# Towards Intuitive Robot Programming Using Finite State Automata *

Lukas Sauer[1], Dominik Henrich[1] and Wim Martens[2]

[1] Lehrstuhl Angewandte Informatik 3 (Robotik und Eingebettete Systeme)
[2] Lehrstuhl Angewandte Informatik 7 (Theoretische Informatik)
Universität Bayreuth, Germany
firstname.lastname@uni-bayreuth.de

**Abstract.** This paper describes an approach to intuitive robot programming, with the aim of enabling non-experts to generate sensor-based, structured programs. The core idea is to generate a variant of a finite state automaton (representing the program) by kinesthetic programming (physically guiding the robot). We use the structure of the automaton for control flow (loops and branching according to conditions of the environment). For programming, we forgo a visual user interface completely to determine to what extent this is viable. Our experiments show that non-expert users are indeed able to successfully program small sample tasks within reasonable time.

**Keywords:** Robotics, Finite State Automata, Programming

## 1 Introduction

While robots are used most in large scale industry, their future use is expected to comprise fields where no expert programmers are available, e.g. small and medium-sized enterprises, workshops, or private households. We describe an approach extending playback programming (*guiding* in [13]) by simple means of program structuring, based on the model of finite state automata, thus taking an automata theory point of view to robot programming. It aims to enable intuitive programming of sensor-based, structured robot programs by non-expert users. Furthermore, we do not use a textual or graphical user interface. This means that no monitor is required, reducing the amount of necessary hardware both to acquire and to set up in the workspace. While personal devices like smartphones are present for many users, these are generally too small for more complex interfaces and program representations. Interaction in our approach takes place solely

* This work has partly been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant agreement He2696/15 INTROP. We acknowledge Katharina Barth, who developed a preliminary version of the presented work and code base upon which we could build.
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-30179-8_25

via the robot (and a small number of hardware buttons, similar to the controls of a media player). Our hypothesis is that, for programming a task known to the user, the visible state of robot and environment is sufficient. In that case, the absence of a textual or graphical interface allows the user to focus exclusively on the robot. The user especially does not need to switch between devices, and always has both hands available to interact with the robot. In the experiments, we explore the viability of such a system.

Robot programming is a well-known field of research, surveys of which can be found in [4,13,20]. In programming by demonstration (PbD), the user guides the robot through tasks. Overviews can be found in [2,5]. Kinesthetic programming as employed here (guiding the robot directly, in a real environment, while it compensates for its weight) has been used in numerous works, e.g. [1,12,15,17]. Finite state automata (FSA) have been used in robotics repeatedly, but mostly at other levels of abstraction. Some applications are [6, 10, 14]. In all of these, states represent more abstract properties of the system, while here, they are associated directly with robot configurations. An exception, [18] can also employ states like we do. But their automaton as a whole is generated in an automatic fashion rather than explicitly by the user. Generalizing demonstrated behaviour is typical for PbD, found e.g. in [8, 21]. This is applied to FSA in [11, 23] in the strongest extent, where the automata are evolved via genetic algorithms.
Most works about simplified robot programming use extensive visual user interfaces, as [7, 16, 26, 28]. This includes approaches commercially available like the Franka Emika Desk interface [9], the Artiminds Robot Programming Suite [3], the TechMan software TMflow [27], Universal Robots' PolyScope [29], or the Rethink Robotics Intera software for their Baxter or Sawyer robots [22]. An approach close to ours in terms of program structuring is [24], with basic loops and branches. But there, too, a visual representation of the program (in the vein of Gantt charts) is essential for the concept.
To summarize, robot programming (by demonstration or using FSA) is and has been an active field of research. But to the best of our knowledge, this approach of explicitly generating structured, sensor-based programs (represented as automata) without a visual interface has not been explored so far.

## 2 Programming Concept

Our model is a variation of FSA that we refer to as *Robot State Automata (RSA)*.

**Definition 1** *An RSA is given as a tuple* $M = (Q, \Sigma, C, \delta, \varphi, q_s, E)$ *where*

- $Q$ *is the finite set of states,*
- $\Sigma$ *is the finite input alphabet,*
- $C$ *is the space of robot joint configurations,*
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \to Q$ *is the state transition function,*
- $\varphi : Q \to C$ *is the function mapping states to robot joint configurations,*
- $q_s \in Q$ *is the initial state, and*
- $E \subseteq Q$ *is the set of terminal states.*

A *(robot joint) configuration* $c \in C$ is a tuple of positions of all joints of the robot, also including gripper opening. States $q \in Q$ are mapped to configurations by $\varphi$. The input alphabet $\Sigma$ is the set of *stimuli* the robot perceives via its sensors. This can be as simple as a color space (e.g. represented as $[0, 255]^3$), when a single stimulus is the average color of a camera image. But it is also possible to use entire images (in which case $\Sigma$ is the set of $n \times m$ matrices of color values for image dimensions $n \times m$) and use an image-based similarity measure to compare stimuli. Another option is extracting information about objects in the image, with stimuli being vectors of properties. Note that $\Sigma$ will, generally, be a large set, and that it is neither practical nor necessary to ever list it explicitly.

The partial function $\delta$ determines the possible state transitions, i.e. if the RSA is in $q \in Q$ and the robot perceives a stimulus $s \in \Sigma$, it will move to $\delta(q, s)$ if defined. By $\varepsilon \notin \Sigma$ we denote, as in FSA, an empty input sequence. We need this for transitions where the robot does not read a stimulus and the RSA directly moves from $q$ to $\delta(q, \varepsilon)$, called *spontaneous transitions*. We require RSA to be deterministic, i.e., if $\delta(q, \varepsilon)$ is defined, then $\delta(q, s)$ must be undefined $\forall s \in \Sigma$. As a consequence, $Q$ can be partitioned as $Q_\varepsilon \uplus Q_b$ where $Q_b = \{q \in Q \mid \exists s \in \Sigma : \delta(q, s) \text{ defined}\}$ are *branching states* and $Q_\varepsilon = Q \setminus Q_b$ are states with spontaneous transitions. In branching states, the automaton can branch into several possible transitions. Fig. 1 provides a visualization of branching in a robot program. Spontaneous transitions are, by definition, always uniquely determined.

In execution, the RSA starts in $q_s$, the robot in $\varphi(q_s)$. In each step, the current state $q$ is either in $Q_\varepsilon$ or in $Q_b$. If $q \in Q_\varepsilon$, the RSA changes state to $q' = \delta(q, \varepsilon)$ and the robot moves from $\varphi(q)$ to $\varphi(q')$. If $q \in Q_b$, the RSA makes the robot take a stimulus $s \in \Sigma$. If $q' = \delta(q, s)$ is defined, the RSA changes state to $q'$ and the robot moves from $\varphi(q)$ to $\varphi(q')$. Otherwise, the RSA remains in $q$ and makes the robot take another stimulus $s$. This keeps repeating until a stimulus $s$ with a defined transition is perceived. Execution stops when the new state is in $E$.

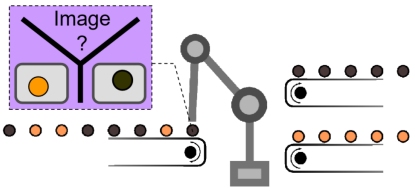


Fig. 1: Example: Parts arriving on the left are sorted onto different conveyor belts. Sensors measure the color of objects (different $s_i \in \Sigma$) to select a branch in the branching state.

We now explain how an RSA is created in our approach. User interaction consists of guiding the robot and using three *command inputs*. Note that command inputs (triggering a command, e.g. by pushing a button) and recorded stimuli (elements $s \in \Sigma$, e.g. observed color values) are different. Command inputs are used to program the robot, as explained below. We start with $Q = \{q_s\}$, no transitions, and $\varphi(q_s)$ as configuration. Then, incrementally, new states are generated. In the default interaction, the user (physically) guides the robot, while the system records its configuration in fixed time

intervals $\Delta$. So, with the RSA in $q$ at time $t$, the system automatically adds a state $q'$ to $Q$ at $t + \Delta$, defines $\delta(q, \varepsilon) = q'$, and sets $\varphi(q') = c$, where $c$ is the configuration at $t + \Delta$.
The first command input is for *branching*. When triggered in some state $q$, this makes the robot record a stimulus $s$. When the user then continues to guide the robot, the next transition is set as $\delta(q, s) = q'$, where $q'$ is a newly generated state. Note that this is not an $\varepsilon$-transition and will require the same stimulus $s$ to be executed. After this transition, the system goes back to recording the guided trajectory.
The second command input is for *recurrence transitions* from a current state $q$ to an already existing state $q'$. For this, the user moves the robot back to the configuration $\varphi(q')$ and triggers the command input. This generates a new $\varepsilon$-transition $\delta(q, \varepsilon) = q'$. Then, the robot starts executing spontaneous transitions from $q'$ until reaching a branching state, where the user can take control again to add a new branch. To introduce a new branch in the branching state, the first command input is used again, just as above.[3] To make recurrence transitions easier to use, it can be decided to only allow branching states as targets (greatly reducing the number of potential targets to keep in mind).
The third command input is used to define terminal states. Triggering it while in state $q$ adds $q$ to the set of terminal states $E$.
These command inputs and guiding the robot are the only interactions necessary. No display is used, and mapping the command inputs to physical buttons installed on the robot itself allows the user to only interact with the manipulator.

So the user generates the robot program (i.e. the RSA) step by step in *programming mode*. In *execution mode*, this program controls the robot. These two modes are interwoven. When reaching a terminal or a previously known state in programming mode, the system switches to execution mode. Then, the automaton is executed from the initial or current state, respectively. When execution reaches a branching state, the robot executes known branches for corresponding stimuli, or the user can expand the program with a new transition. If they record a new transition, the system switches back to programming mode. This interwoven process of programming and execution has the benefit that the automaton can be expanded on demand, even after executing it a number of times.

## 3 Expressiveness of the Approach

Different approaches to robot programming can express programs of different complexity. For our approach, this is the class of FSA. By construction, generated programs are RSA. We can reduce a given RSA to an FSA by removing the excess components ($C$ and $\varphi$). By these steps, generated programs can be mapped to FSA. We argue the reverse to be true as well: any given FSA can be (mapped to an RSA and then) generated in the proposed approach.

[3] Note that looping back is not the only way to reach a branching state for adding new branches. The system can also be programmed up to a terminal state, then executed from the beginning, until reaching the branching state again, as detailed below.

First, if we cut away all unreachable states of an FSA, the result will be functionally equivalent to the original, so we use such a reduced FSA as basis. We then need to specify a mapping from FSA to RSA. The components $Q, \Sigma, \delta, q_s$ and $E$ can be adopted directly. We demand $\varphi(q_s)$ to be a known, fixed start position, and $\varphi(q_e) = \varphi(q_s)\ \forall q_e \in E$. Configurations $\varphi(q)$ for all other $q \in Q$ can be chosen freely (assuming they do not generate collisions with the environment and targets for recurrence transitions have different configurations). Transitions correspond to movements of the robot (possibly degenerate ones from and to the same position), which also need to be collision-free. With some convex region of free space including $\varphi(q_s)$, we can guarantee this e.g. by spacing evenly over that region all states with more than one incoming transition and one additional position for all other non-start and non-end states. This mapping leads to an RSA that satisfies the conditions: All linking targets can be distinguished from each other, and all configurations and movements are in the free space.
It remains to show that any legal RSA can be programmed. We can trivially generate branches and recurrence transitions, and a state that is visited for the first time is generated automatically. So to generate a given RSA, we can choose a sequence of runs of that RSA (each a sequence of states with legal transitions between them) with the conditions that each run must cover at least one state or transition not covered before, and over all runs we must cover all states and transitions in the (reduced) automaton. Since the number of states and transitions is finite, we can do this in a finite number of runs.
Together, this shows that arbitrary FSA can be both represented as RSA and, in the proposed approach, generated in terms of program structure.

## 4 Experimental Evaluation

The prototype setup used in the experiments consisted of a Kuka LWR IV, with a Robotiq three-finger gripper and an IDS uEye color camera as eye-in-hand, as depicted in Fig. 2. Command inputs were given via buttons on a keyboard. For the prototype, recording a stimulus consisted of taking the average color value over a central area of pixels in the camera image.

Experiments were structured as follows: First, an introductory briefing of approximately five minutes was held to detail the programming system. Then, participants were asked to program three small pick-and-place tasks with colored wooden cubes aimed to use the features of the approach, sketched in Fig. 3. In the first task, a branching state with two branches was programmed, testing the color of a block. Green and red blocks were placed in respective target areas. In the second task, a second layer of branches was programmed. Blocks were placed in their target area only if a mat of the same color was present. In the third task, recurrence transitions were used. Green or red blocks were picked up and placed on a blue block, when the latter was present. Blue blocks detected were placed as basis. The states after picking up red and green blocks were linked to program the rest of those branches only once. An automaton for the first task is depicted in Fig. 4, omitting most non-branching states.

Fig. 2: The robot setup as used in the experiments.

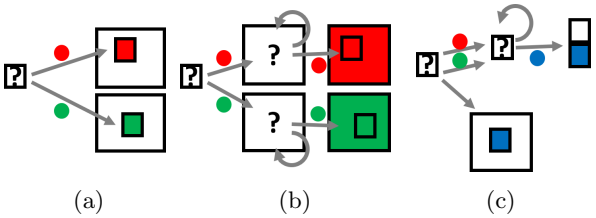


Fig. 3: Task sketch used in the experiments for explanation. First a simple sorting task, using a branching state (a). The second requires a second layer of branching to check the presence of the correct mat (b). The third makes use of recurrence transitions, uniting the branches for green and red (c).

These were conducted with five non-expert participants. Programming times are given in Fig. 5. Overall, the concept was deemed comprehensible by the participants and 12 out of 15 tasks were successfully programmed. The main problem in failed attempts was the camera implementation (small misplacements sometimes led to a wrong color value). The robot also deviated from its position in some configurations (due to imperfect calibration). Apart from these issues, which are orthogonal to the programming concept, participants were found to cope well with this concept of RSA programming, despite having no prior knowledge of finite state automata.
Of the twelve successful tasks, ten were programmed in less than five minutes. Recalling that the study involved non-expert users with a briefing of roughly five minutes, we feel this demonstrates that the approach can be adopted quickly and efficiently, despite not using a visual interface.

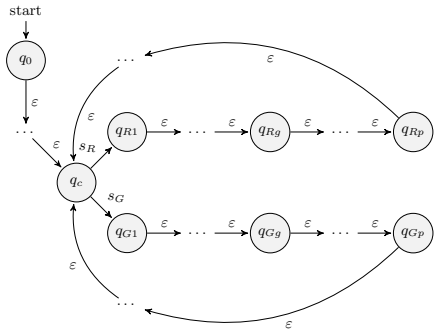


Fig. 4: Automaton for the first task. States go from $q_0$ to branching state $q_c$, to grasp states $q_{\alpha g}$ for picking up blocks ($\alpha \in \{R, G\}$), to placement states $q_{\alpha p}$, then return to $q_c$.

| Participant | Task 1 | Task 2 | Task 3 |
|---|---|---|---|
| 1 | 131s | 160s | |
| 2 | 134s | 290s | 285s |
| 3 | 106s | 253s | 413s |
| 4 | 112s | 234s | |
| 5 | 127s | | 368s |
| average | 122s | 234s | 355s |

Fig. 5: Programming times. Missing values indicate failed tasks.

## 5 Conclusion

We have outlined an approach to intuitive programming via robot state automata and presented the results of experiments on the implemented prototype. The approach aims to enable non-expert users to easily generate sensor-based structures, without the feedback of a visual interface. The experiments confirm that this is indeed possible: Users completely new to robotics were able to program the example pick-and-place tasks quickly and with only a short introduction to the system. More complex tasks and programs are expressible in the system and its underlying formalism. Whether it is also practically feasible to program such tasks in this manner will require further study.
In future work, the automata model can be expanded, allowing for tasks other than pick-and-place with absolute positioning. Here, states correspond to robot configurations, but they could be generalized e.g. to positions relative to perceived objects. This could provide solutions for problems such as placing blocks next to each other, for an variable number of blocks. Different models for perceiving and recording stimuli could be tried and evaluated, especially such that extract more high-level information about the objects in view. Motion planners could be employed to cope with dynamically placed objects. Furthermore, on an interface level, the problems of dealing with errors in programming and of debugging robot programs should be considered. Finally, to compare directly to other approaches, user studies with a larger number of participants should be conducted, with the same tasks being programmed on different systems.

## References

1. Akgun, Baris, et al. “Trajectories and Keyframes for Kinesthetic Teaching.” Proceedings of the Seventh Annual ACM/IEEE International Conference on Human-Robot Interaction, IEEE, 2012, pp. 391–98.
2. Argall, Brenna D., et al. “A survey of robot learning from demonstration.” Robotics and autonomous systems 57.5 (2009): 469-483.
3. ArtiMinds Robotics GmbH. Artiminds Robot Programming Suite. `https://www.artiminds.com/artiminds-rps/`, last visit: July 5, 2019.
4. Biggs, Geoffrey, and Bruce MacDonald. “A survey of robot programming systems.” Proceedings of the Australasian conference on robotics and automation. 2003.
5. Billard, Aude, et al. ”Survey: Robot programming by demonstration.” Handbook of Robotics, Chapter 59. 2008.
6. Brooks, Rodney A. “A Robust Layered Control System For A Mobile Robot.” IEEE Journal of Robotics and Automation, vol. RA-2, no. 1, 1986, pp. 14–23.
7. Brunner, Sebastian G., et al. “RAFCON: A Graphical Tool for Engineering Complex, Robotic Tasks.” IEEE International Conference on Intelligent Robots and Systems, 2016, pp. 3283–90.
8. Calinon, Sylvain, et al. “A Task-Parameterized Probabilistic Model with Minimal Intervention Control.” Proceedings - IEEE International Conference on Robotics and Automation, 2014, pp. 3339–44.
9. Franka Emika GmbH. Franka Emika Panda Capabilites. `https://www.franka.de/capability`, last visit: July 5, 2019.

10. Grollman, Daniel H., and Odest Chadwicke Jenkins. “Can we learn finite state machine robot controllers from interactive demonstration?.” From Motor Learning to Interaction Learning in Robots 264 (2010): 407-430.
11. König, Lukas, Sanaz Mostaghim, and Hartmut Schmeck. “Decentralized evolution of robotic behavior using finite state machines.” International Journal of Intelligent Computing and Cybernetics 2.4 (2009): 695-723.
12. Kormushev, Petar, et al. “Imitation Learning of Positional and Force Skills Demonstrated via Kinesthetic Teaching and Haptic Input.” Advanced Robotics, vol. 25, no. 5, 2011, pp. 581–603.
13. Lozano-Perez, Tomas. “Robot programming.” Proceedings of the IEEE 71.7 (1983): 821-841.
14. Marino, Alessandro, et al. “Behavioral Control for Multi-Robot Perimeter Patrol: A Finite State Automata Approach.” Proceedings - IEEE International Conference on Robotics and Automation, 2009, pp. 831–36.
15. Montebelli, Alberto, et al. “On Handing down Our Tools to Robots: Single-Phase Kinesthetic Teaching for Dynamic in-Contact Tasks.” Proceedings - IEEE International Conference on Robotics and Automation, IEEE, 2015, pp. 5628–34.
16. Nguyen, Hai, et al. “ROS Commander (ROSCo): Behavior Creation for Home Robots.” Proceedings - IEEE International Conference on Robotics and Automation, 2013, pp. 467–74.
17. Orendt, Eric M., et al. “Robot Programming by Non-Experts: Intuitiveness and Robustness of One-Shot Robot Programming.” IEEE International Symposium on Robot and Human Interactive Communication, 2016.
18. Orendt, Eric M., and Dominik Henrich. “Control Flow for Robust One-Shot Robot Programming Using Entity-Based Resources.” 18th International Conference on Advanced Robotics, 2017, pp. 68–74.
19. Ott, C., et al. “A Passivity Based Cartesian Impedance Controller for Flexible Joint Robots Part I: Torque Feedback and Gravity Compensation.” Proceedings of the 2004 IEEE International Conference on Robotics and Automation, 2004, p. 2666–2672 Vol.3.
20. Pan, Zengxi, et al. “Recent progress on programming methods for industrial robots.” Robotics and Computer-Integrated Manufacturing 28.2 (2012): 87-94.
21. Park, Dae Hyung, et al. “Movement Reproduction and Obstacle Avoidance with Dynamic Movement Primitives and Potential Fields.” 8th IEEE-RAS International Conference on Humanoid Robots, 2008, pp. 91–98.
22. Rethink Robotics GmbH. “Intera.” `https://www.rethinkrobotics.com/intera`, last visit: July 9, 2019.
23. Riano, Lorenzo, and T. Martin McGinnity. “Automatically composing and parameterizing skills by evolving finite state automata.” Robotics and Autonomous Systems 60.4 (2012): 639-650.
24. Riedl, Michael, Eric M. Orendt, and Dominik Henrich. “Sensor-Based Loops and Branches for Playback-Programmed Robot Systems.” International Conference on Robotics in Alpe-Adria Danube Region. Springer, Cham, 2017.
25. Schraft, Rolf Dieter, and Christian Meyer. “The need for an intuitive teaching method for small and medium enterprises.” VDI BERICHTE 1956 (2006): 95.
26. Steinmetz, Franz, et al. “RAZER—A HRI for Visual Task-Level Programming and Intuitive Skill Parameterization.” IEEE Robotics and Automation Letters, vol. 3, no. 3, 2018, pp. 1362–69.
27. TechMan Robot Inc. “Software Manual TMflow.” `https://assets.omron.eu/downloads/manual/en/v1/tm_flow_software_manual_installation_manual_en.pdf`, last visit: July 5, 2019.

28. Thomas, Ulrike, et al. “A New Skill Based Robot Programming Language Using UML/P Statecharts.” Proceedings - IEEE International Conference on Robotics and Automation, IEEE, 2013, pp. 461–66.
29. Universal Robots A/S. “Polyscope Manual.” `https://s3-eu-west-1.amazonaws.com/ur-support-site/53076/Software_Manual_en_Global.pdf`, last visit: July 9, 2019.