Iterative Repair of Social Robot Programs from Implicit User Feedback via Bayesian Inference

Michael Jae-Yoon Chung and Maya Cakmak
Computer Science & Engineering
University of Washington
Seattle, Washington 98195
Email: {mjyc, mcakmak}@cs.washington.edu

***Abstract*—Creating natural and autonomous interactions with social robots requires rich, multi-modal sensory input from the user. Writing interactive robot programs that make use of this input can demand tedious and error-prone tuning of program parameters, such as tuning thresholds on noisy sensory streams for detecting whether the robot's user is engaged or not. This tuning process dealing with low-level streams and parameters makes programming of social robots time-consuming and inaccessible for people who could benefit the most from unique use cases of social robots. To address this challenge, we propose the use of iterative program repair, where programmers create an initial program sketch in our new Social Robot Program Transition Sketch Language (SoRTSketch), a domain-specific language that supports expressing uncertainties related to thresholds in transition functions. The program is then iteratively repaired using Bayesian inference based on corrections of interaction traces that are either provided by the programmer or derived from implicit feedback given by the user during the interaction. Based on experiments with a human simulator and with 10 human users, we demonstrate the ease and effectiveness of this approach in improving social robot programming and program outputs that represent three common human-robot interaction patterns. We also show how our approach helps programs adapt to environment changes over time.**

I. Introduction

The past decade has been marked by the rise of social robots, with soaring interest from start-ups, large corporations, and researchers alike. Purchasing and building social robot hardware have been democratized through declining price points and the availability of fabrication and physical computing tools. However, programming a social robot for a particular application still requires specialized software development skills. As a result, the ability to explore new applications is limited to a small population. For example, a children's hospital that wants to incorporate social robots into autism-related therapy can easily purchase a robot platform, but it will struggle to find qualified programmers to customize the robot for their specific needs and incur additional costs every time the robots need re-programming.

Research on end-user programming of social robots aims to address this problem by simplifying the programming process to let end-users to program robots on their own. Most prior work in this area provides a high-level application programming interface (API) [33, 3, 14, 11, 24]), which comes at the cost of expressivity. For example, due to the difficulty of creating robust, application-independent detectors for high-level state variables, such APIs are often limited to basic functionality. Exposing low-level parameters to end-users to increase flexibility of programs is also not a good option given the difficulty of finding a set of program parameters that yields fluent and robust human-robot interactions, even for expert programmers [34, 14, 18]. Tuning such parameters often involves tedious, time-consuming rounds of trial-and-error to understand the impact of each parameter and the interaction between combinations of parameters, and then find the right set of parameters.

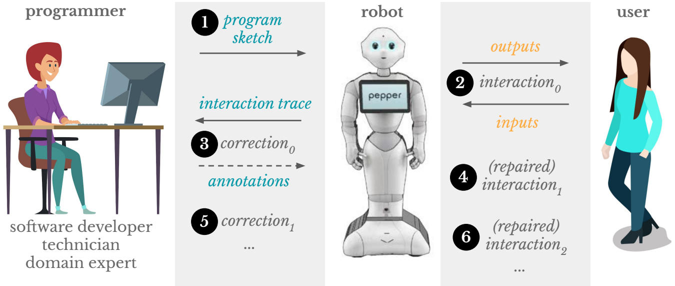


Fig. 1: Overview of the iterative program repair process. (1) The programmer creates a *program sketch* representing a finite state machine, or FSM, (2) the program is executed on the robot, and a user interacts with the robot, (3) the program is automatically repaired based on annotations over the interaction trace that are either provided by the programmer or derived from implicit feedback in user inputs, (4) steps 2-3 are repeated until a satisfactory program is obtained.

In this work, we aim to offer a middle ground, whereby more expressive programs for social robots can be created without exposing low-level parameters. We propose an iterative program repair approach that involves creating a preliminary *program sketch* and improving it over time through feedback on executions of the program (Fig. 1). For example, we envision our approach will let non-roboticist programmers to create a set of social robot behavior templates which later can be customized and tuned by users of the robot. We propose a sketch language called SoRTSketch (Sec. III-A) for representing interactive social robot behaviors as Finite State Machines (FSMs). This language represents parameters in an FSM transition function with *hole* variables and specifies their distributions. Our proposed repair algorithm iteratively estimates the probability distributions over these variables based on *corrections* of erroneous state occurrences during previous executions. Corrected state traces are either obtained from the programmer, e.g., by annotating the interaction trace, or derived from *implicit feedback* from the user, such as tapping the "Go back" button after the program automatically transitioned to the next state. We demonstrate, quantitatively and qualitatively, how programs can be rapidly repaired using this approach, first by using a simulated stochastic human model and then with real human data from 10 users across three and two social robot applications, respectively.

II. Related Work

The robotics community has long been interested in applying formal methods, such as verification and synthesis, to robotics problems. Techniques in formal verification have been applied to assess the correctness of concurrent and time-critical programs [12] that check both general and application-specific properties [26]. Researchers also used program synthesis to find a plan or a controller for navigation [13], mobile manipulation [27], or multi-robot planning problems [40] that satisfy specifications often expressed in a temporal logic language. Other work explored alternative specification languages, such as structured natural languages [20] or adaptations of existing robot programs in new environments interactively with a programmer, e.g., for robot soccer [18] or tabletop manipulation [7]. Most recently, Hammond et al. introduced a system that can automatically recover from errors incurred while running end-user created mobile service robot programs [15]. Our work most closely relates to that of Holtz et al. on automatically repairing robot-soccer playing programs, which are represented as FSMs, using sparse state corrections from a programmer with a MaxSMT solver [18, 19]. However, we focus on repairing social robot programs using Bayesian inference and feedback provided by the human who is interacting with the robot.

Applications of formal methods in robotics are not being extended to include scenarios relevant to human-robot interaction. Formal verification is being used to ensure effective and reliable human-robot teamwork for astronaut-robot collaboration in space missions [6] and validate the safety requirements of robotic assistants [41]. Program synthesis is being used to create safe, human-in-the-loop controllers from high-level temporal specifications [23] and programs for neuro-rehabilitation using social robots [22]. Porfirio et al. [31] contributed a system for authoring human-robot interaction programs for social robots and verifying that the program meets social norms or best practices discovered from the human-robot interaction research community. The same researchers also developed a system to synthesize interactive programs from traces of two people acting out the interaction scenario [32]. While most work in this category focuses on reasoning about high-level human-robot interaction task structure, recent work of Kshirsagar et al. [21] involves synthesizing low-level controllers for human-robot handover tasks. Similarly, our work involves program repair at a lower level while maintaining a high-level program structure.

```
⟨uniary-operator⟩ ::= - | ! | abs | ...
⟨binary-operator⟩ ::= + | * | > | < | && | || | == | ...
⟨temporal-operator⟩ ::= delay | debounce | average | ...
⟨expression⟩ ::= k                                                        constant
    | s                                                                   state
    | v                                                                   variable
    | x                                                                   input
    | θ                                                                   hole
    | ⟨uniary-operator⟩ ⟨expression⟩
    | ⟨expression⟩ ⟨binary-operator⟩ ⟨expression⟩
    | ⟨expression⟩ ? ⟨expression⟩ : ⟨expression⟩                          ternary operator
    | ⟨temporal-operator⟩ ( ⟨expression⟩ , ⟨expression⟩ )
⟨statement⟩ ::= return s;
    | if ( ⟨expression⟩ ) { ⟨statement⟩ } else { ⟨statement⟩ }
    | ⟨statement⟩
⟨transition⟩ ::= ⟨statement⟩
```

Fig. 2: Formal syntax of SoRTSketch for representing FSM transition functions as program sketches.

III. Approach

Our approach, summarized in Fig. 1, starts with a programmer creating a program sketch that is iteratively refined it through interactions with a user. We now describe this process in details.

A. Program Sketches

A *program sketch* is a partial program that encodes the high-level structure of a solution while leaving low-level details unspecified [38]. Program details of a program can be left unspecified using hole variables that are later derived by the repair algorithm. In this paper, we present **SoRTSketch** (Social Robot Program Transition Sketch Language), a domain-specific language for sketching social robot behaviors based on finite state machines (FSMs) with transitions that are not fully specified.

1) FSM description: SoRTSketch is based on FSMs, widely used to represent robot behaviors [8, 33, 4, 28, 29, 25, 37, 9, 1]. Specifically, we represent social robot behaviors as discrete-time *Mealy machines* with continuous inputs and outputs and program variables. Formally, our FSM is a tuple $(S, S_0, V, V_0, \Sigma, \Lambda, T)$, where S is a finite set of states, S_0 is the start state, V is a finite set of program variable values, V_0 is the start program variable values, $\Sigma \in \mathbb{R}^n$ are continuous inputs (obtained from robot sensors), $\Lambda \in \mathbb{R}^m$ are continuous outputs (executed as robot actions), and $T : S \times V \times \Sigma \to S \times V \times \Lambda$ is a transition function. At each time step t, the transition function is executed to update the state $s_{t+1} \in S$ and program variables $v_{t+1} \in V$ (an update could be $s_{t+1} = s_t$ or $v_{t+1} = v_t$) and to output a robot action to be triggered in next time step $\lambda_{t+1} \in \Lambda$ (could be $noaction$).

2) Language description: A transition function in SoRTSketch consists of an `if-else` statement that checks the values of variables derived from the current FSM state $s \in S$, variables $v_t \in V$, and sensor inputs $(x_t^1, ..x_t^n) \in \Sigma$. It assigns values of the new FSM state $s_{t+1} \in S$, variables

$v_{t+1} \in V$, and output action $a_{t+1} \in \Lambda$, accordingly. Sketching capabilities leverage the fact that the condition expressions in `if` statements can include *hole* variables instead of constants. Fig. 2 shows part of the formal syntax for representing transition functions in SoRTSketch.

As an example, consider an interactive storytelling social robot. To ensure the user pays attention while the story unfolds, the robot can move into a paused `"wait"` state whenever it detects that the user has disengaged. This requires the transition function to encode *disengagement* in the `if` expression, by comparing an input variable (such as the person's gaze direction `faceYawAngle`) with a threshold `maxEngagedAngle`. Where conventional programs would require assigning `maxEngagedAngle` to a constant, SoRTSketch lets the variable be added to a list of hole variables for later repair.

Programmers can create a hole variable whenever they are uncertain about the exact value of a constant needed for the transition function. Further, they must also specify the distribution of the variable as a probability density function (e.g., Bernoulli). As described in Sec. III-C, this lets the repair algorithm treat hole variables as random variables in Bayesian inference.

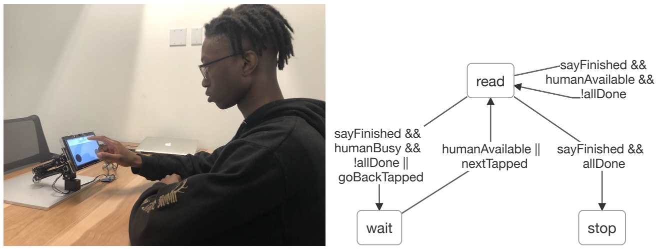
3) Temporal operators: Due to noise in a robot's sensory inputs, knowing only the latest value of sensory inputs can be insufficient for creating fluent human-robot interactions. Therefore, SoRTSketch includes three temporal operators that have access to the history of a sensory stream:

1) `average`(x, τ) returns the average value of an input variable x over the specified duration τ into the past; i.e., between $t - \tau$ and t.
2) `delay`(x, τ) returns the value of an input variable x from duration τ into the past; i.e., it creates an input stream delayed by τ.
3) `debounce`(*`expression`*, τ) returns the conjunction of the boolean *expression* involving an input variable, over the specified duration τ into the past.

As an example, if the `faceYawAngle` input variable is known to be noisy, an `if` expression could compare `average(faceYawAngle, historyDuration)` to a threshold `maxAverageEngagedAngle`, where both duration and threshold variables could be holes.

4) Example program: Fig. 3 shows instantiations of different categories of expressions with holes in a transition function implemented with SoRTSketch for the storytelling robot behavior. For example, it shows: how holes can represent uncertainty in deciding which operator to use (`useAvgOp`); the use of multiple sensory streams (`faceYawAngle` and `voiceLevel`) over time; within a complex expression with multiple parameters, to represent high-level concepts like engagement; and how to make decisions about state transitions.

Category	Variable name	Example value
constant	`nSentences`	10
state	`curState`	`"read"`
variable	`curSentenceIndex`, `hold`	1, `false`
input	`faceYawAngle`, `voiceLevel`, `sayFinished`, `goBackTapped`, `nextTapped`	7, 0.1, `0`, `0`, `0`
hole	`minDisengagedAngle`, `disengagedTimeout`, `maxEngagedAngle`, `engagedTimeout`, `speakingWindow`, `minSpeakingLevel`, `useAvgOp`	15, 1500, 30, 500, 1000, 0.4, `true`


read
sayFinished && humanAvailable && !allDone
sayFinished && humanBusy && !allDone || goBackTapped
humanAvailable || nextTapped
sayFinished && allDone
wait
stop

```
lookingAtRobot = debounce(abs(faceYawAngle <
↪  minDisengagedAngle), disengagedTimeout);
lookingAway = debounce(abs(faceYawAngle >
↪  maxEngagedAngle), engagedTimeout);
speaking = useAvgOp ? average(voiceLevel,
↪  speakingWindow) > minSpeakingLevel :
↪  debounce(voiceLevel, speakingWindow) >
↪  minSpeakingLevel;
allDone = sayFinished && (curSentenceIndex ==
↪  nSentences)
humanAvailable = !lookingAtRobot && !speaking;
humanBusy = lookingAway || speaking;

if (curState == "read" && sayFinished &&
↪  humanAvailable && !allDone) {
  return "read";
} else if (curState == "read" && (sayFinished &&
↪  humanBusy && !allDone || goBackTapped)) {
  return "wait";
} else if (curState == "read" && sayFinished &&
↪  allDone) {
  return "stop";
} else if (curState == "wait" && (humanAvailable ||
↪  nextTapped && !hold))) {
  return "read";
}
```

Fig. 3: An example program in SoRTSketch. (top) Instantiation of SoRTSketch in a storytelling social robot domain with defined constants, inputs, variables, and holes; (middle) the social robot used this paper and an FSM visualization for the storytelling program; (bottom) transition function program with variable definitions and `if-else` statements.

B. State Traces and Corrections

1) Program execution: SoRTSketch program sketches are executable even before they are repaired. For example, the holes can be set to the sampled values from the given distributions. When SoRTSketch programs are executed on a robot, users can interact with the robot in different ways. We refer to a sequence of inputs $I = [(x_1^1, ...x_1^n), ..., (x_\mathcal{T}^1, ...x_\mathcal{T}^n)]$ received over $\mathcal{T}$ steps as an *input trace* and the state values resulting from running the FSM, $O = [s_1, ..., s_\mathcal{T}]$ as a *state trace*. If the program's hole variables are assigned to a set of "good" values, i.e., ones that make the robot behave exactly as the programmer intended during the interaction, the state trace for that interaction can be considered correct.

However, interactions with a program before hole variables are adjusted through program repair often involve errors. We can measure the correctness of program execution based on the

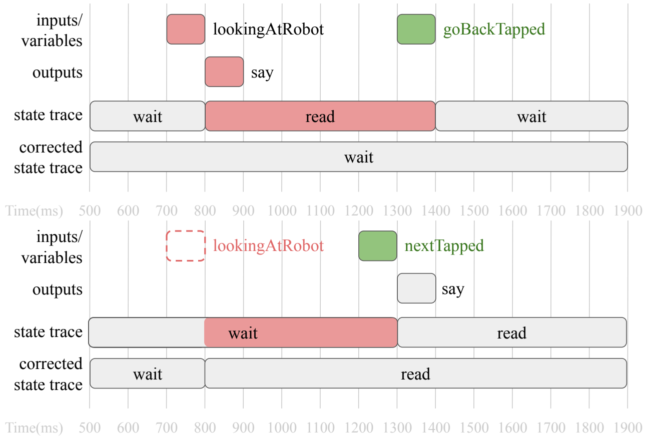


Fig. 4: Incorrect (above) and missing (below) transition errors and recovery examples from the storytelling robot example. Incorrect states and associated variables are highlighted in red. Implicit feedback in the user input is highlighted in green. The dotted red square represents an undetected human action.

overlap between the state trace of that interaction and the state trace of a program that behaves exactly as the programmer intends. Given a state trace, the programmer can annotate the incorrect states and correct the robot's state.

2) *Implicit state correction:* Correcting state traces of user interactions requires programmers to be in-the-loop after deployment. While feasible, this might be redundant in some cases where the interaction trace itself contains information about errors that occur. In particular, two types of transition errors that occur during FSM executions can cause errors in state traces.

1) *Incorrect transition*, which occur when the robot makes a transition to a different state when it was supposed to remain in the same one.
2) *Missed transition*, which occur when the robot remains in the same state when it was supposed to transition to a different one.

For instance, the robot in the storytelling example might make an incorrect transition from the `"wait"` state to the `"read"` state if `minSpeakingLevel` or `speakingWindow` is set too low, i.e., the robot mistakenly assumes the user finished speaking. These two types of transition errors can derail an interaction if no way to recover. An example of a recovery mechanism is the use of "Go back" and "Next" buttons (or other input commands, such as speech or gesture) that let the user to guide the interaction when faced with incorrect and missed transitions.

These mechanisms for repairing the interaction can also serve as implicit feedback for repairing the program; the use of interaction repair mechanisms in the state trace can be converted to corrected state traces usable by the repair algorithm. Alg. 1 shows how this is achieved. A user tapping "Go back" button indicates that the previous transition was incorrect; hence, all states between the last transition and the button tapping in the trace should be annotated as the previous state value. A user tapping the "Next" button indicates a missed transition; in this case, all past states within a predefined window size $\mathcal{W}$ should be annotated as the next state value.

Algorithm 1 ImplicitStateCorrection

```
Input: State trace $\mathbf{O} = [s_1, ..., s_\mathcal{T}]$ and window size $\mathcal{W}$
Output: Corrected state trace $\mathbf{O}'$
Initialize $j$ to 1
for $i = 2, ..., \mathcal{T}$ do
    if $s_i \neq s_{i-1}$ then
        if $GoBack(s_{i-1}) = s_i$ then
            $s_j \leftarrow s_i, ...s_{i-1} \leftarrow s_i$
        else if $Next(s_{i-1}) = s_i$ then
            $s_{\max(i-\mathcal{W},1)} \leftarrow s_i, ...s_{i-1} \leftarrow s_i$
        end if
        $j \leftarrow i - 1$
    end if
end for
$\mathbf{O}' = [s_1, ..., s_\mathcal{T}]$
```

C. Iterative Program Repair

Given a program sketch and corrections over program interaction traces, program repair aims to find the best set of program parameters represented by hole variables in the sketch. We present the first algorithm that achieves this by searching over all combinations of parameter value assignment combinations, as summarized in Alg. 2. Hole variables Θ_1 are initialized based on the variable distribution the programmer provides. The program is then executed to obtain an input trace I_1 and corrected state trace $\widehat{O}_1$. The main update step involves evaluating the following equation:

$$Repair(K, \mathbf{I}, \mathbf{O}) = \underset{\Theta}{\operatorname{argmax}} \sum_{j=1}^{|\mathbf{I}|} Overlap(K[H := \Theta](I_j), \widehat{O}_j).$$

Given an initial program sketch K and all program input traces $\mathbf{I} = (I_1, ..., I_i)$ and corrected state traces $\mathbf{O} = (\widehat{O}_1, ..., \widehat{O}_i)$ up to the current iteration i, the goal is to find the hole variable values that maximize the total overlap between corrected state traces $\mathbf{O}$ and output state traces from executing the program sketch with those hole variable values $K[H := \Theta]$. Intuitively, the algorithm searches for the hole values that make the program as consistent as possible with the corrected state traces.

This approach requires saving all traces of a program, which is not memory-efficient, and solving a harder satisfiability problem as increasing numbers of traces are obtained. Also, the user's behavior or the operation environment may change over time, in which case using all past traces could degrade performance. To address these issues, we propose an alternative algorithm, Alg. 3, that uses Bayesian filtering, a technique commonly applied to analyze sequential data. The main update step involves evaluating the following equation:

$$BayesRepair(K, I_i, \widehat{O}_i) = \underset{\Theta_i}{\operatorname{argmax}} \Pr(\Theta_i | \widehat{O}_i; K, I_i), \quad (1)$$

Algorithm 2 IterativeRepair

```
Input: Program sketch $K$ and initial values for holes $\Theta_1$
initialize $\mathbf{I}$ and $\mathbf{O}$ to empty sets
for $i = 1, 2, ...$ do
    run the program $K[H := \Theta_i]$ and record program inputs $I_i$ and output state trace $O_i$
    $\widehat{O}_i \leftarrow$ ImplicitStateCorrection($O_i$)
    add $I_i$ and $\widehat{O}_i$ into $\mathbf{I}$ and $\mathbf{O}$, respectively
    $\Theta_{i+1} \leftarrow Repair(K, \mathbf{I}, \mathbf{O})$
end for
```

Algorithm 3 IterativeBayesRepair

```
Input: Program sketch $K$ and prior $\Pr(\Theta_1)$
for $i = 1, 2, ...$ do
    run the program $K[H := \Theta_i]$ and record program inputs $I_i$ and output state trace $O_i$
    $\widehat{O}_i \leftarrow$ ImplicitStateCorrection($O_i$)
    $\Theta_i \leftarrow BayesRepair(K, I_i, \widehat{O}_i)$
    $\Pr(\Theta_{i+1}) \leftarrow \Pr(\Theta_i | \widehat{O}_i; K, I_i)$
end for
```

where:

$$\Pr(\Theta_i | \widehat{O}_i; K, I_i) = \frac{\Pr(\widehat{O}_i | \Theta_i; K, I_i) \Pr(\Theta_i)}{\sum_{j=1}^{|\mathbf{O}|} \Pr(\widehat{O}_j | \Theta_j; K, I_j) \Pr(\Theta_j)} \tag{2}$$

and the likelihood function is a percentage overlap function

$$\Pr(\widehat{O}_i | \Theta_i; K, I_i) \propto Overlap(K[H := \Theta_i](I_i), \widehat{O}_i). \tag{3}$$

At every iteration i, the algorithm first computes the posterior (2) and then uses a maximum a posteriori (MAP) estimation to determine hole variable values (1). The subsequent repair starts by setting the new prior to the current posterior using

$$\Pr(\Theta_{i+1}) \leftarrow \Pr(\Theta_i | \widehat{O}_i; K, I_i).$$

At a high level, IterativeBayesRepair compactly encodes information from the past into the prior $\Pr(\Theta_i)$ instead of computing it from stored data at every iteration. For the repair algorithms in both approaches, we use an exhaustive enumeration algorithm by binning parameter spaces into discrete sets.

We built the entire system in JavaScript. We used Cycle.js framework [39] to implement robot behaviors as Cycle.js applications and tools for recording and replaying interaction traces. The repair algorithms and ImplicitStateCorrection were implemented from scratch.

IV. Evaluation

To evaluate the feasibility and effectiveness of our approach, we conducted: (1) a simulation experiment using stochastic human simulators, (2) a human experiment with 10 participants and a tabletop robot.

A. Social Robot Tasks

We evaluated the repair of three human-robot interaction programs. Each represents an interaction pattern that is common across social robot application domains [35, 10, 2, 5, 36, 14], as described below:

1) Storytelling: The robot reads a story line-by-line until the story ends. The robot should pause the reading when the user looks away or interrupt the robot by speaking to it. The robot should resume the reading when the user looks at the robot and no longer speaks. During the interaction, the robot offers "Pause" and "Resume" buttons to let the user to manually control it in case it incorrectly pauses or resumes the reading. The storytelling scenario captures how robots handle *engagement and disengagement patterns* [5, 14].

2) Neck Exercise: The robot instructs the user to perform a sequence of neck exercises. It should display "Tilt your head to the LEFT" and move on to the next instruction when it detects the user has correctly followed the first one. The robot repeats the two different instructions three times (i.e., six instructions in total). During the interaction, the robot offers the "Next" and "Go back" buttons to let the user manually switch to the next or previous instruction in case the robot does not detect or incorrectly detects the user's action. The neck exercise represents *instruction and monitor patterns* [36] that require the robot to detect the human action, either to verify completion of an instructed task or as an input mechanism.

3) Open-Ended Q&A: The robot asks a series of open-ended questions. It displays each question on the screen and waits for the user's verbal response. While waiting, the robot should be aware of the user's use of gaze-aversion for holding the conversation floor [2]. When the robot detects the user has finished answering the question, it should ask the next one, for a total of five questions. During the interaction, the robot offers the "Next" and "Go back" buttons to let users manually switch to the next or previous question in case it mistakenly does or does not move on correctly. Open-ended Q&As represent the *turn-taking pattern*, which requires the robot to detect multi-model turn-yielding signals [35, 10, 2].

Each interaction program has a transition function that depends on features computed from continuous sensor streams of the robot. We repaired 5 (storytelling), 3 (neck exercise), and 7 (Q&A) FSM transition parameters across 2, 2, and 1 transitions, respectively.

B. Simulation Experiment

To systemically evaluate our approach with larger amounts of data, we created a human simulator to obtain FSM input traces and ground truth state traces for each interaction. The human simulators started with an initial human intention (e.g., engage or disengage), which changed its value after a time interval sampled from a predefined uniform distribution per interaction. We created ground truth state traces by sampling values from the simulated intention traces at the robot sensor streaming frequency (10hz). FSM input traces were created by simulating sensor input values using the uniform distributions defined per each intention and interaction. To emulate an experiment involving real human users, we selected simulator parameters based on data collected from author interactions with the robot for each interaction. For the storytelling, neck

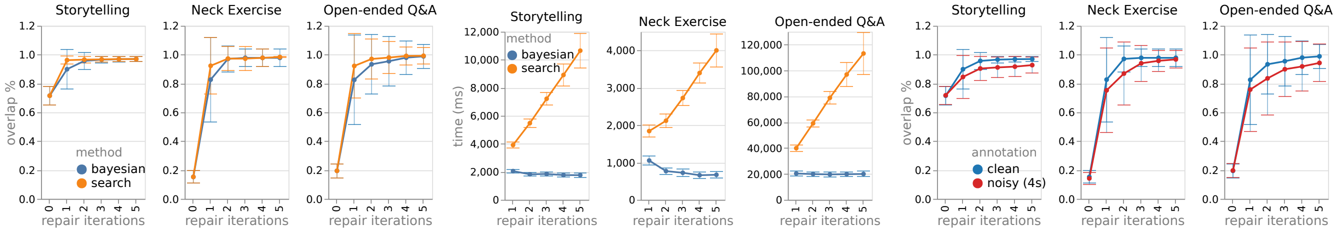


Fig. 5: Results from the simulation experiment. (left) Percentage overlap over five repair iterations using IterativeRepair (labeled as search) and IterativeBayesRepair (labeled as bayesian) algorithms across three tasks (storytelling, neck exercise, Q&A). (middle) Computation time of the algorithms for the same settings. (right) Overlap percentage over five repair iterations for the IterativeBayesRepair algorithm using clean and noisy state corrections for repair. Error bars show standard deviations.

exercise, and open-ended Q&A interactions, the human simulator stopped when it changed the ground truth states 5, 5, 6, respectively, and the average total duration of 100 simulated FSM input traces was 27.02 (SD = 5.53), 52.55 (SD = 5.05), 61.63 (SD = 6.78) seconds, respectively.

1) Procedure: We first simulated 100 input traces and ground truth state traces as a test dataset. At each iteration, we simulated a new FSM input trace and desired state trace pair and then applied the IterativeRepair (Alg. 2) and IterativeBayesRepair (Alg. 3) to acquire a set of repaired parameters. For the IterativeBayesRepair, we compared using the ground truth state trace to using the noisy state trace for repair, where the latter represented noisy corrections provided by the real programmer or user. The noisy state trace was computed by adding uniform noise of $[-2, 2]$ to every human intention change that occurred when generating the ground truth state (Sec. IV-B); total noise duration could at worst equal 37% of task duration.[1] The ±2 seconds window size was selected based on observing two first time users in an informal study. Repaired parameters were then used to produce a set of FSM output states using the set of FSM input traces in the test dataset. We initialized program parameters so the initial program always produced the missing transition errors. We repeated this procedure 64 times (using the same test dataset), with each run producing a different training input sequence.

2) Measures: We measured the interaction quality of the repaired programs as the percentage overlap (Eq. 3) between the output state trace and the ground truth state traces in the test dataset. Although this overlap had limitations—such as its sensitivity to the interaction length we controlled in this experiment—it captured interaction timing, an important factor in human-robot interaction [17]. We also measured the speed of repair algorithms in milliseconds to gauge the feasibility of repairing a program in front of a real user.

3) Results: IterativeRepair and IterativeBayesRepair algorithms performed similarly in terms of interaction quality(Fig. 5left). Both algorithms' average percentage overlaps increased monotonically over the five iterations, reached average overlaps above 95% by the third iteration, and produced standard deviations lower than 9% by the final iteration in all three tasks. The average computation time for IterativeBayesRepair remained constant over repair iterations, while that for IterativeRepair grew linearly (Fig. 5middle). Using the noisy state corrections decreased performance (Fig. 5right). Nonetheless, for all three tasks, the final iteration reached 91% mean overlap (with below 14% standard deviation).

C. Human Experiment

To evaluate the full implementation in a realistic setting, we conducted a human experiment. Ten participants interacted with a robot for the neck exercise or open-ended Q&A scenarios (Sec. IV-A) over four repair iterations.

1) Robot Platform: We used a custom-built social robot TaRo (**Ta**blet **Ro**bot). The TaRo consists of a face, GeeekPi 7inch touchscreen for displaying messages and facial expression and a neck, 5-DoF Open MANIPULATOR-X robot arm for making head gestures such as nod and shake. It also had access to a speaker, a webcam, and a microphone attached to the connected computer, which the robot used to play text-to-speech outputs or to estimate face poses of a human user (using PoseNet [30]) and loudness of the environment at the synchronized frequency of 10hz.

2) Procedure: The 10 participants (three females) were recruited from the University of Washington (UW) campus community through mailing lists. Their average age was 23.2 (SD = 5.58). Upon their arrival, the researcher explained the study's purpose, introduced the robot and demonstrated how to interact with it. The first five participants were asked to run the neck exercise and open-ended Q&A programs in order. The last five were asked to do the same in the reverse order. For each task, participants were asked to interact with the robot for four iterations and to reply to the questionnaire after each iteration. Over the four iterations, FSM transition parameters were repaired using IterativeBayesRepair. After each iteration, the desired state traces for IterativeBayesRepair were acquired from participants' recorded button tap inputs using ImplicitStateCorrection (Alg. 1). All transition parameters were initialized to make the robot not respond to users except when they tapped buttons like "Go back."

3) Measures: Objective measures of program interaction quality included (1) the percentage overlap before and after the repair, and (2) the number of user inputs needed to correct

[1] Calculated by 2s × 5 state changes/ 27.02s = 37% (storytelling)

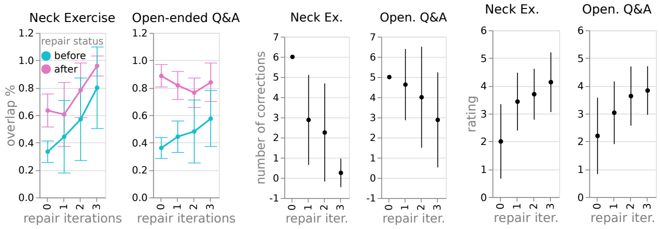


Fig. 6: Results from the user study of four repair iterations averaged across 10 participants. (left) Percentage overlaps before and after repair. (middle) Average number of interaction corrections (e.g., tapping "Go back") used as implicit feedback for repair. (right) Subjective ratings (reversed as appropriate, with higher values representing more fluency).

transition errors, i.e., button taps. Percentage overlaps were calculated using the state trace generated with the transition parameters before or after the repair and the desired state trace acquired from the human user in each iteration. We also gathered subjective measures of interaction quality, asking for user agreement with the statements:

- "I had to carry the weight to make the human-robot interaction better." (reverse scale)
- "I trusted the robot to do the right thing at the right time."
- "The robot contributed to the fluency of the interaction."

Available answer choices were 1 (strongly disagree) through 5 (strongly agree). We selected these questions from Hoffman's questionnaire, which was designed to measure the perceived fluency of a human-robot interaction [16]. To investigate the source of potential failures, we asked the open-ended question "What problems did you experience while interacting with the robot during the *taskname* activity?" in addition to the other questions after participants completed each task.

4) Results: All average objective measures of interaction quality, except after-repair percentage overlaps, increased over the four iterations of the two tasks (Fig. 6). We observed (1) a consistent increase in the average before-repair percentage overlap measures, (2) a consistent decrease in the average number of corrective human inputs measures, and (3) a consistent increase in the average subjective ratings between every pair of consecutive iterations. Between the first and last iterations, before-repair percentage overlaps increased from M = 0.33 & SD = 0.08 to M = 0.80 & SD = 0.30 for the neck exercise, and M = 0.36 & SD = 0.08 to M = 0.57 & SD = 0.21 for open-ended Q&As. In both scenarios, initial programs always required participants to tap a button to proceed, i.e., the number of corrective human inputs were 6 and 5, respectively, but the numbers dropped to 0.25 (SD = 0.25) and 2.88 (SD = 2.88), respectively, by the fourth iteration. Subjectively, participants did not initially perceive the interactions as being fluent (M = 2.00 & SD = 1.34 for neck exercise; M = 2.20 & SD = 1.37 for open-ended Q&A); however, they eventually perceived them as being slightly fluent by the last iteration (M = 4.13 & SD = 1.07 for neck exercise, M = 3.83 & SD = 0.88 for open-ended Q&A).

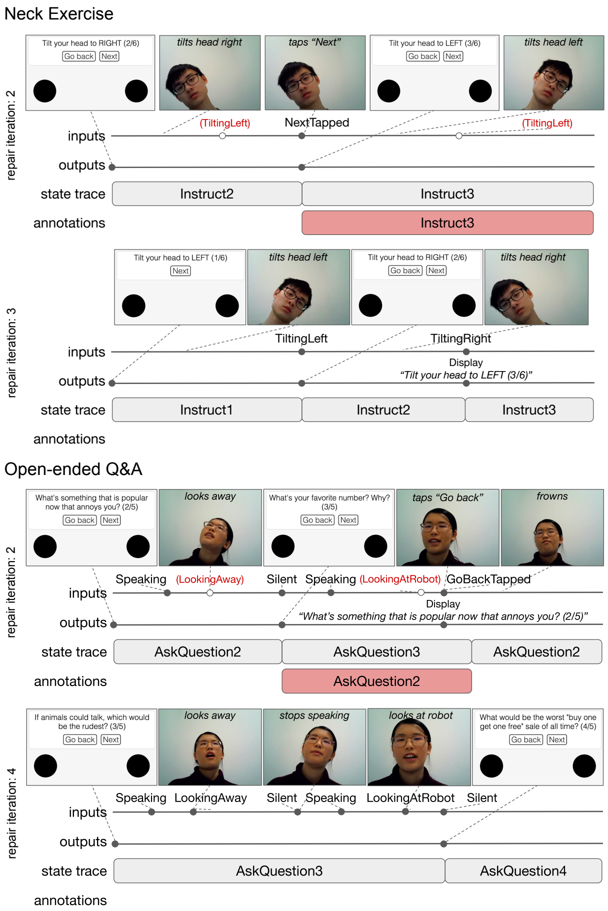


Fig. 7: Example interaction traces in the neck (top) and Q&A (bottom) scenarios that qualitatively demonstrate the impact of program repair. Undetected human inputs and state annotations are highlighted in red.

We analyzed the system logs of the three participants who thought the interaction had not improved by the last iteration. One said of the Q&A task, *"The robot over time got very fast in skipping questions while I'm still talking."* We observed that our approach did not improve interaction quality when unexpected sensor values occurred or the user did not fix incorrect transition errors. For example, some participants frequently moved out of the robot's field of view, while others did not tap the "Go back" button when the robot skipped to the next question; both behaviors caused the algorithm to find ineffective or even counterproductive parameters.

To demonstrate the qualitative differences in human-robot interactions before and after the repair, we present two pairs of interaction traces selected from the user study (Fig. 7). In the neck exercise example (above), the program parameters used in the second iteration made the robot incapable of detecting the user following the instruction action. The user noticed this and tapped the "Next" button to make the robot move onto the next state. After the repair in the third iteration, the robot was able to promptly move onto the next state when the

TABLE I: Percentage overlaps before and after the repair, and the number of human inputs measured in the behavior adaptation experiment.

repair iteration	1	2	3	4	5	6
environment description	quiet	quiet	quiet	noisy	noisy	noisy
% overlap before	33.12	99.88	99.92	17.90	57.70	75.42
% overlap after	94.56	99.88	99.92	99.97	99.92	99.93
number of human inputs	5	0	0	5	3	2

user followed the robot's instructions. In the open-ended Q&A example (below), the program parameters used in the second iteration made the robot unable to detect whether the human is not looking at the robot to hold the conversation floor and hence caused the robot to skip to the next question when the human stops speaking. The user then tapped the "Go back" button to continue answering now the previous question. The system used this input to annotate the part of state trace in which the robot accidentally moved onto the next state. After the repair in the fourth iteration, the user again looks away from the robot and stops speaking but the robot waits until the user looks back at the robot to move on to the next question.

5) Robot Behavior Adaptation Experiment: To demonstrate how our approach handled changes in environment that impact a program's interaction quality, we conducted an experiment involving one user who also participated in the previous human experiment. This user interacted with the robot that ran the open-ended Q&A FSM over six repair iterations; the first three occurred in a quiet room, and the last three in a noisy open indoor area. Between all iterations, we applied IterativeBayesRepair. We measured before and after percentage overlaps and the number of corrective human inputs. Table I shows these results. The interactive quality of the program quickly improved when the experiment took place in the quiet room (e.g., the required number of human inputs dropped from 5 to 0). The program performed poorly immediately after changing its environment, i.e., the % overlap dropped from 99.82% to 17.90%, and the number of human inputs increased from 0 to 5 but slowly improved in subsequent iterations. Based on analyzing the system log, we found that the improvement over the last three iterations was smaller and took a longer time than that over the first three iterations because (1) the noisy environment produced signals with larger variance, which made the repair more challenging, and (2) the system had to update the prior that was tuned for the first environment, i.e., it had to partially re-learn how to interact with humans.

V. Discussion and Future Work

We believe the results demonstrate the usefulness of the proposed approach, with some limitations.

In the human experiment results, we observed that three users did not fix the incorrect transition that occurred during the open-ended Q&A interaction. We anticipate that users will be more likely to provide adversarial feedback as a robot behavior FSM becomes more complex or an input mechanism for corrective feedback becomes more noisy. Hence, it will be necessary to identify robot behavior FSM design patterns or guidelines for creating interactions that avoid problematic feedback for repair tasks and to investigate intuitive recovery methods for resetting inappropriately repaired parameters.

While the proposed algorithms improved interactions measured in our experiments, we expect the following modifications will make them more useful. The proposed algorithm for realizing programmer-free repair, ImplicitStateCorrection, requires the window size parameter to correct the missing transition error. Investigating an effective way to remove this requirement, e.g., by using a credit assignment strategy instead, may help with handling more complex interactions. The complexity of the core IterativeBayesRepair uses exhaustive search, which grows exponentially with respect to the number of transitions, not the number of hole variables. Finding a more scalable inference algorithm or one that does not require any prior distributions for hole variables may increase usability and applicability.

Returning to the goal of our research (Fig. 1), we aim to make program repair ultimately benefit end-users who are not proficient software developers. To that end, further research in the workflow for enabling end-users to create and tune robot behaviors by themselves, e.g., by providing a graphical user interface for creating initial FSMs with a list of templates, is imperative. It is also important to further test our approach in the wild and for long-term scenarios where user preferences and environments may change over time to surface and investigate issues that arise.

VI. Conclusion

This paper presented an iterative program repair approach for creating robust and fluent social robot programs without painstakingly tuning program parameters. Our approach helps programmers to implement robot programs without complete low-level details and incrementally repair programs over time using corrective feedback provided by the programmer or the robot's user. We examined the feasibility and effectiveness of our approach via two experiments involving human simulators and 10 real human users across three representative social robot use cases; results demonstrate the utility and potential of the proposed approach.

Acknowledgments

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 Rajesh P.N. Rao for discussions that inspired many ideas during early phases of the project.

References

[1] Philipp Allgeuer and Sven Behnke. Hierarchical and state-based architectures for robot behavior planning and control. *arXiv preprint arXiv:1809.11067*, 2018.

[2] Sean Andrist, Xiang Zhi Tan, Michael Gleicher, and Bilge Mutlu. Conversational gaze aversion for humanlike

robots. In *International Conference on Human-Robot Interaction*, pages 25–32. ACM, 2014.

[3] Emilia I Barakova, Jan CC Gillesen, Bibi EBM Huskens, and Tino Lourens. End-user programming architecture facilitates the uptake of robots in social therapies. *Robotics and Autonomous Systems*, 61(7):704–713, 2013.

[4] Jonathan Bohren and Steve Cousins. The smach high-level executive [ros news]. *Robotics & Automation Magazine*, 17(4):18–20, 2010.

[5] Dan Bohus and Eric Horvitz. Managing human-robot engagement with forecasts and... um... hesitations. In *International Conference on Multimodal Interaction*, pages 2–9. ACM, 2014.

[6] Rafael H Bordini, Michael Fisher, and Maarten Sierhuis. Formal verification of human-robot teamwork. In *International Conference on Human-Robot Interaction*, pages 267–268. ACM/IEEE, 2009.

[7] Adrian Boteanu, Jacob Arkin, Siddharth Patki, Thomas Howard, and Hadas Kress-Gazit. Robot-initiated specification repair through grounded language interaction. *arXiv preprint arXiv:1710.01417*, 2017.

[8] Rodney Brooks. A robust layered control system for a mobile robot. *Journal on robotics and automation*, 2(1): 14–23, 1986.

[9] Sebastian G Brunner, Franz Steinmetz, Rico Belder, and Andreas Dömel. Rafcon: A graphical tool for engineering complex, robotic tasks. In *International Conference on Intelligent Robots and Systems*, pages 3283–3290. IEEE, 2016.

[10] Crystal Chao and Andrea L Thomaz. Controlling social dynamics with a parametrized model of floor regulation. *Journal of Human-Robot Interaction*, 2(1):4–29, 2013.

[11] James Diprose, Bruce MacDonald, John Hosking, and Beryl Plimmer. Designing an api at an appropriate abstraction level for programming social robot applications. *Journal of Visual Languages & Computing*, 39:22–40, 2017.

[12] Bernard Espiau, Konstantinos Kapellos, and Muriel Jourdan. Formal verification in robotics: Why and how? In *Robotics Research*, pages 225–236. Springer, 1996.

[13] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. Temporal logic motion planning for mobile robots. In *International Conference on Robotics and Automation*, pages 2020–2025. IEEE, 2005.

[14] Dylan F Glas, Takayuki Kanda, and Hiroshi Ishiguro. Human-robot interaction design using interaction composer eight years of lessons learned. In *International Conference on Human-Robot Interaction*, pages 303–310. ACM/IEEE, 2016.

[15] Jenna Claire Hammond, Joydeep Biswas, and Arjun Guha. Automatic failure recovery for end-user programs on service mobile robots. *arXiv preprint arXiv:1909.02778*, 2019.

[16] Guy Hoffman. Evaluating fluency in human–robot collaboration. *IEEE Transactions on Human-Machine Systems*, 49(3):209–218, 2019.

[17] Guy Hoffman, Maya Cakmak, and Crystal Chao. Timing in human-robot interaction. In *International Conference on Human-robot interaction*, pages 509–510. ACM/IEEE, 2014.

[18] Jarrett Holtz, Arjun Guha, and Joydeep Biswas. Interactive robot transition repair with smt. In *International Joint Conference on Artificial Intelligence*, pages 4905–4911.

[19] Jarrett Holtz, Arjun Guha, and Joydeep Biswas. Smt-based robot transition repair. *arXiv preprint arXiv:2001.04397*, 2020.

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

[21] Alap Kshirsagar, Hadas Kress-Gazit, and Guy Hoffman. Specifying and synthesizing human-robot handovers. In *International Conference on Intelligent Robots and Systems*. IEEE/RSJ, 2019.

[22] Alyssa Kubota, Emma IC Peterson, Vaishali Rajendren, Hadas Kress-Gazit, and Laurel D Riek. Jessie: Synthesizing social robot behaviors for personalized neurorehabilitation and beyond.

[23] Wenchao Li, Dorsa Sadigh, S Shankar Sastry, and Sanjit A Seshia. Synthesis for human-in-the-loop control systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–484, 2014.

[24] Marco Manca, Fabio Paternò, and Carmen Santoro. Analyzing trigger-action programming for personalization of robot behaviour in iot environments. In *International Symposium on End User Development*, pages 100–114, 2019.

[25] Spyros Maniatopoulos, Philipp Schillinger, Vitchyr Pong, David C Conner, and Hadas Kress-Gazit. Reactive high-level behavior synthesis for an atlas humanoid robot. In *International Conference on Robotics and Automation*, pages 4192–4199. IEEE, 2016.

[26] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *International Conference on Intelligent Robots and Systems*, pages 3869–3876. IEEE/RSJ, 2017.

[27] Srinivas Nedunuri, Sailesh Prabhu, Mark Moll, Swarat Chaudhuri, and LE Kavraki. Smt-based synthesis of integrated task and motion plans for mobile manipulation. In *International Conference on Robotics and Automation*. IEEE, 2014.

[28] Hai Nguyen, Matei Ciocarlie, Kaijen Hsiao, and Charles C Kemp. Ros commander (rosco): Behavior creation for home robots. In *International Conference on Robotics and Automation*, pages 467–474. IEEE, 2013.

[29] Scott Niekum, Sarah Osentoski, George Konidaris, Sachin Chitta, Bhaskara Marthi, and Andrew G Barto. Learning grounded finite-state representations from unstructured demonstrations. *The International Journal of Robotics Research*, 34(2):131–157, 2015.

[30] D Oved. Real-time human pose estimation in the browser with tensorflow.js. *TensorFlow Medium, May*, 2018.

[31] David Porfirio, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. Authoring and verifying human-robot interactions. In *Symposium on User Interface Software and Technology*, pages 75–86. ACM, 2018.

[32] David Porfirio, Evan Fisher, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. Bodystorming human-robot interactions. In *Symposium on User Interface Software and Technology*, 2019.

[33] Emmanuel Pot, Jérôme Monceaux, Rodolphe Gelin, and Bruno Maisonnier. Choregraphe: a graphical tool for humanoid robot programming. In *The International Symposium on Robot and Human Interactive Communication*, pages 46–51. IEEE, 2009.

[34] Mattia Racca, Ville Kyrki, and Maya Cakmak. Interactive tuning of robot program parameters via expected divergence maximization.

[35] Charles Rich, Brett Ponsler, Aaron Holroyd, and Candace L Sidner. Recognizing engagement in human-robot interaction. In *International Conference on Human-Robot Interaction*, pages 375–382. ACM/IEEE, 2010.

[36] Allison Sauppé and Bilge Mutlu. Robot deictics: How gesture and context shape referential communication. In *International Conference on Human-Robot Interaction*, pages 342–349. ACM, 2014.

[37] Philipp Schillinger, Stefan Kohlbrecher, and Oskar von Stryk. Human-robot collaborative high-level control with application to rescue robotics. In *International Conference on Robotics and Automation*, pages 2796–2802. IEEE, 2016.

[38] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[39] André Staltz. Cycle.js. https://cycle.js.org/, 2016. Accessed: 2019-10-1.

[40] Jana Tumova and Dimos V Dimarogonas. Multi-agent planning under local ltl specifications and event-based synchronization. *Automatica*, 70:239–248, 2016.

[41] Matt Webster, Clare Dixon, Michael Fisher, Maha Salem, Joe Saunders, Kheng Lee Koay, Kerstin Dautenhahn, and Joan Saez-Pons. Toward reliable autonomous robotic assistants through formal verification: a case study. *Transactions on Human-Machine Systems*, 46(2):186–196, 2015.