THREE CONCODEIT! IMPLEMENTATIONS

async

Action-Action & Wait-for-All
run two actions in parallel

```
startSaying("hi");
startGesturing("happy");
while (isSaying() ||
  ↪ isGesturing()) {
  sleep(1);
}
// do the next thing
```

Event-Event & Wait-for-One
wait for one of two alternate inputs

```
startDisplayingButton("Start",
  ↪ 1000)
var lastSpeechVal =
  ↪ getLastDetectedSpeech();
var lastButtonVal =
  ↪ getLastPressedButton();
while(lastSpeechVal == null &&
  ↪ lastButtonVal == null) {
  sleep(1);
  lastSpeechVal =
    ↪ getLastDetectedSpeech();
  lastButtonVal =
    ↪ getLastPressedButton();
}
// do the next thing
```

Action-Event & WaitForOne
wait for input while running an action

```
startDisplayingButton("Continue",
  ↪ 1000);
var lastButtonVal =
  ↪ getLastPressedButton();
startSaying("very long
  ↪ sentence")
while(isSaying() &&
  ↪ lastButtonVal == null) {
  sleep(1);
  lastButtonVal =
    ↪ getLastPressedButton();
}
// do the next thing
```

callback

```
startSaying("hi")
startGesturing("happy")

when("sayDone", function() {
  if (isGesturing()) return;
  // do the next thing
});

when("gestureDone", function() {
  if (isSaying()) return;
  // do the next thing
});
```

```
startDisplayingButton("Start",
  ↪ 1000)

when("speechDetected",
  ↪ function() {
  // do the next thing
})

when("buttonPressed", function()
  ↪ {
  // do the next thing
})
```

```
displayButton("Continue", 1000);
startSaying("a very very long
  ↪ sentence");

when("displayingButtonDone",
  ↪ function() {
  // do the next thing
});

when("buttonPressed", function()
  ↪ {
  // do the next thing
});
```

waitfor

```
waitForAll(
  'say("Hi")',
  'gesture(happy)'
);
// do the next thing
```

```
waitForOne(
  'displayButton("Start",
    ↪ 1000)',
  'wait("speechDetected")',
  'wait("buttonPressed")'
);
// do the next thing
```

```
waitForOne(
  'displayButton("Continue",
    ↪ 1000)',
  'say("a very very long
    ↪ sentence")',
  'wait("buttonPressed")'
)
// do the next thing
```

TABLE V

UNIT CONCURRENT ROBOT BEHAVIOR DESCRIPTIONS

	Wait-for-All (WA)	**Wait-for-One (WO)**
Action-Action (AA)	*Step 1:* The robot should say "Hello there!" and do the "greet" gesture. *Step 2:* The robot should say "My name is Meebo" and do the "happy" gesture.	*Step 1:* The robot should say "Hello" and sleep for 3 seconds. *Step 2:* The robot should say "timed out."
Action-Event (AE)	*Step 1:* The robot should say "Hello there!" and wait for a face to appear. *Step 2:* The robot should say "Nice to meet you!"	*Step 1:* The robot should say "Hello there, my name is Meebo. Goodbye now!" and wait for the human's face to disappear. *Step 2:* The robot should display "On standby."
Event-Event (EE)	*Step 1:* The robot should wait for the human to look to the center and stop speaking. *Step 2:* The robot should say "Hello."	*Step 1:* The robot should wait for the human to look to the left and wait for the human to look right. *Step 2:* The robot should display "Bye now!"

excepts the number of branches). Next was **async** since it had the highest averages in two measures (i.e., the number of branches and the number of conditions). Finally, **waitfor** resulted in the simplest programs since it had the lowest averages in all measures (Fig. 2).

Only **callback** programs required the use of functions and if statement blocks. These programs typically needed separate `when` blocks for each action/event to be detected, variables to monitor whether an action/event had been completed, then a separate function that was triggered in each when block to perform the final action (e.g., see Fig. 3b). Another reason for **callback** programs' use of the highest number of variables on average was individual variables needed to monitor the state of the robot's interactions.

Only **async** programs required the use of a loop statement block. These programs typically needed to continuously block the rest of the program from executing until either one or all of the events/actions were completed, which was done using a while loop with a `pass` block inside (see Fig. 3a for an example).

Finally, only **waitfor** programs did not require the use of any logical connective or arithmetic comparison operator blocks. These programs typically used one of the concurrent blocks that waited for one or all of the actions/events within it, and no other blocks were needed (see Fig. 3c for an example).

TABLE VI
USER STUDY TASK DESCRIPTIONS AND RUBRIC

Practice task
Step 1: On start, the robot should say "rain rain go away" and do the "sad" gesture. (WA-AA)
Step 2: Once the robot finishes saying "rain rain go away" and doing a "sad" gesture, it should say "little johnny wants to play" and do the "happy" gesture. (WA-AA)
Main task
Step 1: On start, the robot should display "Press or say 'start' to begin interaction" as well as a button with "start" text. (WO-EE)
Step 2: If the user presses the button or says "start," then the robot should introduce itself by saying "Hello, my name is Meebo" and make a happy gesture. (WA-AA)
Step 3: If the robot finishes both actions (i.e., saying and gesturing), the robot should display a question "What is 2^{12}? You have 10 seconds to answer" as well as a button with "I'm ready to answer" text. The question and button should be displayed for 10 seconds. (WO-AE)
Step 4: If 10 seconds pass without the user pressing the button, the robot should display "You are out of time." If the button is pressed in time, the robot should display "Please say your answer" and wait for the user to say the answer. If the user says an answer, the robot should check the answer and accordingly display "correct" or "wrong" for 5 seconds.
Rubric for the main task
Step 1.1: displays both button and text until next step
Step 1.2: moves on when "start" button is pressed
Step 1.3: moves on when "start" human speech is detected
Step 1.4: correctly implements Wait-for-One behavior
Step 2.1: introduces itself and makes a gesture
Step 2.2: waits for "happy" gesture to finish
Step 2.3: waits for the robot to finish speaking
Step 2.4: only moves on when both robot actions are over
Step 3.1: asks a question and displays a button
Step 3.2: waits for 10 seconds
Step 3.3: moves on when the button is pressed
Step 3.4: displays "You are out of time" when button is not pressed after 10 second
Step 4.1: displays text to prompt answer
Step 4.2: waits for user to input speech
Step 4.3: moves on when speech is detected
Step 4.4: displays whether input is correct or not

VI. USER STUDY

A. Study Design

Like the systematic evaluation (Sec. V), the user study had three conditions that represented the three concurrent robot programming approaches provided by the three ConCodeIt! interfaces (Sec. III-C): **async**, **callback**, **waitfor**. The goal of the study was to compare the three concurrent programming approaches. We used a between-groups design, i.e., each participant was assigned to one of the three conditions. The study was conducted online using a Google form that contained instructions and a ConCodeIt! web page (Fig. 1). Participants were asked to spend around 60 to 90 minutes to complete the study and were offered a $20 Amazon.com gift card for their participation.

Participants in all conditions were shown a video tutorial explaining how to use the ConCodeIt! interface and interact with the simulated robot. In addition, all participants were given an interface-specific video tutorial to explain the assigned ConCodeIt! interface-specific functions (Sec. III-C). They were then asked to implement two tasks (Table VItop,middle). The first was a practice task, and the solution was provided. Next, participants were asked to implement the main task and submit their program once finished. After completing the task, they answered a post-study questionnaire (Sec. VI-B.3).

B. Measures

The ease of use of each condition was estimated using the following measures.

1) Systematic evaluation measures: These included the six block count measures used in Sec. V-B.

2) Correctness of programs: We developed a rubric to evaluate the correctness of participant-submitted programs (see Table VIbottom).

3) Post-study questionnaire: We asked our participants to complete the System Usability Scale (SUS) questionnaire[7]. We also asked all participants to rank their programming experience on a scale of 1 to 5, with 1 being no prior experience and 5 being a professional level of experience, and to list any programming courses they completed. The final open-ended question aimed to elicit general comments, i.e., "Any other comments about the study?"

C. Participants

The target users of ConCodeIt! are people with basic programming knowledge and no robot programming experience. We recruited participants using introductory class and general mailing lists from the University of Washington Computer Science and Engineering (CSE) Department. The participants were dividend three separate groups, one for each ConCodeIt! interface. Group 1 was asked to use **async**, Group 2 to use **callback**, and Group 3 to use **waitfor**.

There was a total of 21 participants, six for Group 1 (four female) and Group 2 (three female) conditions and nine for Group 3 (seven female). The means and standard deviations of age were: M = 20.67 & SD = 2.80 (Group 1); M = 21.71 & SD = 2.40 (Group 2); M = 19.67 & SD = 0.87 (Group 3). The means and standard deviations of self-reported programming experience were: M = 3.00 & SD = 0.63 (Group 1); M = 1.03 & SD = 1.90 (Group 2); M = 3.22 & SD = 0.67 (Group 3). Of the 23 participants, 1 of 23 had taken no programming classes, 17 of 23 completed introductory courses, 9 of 23 had experience taking mid-level programming courses, and 1 of 23 took advanced programming courses offered by the University of Washington.

D. Results

Fig. 4 shows the six measures applied to the participant submitted programs. Overall, the programs in **waitfor** required the lowest average number of blocks (in 3/6 measures, i.e., number of blocks, variables, and loops), and the programs in **async** required the highest average number of blocks (in 4/6 measures, i.e., number of blocks, variables, and loops, and conditions). **callback** required the highest average number of functions, and the average number of branches and **async** required the highest average number of loops.

[7]https://www.usability.gov

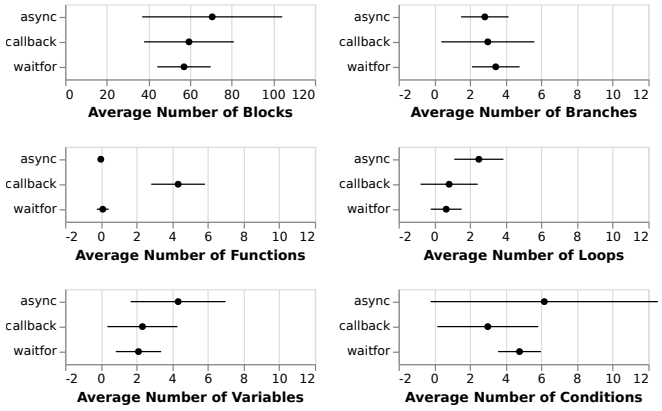


Fig. 4. User study results: interfaces vs. measures.

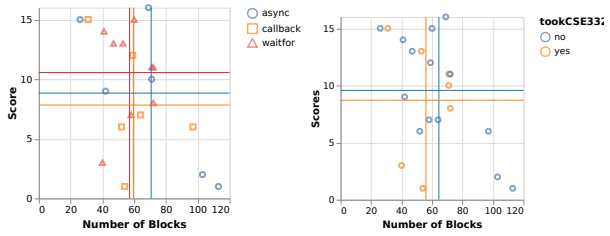


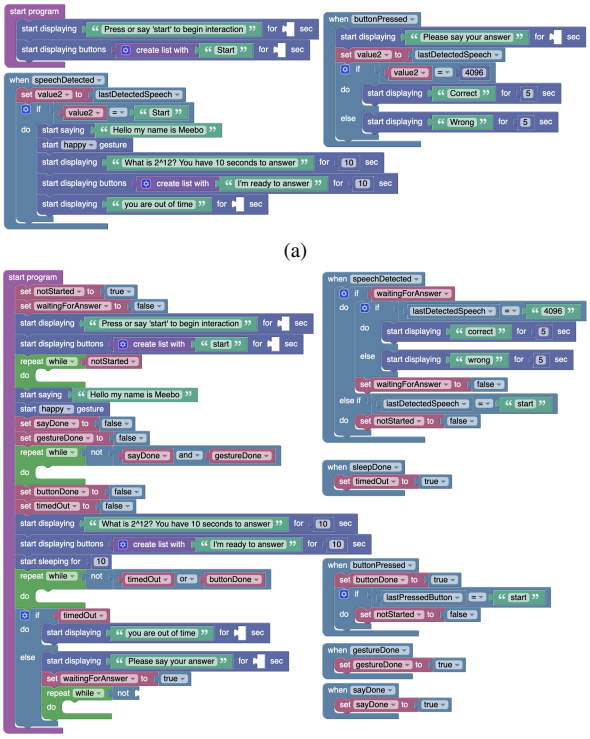
Fig. 5. Number of blocks vs. user scores vs. interfaces (left) or vs. CSE 332 background (right). Averages are displayed as lines.

Regarding the correctness of programs, the mean and standard deviation of score were: M = 8.83 & SD = 6.31 (Group 1); M = 7.83 & SD = 4.96 (Group 2); M = 10.56 & SD = 3.88 (Group 3). The highest possible score was 16. The relationship between the number of blocks and scores across the three interfaces is shown in Fig. 5left. We visualized the number of blocks and scores across participants who took a programming course that covers concurrency (CSE 332) and those who did not (Fig. 5right). The average for participants who took CSE 332 was not higher than for those who did not.

The means and standard deviations of SUS score were: M = 40.00 & SD = 9.22 (Group 1); M = 42.08 & SD = 22.27 (Group 2); M = 46.39 & SD = 16.64 (Group 3). The scores are below the SUS average score (68). Investigating the open-ended comments from 19 participants, seven mentioned issues related to the Blockly editor (*"... annoying to figure out making lists one element"*, *"Would be nice to have a zoom in/out feature on the interface."*); nine people mentioned that they did not understand how to display buttons and detect a pressed button (*"... I spent 10 minutes figuring out how to display a button"*); and three people mentioned the potential benefit of having a reference sheet (*"It would be helpful to have a glossary/element lookup."*). Only one person explicitly mentioned the challenge with concurrent programming (*"The parallel programming part definitely needs to be explained very explicitly ..."*).

E. User-Created Program Examples

Participants in the study thought of ways to create programs with ConCodeIt! interfaces that we did not envision



(a)

(b)

Fig. 6. User-created programs using "callback" ConCodeIt!

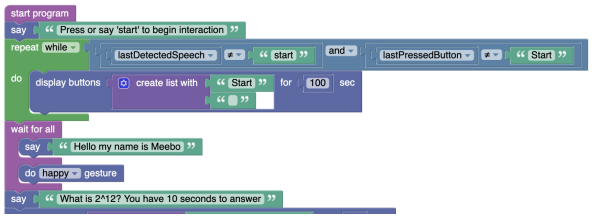


Fig. 7. A user-created program using "waitfor" ConCodeIt!

when initially designing them. For example, we expected participants to use **callback** to implement the most logic in each callback function, as shown in Fig. 6a. However, some participants created the main loop with sub-loops for checking the global variables representing the state of the program. In this case, the callback functions (i.e., `when` blocks) were used to update the global variables to indicate the change in the state (see Fig. 6b). Another unexpected pattern we observed was in the programs created with **waitfor**. While **waitfor** lets programmers avoid using loops to wait for an event (see examples in Fig. 3a,c), some participants used loops, like the participants who used **async** did (Fig. 7).

VII. Discussion

Based on our study, the **waitfor** interface, which is based on imperative programming with promise-like event synchronization utilities, was the easiest to use in the context of programming interactive robots. Participants who used this interface created the most concise programs with the highest scores on average (Sec. VI-D). These participants had the youngest average age and had taken the least number of

programming courses compared to participants who used the other ConCodeIt! interfaces (Sec. VI-C), viz., **callback** based on event-driven programming and **asynch** based on imperative programming with asynchronous procedure call utilities. However, we acknowledge that the number of participants involved in our study was small.

One question we considered throughout the study was why **callback** was challenging. Investigating the user-created programs from the user study (Sec. VI-E), we noticed that explicit state management (e.g., using variables to indicate which "state" the robot is in) and the multiple patterns a programmer could employ (e.g., using the main loop and global variables that are modified in callbacks vs. chaining callback functions) were the key areas of difficulty. Participants may have also had problems due to their unfamiliarity with event-driven programming. Most participants had taken the introductory and mid-level programming courses at CSE that use imperative, not event-driven, programming.

Compared to **waitfor**, the **asynch** interface, which also uses imperative programming but with asynchronous procedure call utilities, made participants perform redundant work to implement common concurrency patterns (Fig. 3a). Overall, we believe that adding minimal features to support concurrency while allowing programmers to keep a consistent mental model is key to achieving ease of use while supporting desired concurrency behaviors.

VIII. Conclusion

This paper presented ConCodeIt!, a block-based visual programming system for programming interactive robots. We first defined a framework for (1) identifying programming constructs required for expressing concurrency, and (2) categorizing common concurrency patterns in the context of programming robots. We then proposed three programming systems that represent common approaches for expressing concurrency and compared them via a systematic evaluation and an online user study. Our results show that the imperative programming paradigm with synchronization support produced more concise and predictable programs, while the event-driven one was more challenging for programmers without robotics knowledge.

ACKNOWLEDGMENT

This work was supported by the National Science Foundation, Awards IIS-1552427 "CAREER: End-User Programming of General-Purpose Robots " and IIS-1925043 "NRI: INT: COLLAB: Program Verification and Synthesis for Collaborative Robots." We thank Heather Harvey, Justin Huang, Gitanjali Nandi, Lily Orth-Smith, and Wendy M. Xu for their contributions in the earlier phases of this project.

References

[1] D. F. Glas, T. Kanda, and H. Ishiguro, "Human-robot interaction design using interaction composer eight years of lessons learned," in *International Conference on Human-Robot Interaction*. ACM/IEEE, 2016, pp. 303–310.

[2] J. Huang, T. Lau, and M. Cakmak, "Design and evaluation of a rapid programming system for service robots," in *ACM/IEEE International Conference on Human Robot Interaction (HRI)*, 2016, pp. 295–302.

[3] A. Kubota, E. I. Peterson, V. Rajendren, H. Kress-Gazit, and L. D. Riek, "Jessie: Synthesizing social robot behaviors for personalized neurorehabilitation and beyond," in *ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2020, pp. 121–130.

[4] D. Porfirio, E. Fisher, A. Sauppé, A. Albarghouthi, and B. Mutlu, "Bodystorming human-robot interactions," in *Symposium on User Interface Software and Technology*, 2019.

[5] M. J.-Y. Chung, J. Huang, L. Takayama, T. Lau, and M. Cakmak, "Iterative design of a system for programming socially interactive service robots," in *International Conference on Social Robotics*, 2016, pp. 919–929.

[6] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.

[7] T. Lourens and E. Barakova, "User-friendly robot environment for creation of social scenarios," in *International Work-Conference on the Interplay between Natural and Artificial Computation*, 2011, pp. 212–221.

[8] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier, "Choregraphe: a graphical tool for humanoid robot programming," in *The International Symposium on Robot and Human Interactive Communication*. IEEE, 2009, pp. 46–51.

[9] D. Glas, S. Satake, T. Kanda, and N. Hagita, "An interaction design framework for social robots," in *Robotics: Science and Systems*, vol. 7, 2012, p. 89.

[10] J. Diprose, B. MacDonald, J. Hosking, and B. Plimmer, "Designing an api at an appropriate abstraction level for programming social robot applications," *Journal of Visual Languages & Computing*, vol. 39, pp. 22–40, 2017.

[11] J. Huang and M. Cakmak, "Code3: A system for end-to-end programming of mobile manipulator robots for novices and experts," in *International Conference on Human-Robot Interaction*. ACM/IEEE, 2017, pp. 453–462.

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

[13] F. Steinmetz, A. Wollschläger, and R. Weitschat, "Razer-a human-robot interface for visual task-level programming and intuitive skill parameterization," *Robotics and Automation Letters*, vol. 3, no. 3, pp. 1362–1369, 2018.

[14] C. Paxton, F. Jonathan, A. Hundt, B. Mutlu, and G. D. Hager, "Evaluating methods for end-user creation of robot task plans," in *International Conference on Intelligent Robots and Systems*. IEEE, 2018, pp. 6086–6092.

[15] N. Leonardi, M. Manca, F. Paternò, and C. Santoro, "Trigger-action programming for personalising humanoid robot behaviour," in *Conference on Human Factors in Computing Systems*, 2019, pp. 1–13.

[16] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "Costar: Instructing collaborative robots with behavior trees and vision," in *Onternational Conference on Robotics and Automation*. IEEE, 2017, pp. 564–571.

[17] Y. Gao and C.-M. Huang, "Pati: a projection-based augmented table-top interface for robot programming," in *International Conference on Intelligent User Interfaces*, 2019, pp. 345–355.

[18] N. Buchina, S. Kamel, and E. Barakova, "Design and evaluation of an end-user friendly tool for robot programming," in *International Symposium on Robot and Human Interactive Communication*. IEEE, 2016, pp. 185–191.

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

[20] C. Chao and A. L. Thomaz, "Timing in multimodal turn-taking interactions: Control and analysis using timed petri nets," *Journal of Human-Robot Interaction*, vol. 1, no. 1, pp. 4–25, 2012.

[21] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *International Conference on Robotics and Automation*. IEEE, 2014, pp. 5420–5427.

[22] V. Berenz and S. Schaal, "The playful software platform: Reactive programming for orchestrating robotic behavior," *Robotics & Automation Magazine*, vol. 25, no. 3, pp. 49–60, 2018.

[23] N. Fraser, "Blockly: A visual programming editor," *URL: https://code. google. com/p/blockly*, 2013.