Principles of Robot Programming Systems

Jacob W. Crandall, Vimitha Manohar, and Issak Gezehei
Masdar Institute of Science and Technology
Abu Dhabi, UAE
{jcrandall, vmanohar, igezehei}@masdar.ac.ae

Alan Atherton
5D Robotics
San Diego, California, USA
jalanatherton@gmail.com

Michael A. Goodrich
Brigham Young University
Provo, Utah, USA
mike@cs.byu.edu

Abstract—Despite substantial advances in robotics technologies, robots continue to be underutilized. This trend is likely to continue for the foreseeable future unless end users can better program their own robots. Robot programming systems must be developed that allow end users (who often lack technological expertise) to quickly program their own robots to perform customized tasks in environments of their choosing. Despite this need, we are not aware of a formal agenda for the development of such systems. In this paper, we begin to define a set of performance standards that robot programming systems should achieve. To demonstrate the usefulness of these standards, we describe a user study in which we evaluated several robot programming systems using several metrics defined in these standards.

I. INTRODUCTION

Robotics technologies have improved substantially over the last few decades. Robots are now commonly used in space exploration (e.g., [1][2]), law enforcement and national defense, medical and health care applications (e.g., [3][4]), entertainment (e.g., [5][6]), and search and rescue (e.g., [7]). Despite these broad uses, robots are still mostly expensive, domain- and environment-specific, tools.

One primary reason for this limitation is that robots are difficult to program. As an example, researchers and clinicians at Brigham Young University recently conducted two long-term studies in which a robot was used in therapy provided to children with autism spectrum disorder [4][8]. In post-study interviews, the clinicians indicated that the greatest obstacle for using robots in therapy is that it takes too much time and effort to change robot behavior. They also indicated that many of the robot behaviors they had available lacked energy and were not sufficiently expressive. They needed behaviors that better suited the children with whom the robots interacted, but did not have means to easily create these behaviors themselves.

Such experiences illustrate that end users must be able to program their own robots [9][10][11]. Robot programming systems (RPSs) need to be developed that permit end users who lack technological expertise to quickly program their own robots to perform customized tasks in environments of their own choosing. It is apparent that successful RPSs will carefully combine advanced interface technologies with machine learning and other forms of artificial intelligence.

In recognition of the need to allow end users to program robots, research on RPSs has recently increased dramatically. Areas of focus have included programming by demonstration (also called learning from demonstration and teaching by demonstration; e.g., [12][13][14]), visual programming (e.g., [15][16][17][18]), and human motion capture (e.g., [19][20][21]). Despite the popularity of these methods, research evaluating how well end users can program robots using these technologies is uncommon. Several exceptions exist (e.g., [10][11]), though a comprehensive set of standards for what these RPSs should achieve has still not been articulated.

In this paper, we begin to define formal standards for evaluating RPSs. In so doing, we define a set of challenges. Each challenge is linked to a set of metrics that can be used to measure the success of RPSs and to help diagnose the shortcomings of these systems. We illustrate the use of some of these metrics via a user study in which we compare and contrast several programming methods used in existing RPSs.

II. ROBOT PROGRAMMING SYSTEMS

We begin by identifying basic terminology, components, and processes typical of RPSs.

A. Definition

RPSs are designed to allow end users to easily program customized behavior rules for their own robots. A behavior rule specifies the robot’s output (typically sequences of movements and audio expressions), which is potentially contingent on stimuli from the environment. We refer to a complete sequence of actions as a robot behavior.

An RPS has four interacting components: the programmer (hereafter referred to as the user or end user), the interface, the robot, and processing algorithms (Fig. 1). The end user begins with an idea of the desired robot behavior, which she communicates to the robot via the interface. Intuitive mechanisms for communicating the desired behavior typically require algorithms that interpret, process, and synthesize the user’s input. For example, some RPSs require algorithms to (1) identify the end user’s gestures, (2) morph human movements into robot movements, and/or (3) produce robot

Fig. 1. A RPS consists of four components: (1) the programmer (i.e., the end user), (2) the interface, (3) processing algorithms, and (4) the robot. Arrows indicate interactions among the components.
behavior choices for the user from sensor data. Each of these algorithms can introduce delays and errors that must be dealt with. Processing algorithms can either be run onboard the robot or on an intermediary computer. Given the input from the user and the processing algorithms, the robot computes its behavior. The currently defined robot behavior is communicated to the end user via the human-robot interface.

B. Typical Programming Process

The programmer, processing algorithms, and robot interact to create a robot behavior by iteratively transitioning between three stages: behavior creation, playback and testing, and behavior editing (Fig. 2). In behavior creation, an initial robot behavior is created. The user then usually plays back the behavior to evaluate its quality. If unsatisfied, the user moves to a behavior-editing stage or recreates the behavior. This process continues until the user is satisfied or gives up trying.

C. Attributes of Robots and Robot Behaviors

The appropriateness of the programming methods used in RPSs are contingent on attributes of the robot and the desired robot behavior. Programming methods that are effective for creating some forms of behaviors for some robots may not be suitable for creating other forms of behaviors for other robots. Thus, evaluations of programming methods should be made in the context of these attributes. With this in mind, we have begun to characterize common, relevant attributes of both robots and robot behaviors in Tables I and II.

Table I illustrates some commonly relevant attributes of robots to RPSs. For example, the robot’s output DOFs impact the effectiveness of input methods for programming by demonstration. If a robot has more DOFs than a user can demonstrate at one time, learning by demonstration becomes more complex. Thus, since whole-body motion capture typically allows the user to simultaneously demonstrate movements for more DOFs than kinesthetic teaching, motion capture may be more appropriate for robots with high DOFs. On the other hand, motion capture is typically only appropriate when mappings between human and robot movements are easy to derive (such as with humanoids). Thus, kinesthetic teaching may be more appropriate than motion capture for robots whose forms do not closely resemble humans.

Robot behaviors also differ along several dimensions (Table II). For example, the output DOFs required by a robot behavior can vary substantially. Programming methods devised for behaviors requiring movements along only a few DOFs may not be appropriate for behaviors that require movements along many DOFs and vice versa. Second, some robot behaviors are open loop; they require no interaction with the environment. We refer to such behaviors as rote behaviors, or behaviors in which the robot simply executes a sequence of actions or movements. Such behaviors are typically much easier to program than closed-loop behaviors, which we call reactive behaviors. In reactive behaviors, the robot varies its actions based on its sensor readings. Reactive behaviors become even more complex when they require interactions with collaborating partners. We call these reactive behaviors collaborative behaviors. Finally, the spatiotemporal span of the intended behavior is also important. Behaviors that take longer to execute (behavior length) and that span multiple strategic and operational uses are likely to be more difficult to create (and may require higher degrees of testing and editing) than behaviors that are tailored for very domain- and environment-specific use.

III. Performance Standards

Successful RPSs must provide end users with means to quickly program effective robot behaviors in the domains and environments of their choosing. To do this, we have identified four challenges that RPSs must meet (Table III). In this section, we discuss each of these challenges. In so doing, we identify properties that should be measured to effectively evaluate RPSs with respect to each challenge. For some challenges, we propose new evaluation metrics that have not previously been used to evaluate RPSs.

A. Programming Time

The time required of end users to create effective robot behaviors, or programming time, is of upmost importance. While this has been an implicit goal of RPSs since their inception, few evaluations of RPSs or related methods actually report measures of programming time. Such analysis should become standard. Programming time should be evaluated with respect to two objectives. First, an RPS should facilitate low programming times. Second, programming time should scale gracefully as the complexity of the intended behavior increases. These two goals can be captured by a complexity (“Big-O”) analysis of programming time as a function of the attributes of the intended robot behavior $B$.

As with algorithmic run-time analysis, evaluations of both best-case and average-case programming time can be useful.
**TABLE III.** **SUCCESSFUL RPSs MUST ADDRESS FOUR CHALLENGES.**

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming time</td>
<td>- Best-case programming time: $PT^*(B)$</td>
</tr>
<tr>
<td></td>
<td>- Average-case programming time: $\overline{PT}(B)$</td>
</tr>
<tr>
<td></td>
<td>- RPS efficiency ratio: $ER(T) = \frac{\overline{PT}(B)}{PT^*(B)}$</td>
</tr>
<tr>
<td>Behavior quality</td>
<td>- Completion rate, discernibility</td>
</tr>
<tr>
<td></td>
<td>- Intercant assessments: aesthetics, expressiveness</td>
</tr>
<tr>
<td>Sustained interaction</td>
<td>- Persistence rate</td>
</tr>
<tr>
<td>with the programmer</td>
<td>- Programmer experience: idea transfer,</td>
</tr>
<tr>
<td></td>
<td>- creativity boosting, subjective metrics</td>
</tr>
<tr>
<td>Safety</td>
<td>- Compliance of mechanisms</td>
</tr>
<tr>
<td></td>
<td>- Safety-margin guarantees: torque limits,</td>
</tr>
<tr>
<td></td>
<td>- absence of pinch points, etc.</td>
</tr>
</tbody>
</table>

*Best-case programming time*, denoted $PT^*(B)$, is the minimum possible time required to create (using the RPS) a robot behavior $B$ as a function of the behavioral attributes of $B$. It is typically computed by considering an ideal user who does not need to engage in testing or behavior editing (Fig. 2). Since $PT^*(B)$ is a lower bound on programming time, it is useful for assessing whether a particular programming method can possibly lead to satisfactory results in a timely fashion. Note that $PT^*(B)$ is related to Kolmogorov Complexity [22].

On the other hand, *average-case programming time*, denoted $\overline{PT}(B)$, is the average amount of time taken by end users from a particular demographic to create robot behaviors with attributes similar to $B$. Given that users will typically need to test and modify the behaviors they create (Fig. 2), $\overline{PT}(B)$ is a more comprehensive evaluation of an RPS than $PT^*(B)$.

Effective interfaces and processing algorithms help the user to minimize $\overline{PT}(B)$ such that the difference between $\overline{PT}(B)$ and $PT^*(B)$ is not substantial. This gives rise to a third metric of programming time in which we compare $\overline{PT}(B)$ to $PT^*(B)$. Formally, we define the RPS efficiency ratio as:

$$ER(B) = \frac{\overline{PT}(B)}{PT^*(B)}.$$  \hspace{1cm} (1)

Effective RPSs should be designed so that $ER(B)$ is as close to unity as possible for all $B$.

**B. Behavior Quality**

Behavior quality has been the primary consideration in the evaluation of RPSs to date (e.g., [10, 11]). While appropriate metrics for evaluating behavior quality are application specific, several forms of metrics have become standard. A first set of metrics define the success of the behavior the user has programmed, such as how well the robot behavior leads to successful completion of the intended task. A second set of metrics concerns the perceptions of the behavior by interactants (the robot’s collaborators during run-time). Such metrics include the expressiveness, the aesthetics, and the discernibility of intent of the behavior.

Though RPSs have matured substantially, several recent studies indicate that substantial progress remains to be made [10, 11]. These studies show that even rote behaviors created by existing RPSs have high failure rates. Furthermore, more complex behaviors (such as collaborative behaviors) for sophisticated robots have not commonly been programmed with RPSs so far.

**C. Sustained Interaction with the Programmer**

An RPS will only be widely used if users have positive experiences when using it. These positive experiences must occur soon after the user begins programming behaviors with the system. Fig. 3 compares behavior quality when using two hypothetical RPSs as a function of the number of behaviors created by a user. While people are able to eventually produce better behaviors with RPS A than with RPS B, we anticipate that RPS B would be more easily accepted than RPS A. Typical users are unlikely to persist in using RPS A long enough to create behaviors that realize RPS A’s higher potential. Thus, a primary challenge is to create RPS that users continue to utilize. We refer to this metric as persistence rate, defined as the median number of behaviors a user creates with the RPS.

An RPS’s persistence rate is contingent on many things. One such aspect is idea transfer, or how well the RPS allows the user to transfer his or her creative vision to the robot. However, our observations of people creating robot behaviors in user studies causes us to believe that high idea transfer is necessary, but insufficient. Many individuals appear to fail to formulate ideas of effective robot behavior, particularly when behaviors require an artistic touch, such as in expressions of emotion. Thus, effective RPS should use mediums and processes that enhance the user’s creative vision. This idea is captured by the dotted arrow in Fig. 1. We call this concept creativity boosting. For example, we have observed that procedural characteristics used in RPSs can impact the quality of verbal behaviors users record [11].

Since idea transfer and creativity boosting are sometimes difficult to measure, we must often settle for subjective metrics in which users provide subjective ratings in questionnaires.

**D. Safety**

Industrial-strength robot behaviors disseminated to end users undergo careful analysis and testing. These standards address compliance of mechanisms, torque limits, and the absence of pinch points [23, 24]. However, robot behaviors created by RPSs do not typically undergo the same degree of testing. A traditional, extensive testing process would likely be too costly to satisfy programming time requirements. However, behaviors generated by RPSs must meet the same safety standards required of behaviors develop by other means. Thus, new methods for testing RPSs to ensure compliance with safety standards are needed. The lack of such methods presents a serious challenge that must be resolved if RPSs are to play a prominent role in the development of robot technologies.
IV. Case Study

To begin to demonstrate the usefulness of the standards proposed in the previous section, we compare four different methods of programming by demonstration in the creation of rote behaviors using several of the metrics we have discussed. These programming methods differ by input type (human motion capture and kinesthetic teaching) and demonstration style (trajectory- and keyframe-style demonstrations). We refer to the four resulting programming methods as HMC-Trajectory, HMC-Keyframe, KT-Trajectory, and KT-Keyframe.

Several previous studies have compared subsets of these programming methods. For example, Akgun et. al [10] compared the behavior qualities of single-arm behaviors programmed using KT-Trajectory and KT-Keyframe. Comparisons of HMC-Trajectory and HMC-Keyframe for animating software characters have also been performed [25]. As a point of distinction, we focus primarily on evaluating these systems with respect to best-case and average-case programming time, which was not done in these past evaluations. We do, however, also report evaluations of behavior quality and user experience.

A. Best-Case Programming Time

We derive best-case programming times for trajectory- and keyframe-style demonstrations separately. Both derivations are applicable for KT and HMC.

1) Trajectory-style demonstrations: In trajectory-style demonstrations, the user demonstrates the continuous movements of each DOF on the robot from start to finish. As such, the user must perform a demonstration of length \( \alpha T_B \) for each DOF used in the behavior. Here, \( T_B \) is the behavior length of \( B \), or the time required for the behavior to execute, and \( \alpha \) is the ratio of the length of the demonstration to \( T_B \). That is, \( \alpha = 1 \) if the robot behavior is played at the demonstrated speed, \( \alpha > 1 \) if the robot moves faster than the demonstrated speed, and \( \alpha < 1 \) otherwise.

The input method dictates the number of DOFs that the user can demonstrate at one time (denoted \( D_I \)). Thus, the minimum number of demonstrations that the user must supply to express a complete behavior is\( \left\lceil \frac{D_B}{D_I} \right\rceil \), where \( D_R \) is the number of DOFs used in the robot behavior. Thus, the best-case programming time of trajectory-style demonstrations is

\[
\mathcal{P}T_{\text{traj}}^*(B) = \alpha T_B \left\lceil \frac{D_R}{D_I} \right\rceil .
\]

This lower bound illustrates that programming time using trajectory-style demonstrations grows linearly in \( T_B \) at best.

2) Keyframe-style demonstrations: In keyframe-style demonstrations (or keyframing), the user demonstrates the robot’s pose at various points in time. The robot then automatically generates movements that traverse these poses. Thus, best-case programming time depends on (1) the number of keyframes that are required \( N_{\text{frames}} \) and (2) the time required to create a single keyframe \( T_{\text{pose}} \). Thus, the best-case programming time of keyframing is

\[
\mathcal{P}T_{\text{key}}^*(B) = N_{\text{frames}} T_{\text{pose}}.
\]

It is difficult to determine \( N_{\text{frames}} \) before a behavior is created. Longer behaviors will typically have more keyframes, but it is possible for a complex short behavior to have more keyframes than a long simple behavior. Thus, we use animation complexity \( \phi_B \), as a heuristic to determine \( N_{\text{frames}} \), where \( \phi_B > 0 \) is proportional to the rate of change in acceleration for each joint in an animation. Intuitively, most methods for interpolating movements between keyframes are restricted to smooth movements; non-smooth movements (i.e., those with sudden changes in acceleration) require additional keyframes. Thus, we have that \( N_{\text{frames}} = \phi_B T_B \).

\( T_{\text{pose}} \), the time necessary to create a single keyframe, has two components. The first component is the time it takes the user to select where a keyframe should be placed. Various methods exist in the literature for making this decision, but we neglect this component since the second component, the time required to capture a pose (denoted \( T_{\text{capture}} \)), is more costly. \( T_{\text{capture}} \) consists of the time it takes to move all of the robot’s parts into position and issue the command to save the pose. As in the case of trajectory-style demonstrations, the user cannot always move the entire robot at once; it takes at least \( \left\lceil \frac{D_B}{D_I} \right\rceil \) passes to do so. Thus, \( T_{\text{pose}} = T_{\text{capture}} \left\lceil \frac{D_B}{D_I} \right\rceil \).

Substituting these quantities into Eq. (2), we get

\[
\mathcal{P}T_{\text{key}}^*(B) = \phi_B T_B \left( T_{\text{capture}} \left\lceil \frac{D_R}{D_I} \right\rceil \right) .
\]

Given that \( T_{\text{capture}} \) is constant for a given robot and input method, best-case programming time using keyframing is linear in behavior length \( (T_B) \) and animation complexity \( \phi_B \).

B. User Study

Though the best-case programming times of trajectory- and keyframe-style demonstration methods can be linear in \( T_B \), average-case programming times can differ substantially. Thus, we also evaluate the average-case programming time of these programming methods via user study. In the study, 24 graduate students (with a mean age of 25.4) programmed rote behaviors for a Nao robot. None of the students had previous experience using these systems, though two users had limited experience programming robots.

The participants programmed Nao using HMC-Trajectory, HMC-Keyframe, KT-Trajectory, and KT-Keyframe. These programming methods were implemented via Choregraphe, which is shipped with Nao. To focus on the four programming methods, the experiment administrators performed the low-level functions for the participants in Choregraphe. The participants indicated verbally to the administrators when to start and stop demonstrations, and when to record keyframes. Human motion capture was achieved using a Kinect sensor.

Each participant was assigned to use either trajectory- or keyframe-style demonstrations (12 subjects assigned to each type). Using the assigned demonstration style, each participant programmed four robot behaviors, two using KT and two using HMC. The four robot behaviors were selected from six rote behaviors consisting solely of upper-body movements (arms and head). These six behaviors were (1) clapping three times, (2) waving, (3) signaling the letter-sequence YMCA, patterned after the 1978 disco song, (4) playing peek-a-boo, (5) performing an eight-step exercise which consisted of moving between scripted poses with crisp and coordinated head and arm movements, and (6) performing a head-bang...
dance consisting of both head and arm movements. The first three behaviors require only arm movements. We refer to these behaviors as simple behaviors. The last three behaviors (called complex behaviors) require movement of the robot’s arms and head. Sample still poses of the robot performing each of the six behaviors are shown in Fig. 4.

1) Average-case programming time: This user study allows us to observe average-case programming times using each programming method. These times are shown in Fig. 5(a). The average lengths of these behaviors are shown in Fig. 5(b). These figures show that it took substantially less time to program longer behaviors using trajectory-style demonstrations than keyframes. Furthermore, we observe that programming time for simple and complex behaviors was nearly identical when using HMC. However, participants took substantially less time to program simple behaviors than complex behaviors when using KT.

Both of the trends are captured by computing the best-case programming times of these behaviors (using Eqs. 2 and 4), which are shown in Fig. 6. As with the average-case programming time, the best-case programming times for these behaviors is substantially higher for keyframe-style demonstrations than for trajectory-style demonstrations. Furthermore, the best-case model shows little difference in programming times when using HMC. However, because the user only has two hands, the input DOFs ($D_I$) for KT is less than the require DOFs in the complex behaviors ($D_R$), but not the simple behaviors. Thus, the programming time is twice as high for complex behaviors then simple behaviors when using KT (as predicted by Eq. 2). These results help to validate the usefulness of best-case programming time models in evaluating the potential of programming methods.

Average-case and best-case programming times can be used to compute RPS efficiency ratios (Eq. 1) for each system and behavior complexity. These ratios give an indication of the efficiency of the interfaces and processing algorithms offered to users, with higher values indicating less efficient RPSs. Fig. 7 shows that these systems could be improved substantially, though less so for KT-Keyframe. The figure also shows that keyframing was more efficient than trajectory-style demonstrations in our study, due at least in part to the fact that users tended to return to behavior creation less often when using keyframes (Fig. 5(c)).

2) Behavior quality and user experience: We also collected subjective ratings of behavior quality and user experience.
Participants rated each of the behaviors they created using a 5-point Likert scale (5=Excellent, 3=Effective, 1=Unsatisfactory). An expert panel also rated the correctness of the behaviors on a similar scale. The participants were also asked to rate each of the systems they used on the scale 1-5 for an indication of user experience (system rating).

The results are summarized in Table IV. Despite high programming times, behavior quality and correctness was highest for behaviors created with KT-Keyframe. On the other hand, users rated the HMC-Keyframe and KT-Trajectory systems the highest. Two-way ANOVAs show main affects for other hand, users rated the HMC-Keyframe and KT-Trajectory system rating

<table>
<thead>
<tr>
<th>Metric</th>
<th>HMC-Trajectory</th>
<th>HMC-Keyframe</th>
<th>KT-Trajectory</th>
<th>KT-Keyframe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior correctness</td>
<td>2.35</td>
<td>2.92</td>
<td>3.49</td>
<td>3.85</td>
</tr>
<tr>
<td>Behavior quality</td>
<td>2.72</td>
<td>3.00</td>
<td>3.63</td>
<td>3.83</td>
</tr>
<tr>
<td>System rating</td>
<td>3.17</td>
<td>4.00</td>
<td>4.00</td>
<td>3.58</td>
</tr>
</tbody>
</table>

V. CONCLUSIONS

In this paper, we have begun to propose a set of standards to inform the development of robot programming systems (RPSs). These standards address the quality of robot behaviors the user should be able to develop, the safety characteristics of these robot behaviors, the time required to program the behaviors, and the ability of RPSs to sustain interactions with the programmer. We then demonstrated how these standards can be used to evaluate several methods for programming by demonstration. We anticipate that continuing to validate and refine such standards, and then using such standards to evaluate and inform the development of future RPSs will facilitate more wide-spread use of robotic technologies.

REFERENCES