Structure Synthesis for Extended Robot State Automata [⋆]

Lukas Sauer[0000−0002−7808−0907] and Dominik Henrich

Universität Bayreuth, Universitätsstr. 30, 95447 Bayreuth, Germany
{lukas.sauer, dominik.henrich}@uni-bayreuth.de

Abstract. Opening up untapped potential in smaller enterprises requires methods for robot programming usable by non-experts. In a previous work on an automata-based programming approach without a graphical interface, linking to previous states to close loop structures was identified as challenging for users. In this paper, we propose an approach to generate such structures automatically from small overlap within the programmed instructions. To this end, a marking function for state pairs is calculated, which evaluates whether the structures starting from two states are conflicting or in agreement (and in the latter case, how much). Algorithms for calculating and utilizing this marking function are presented. Our experiments on an example task confirm the effectiveness of the approach.

Keywords: robot programming · automata · automatic synthesis

1 Introduction

While robots have a long and successful history in large scale industry, there is both increasing use and still untapped potential in small and medium-sized enterprises or workshops [1], where only domain experts are readily available, but no expert robot programmers. This necessitates more intuitive forms of robot programming. Automata-based robot programs are promising in this regard, because they can express control flow in a conceptually easy way: in a given situation, specify what happens next. In our research [2,3], we attempt to gauge the viability of an automata-based approach which does not use a graphical editor. This can reduce the amount of hardware required in the workspace and the number of changes in input devices during programming. Potentially, all interactions necessary to program the robot can be done via the robot in this system. The overall approach from our previous work is as follows: In each step, the user

[⋆] This work has partly been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant agreement He2696/15 INTROP
This is a preprint of the following chapter: Sauer, L., Henrich, D., Structure Synthesis for Extended Robot State Automata, published in: Advances in Service and Industrial Robotics RAAD 2022, edited by: Müller, A., Brandstötter, M., 2022, Springer reproduced with permission of Springer Nature Switzerland AG. The final authenticated version is available online at: https://doi.org/10.1007/978-3-031-04870-8_9

specifies the system's next action, which in the automaton representation generates a new state and a corresponding transition. Actions can be e.g. movement to some target robot pose or connecting the current state to another existing state (called *linking*). Actions can either be defined as spontaneous (i.e. in execution, they will trigger automatically) or requiring a specific input or perception.
Closing loops within the program (by linking) without a graphical representation proved to be conceptually challenging in the experiments for our previous work [3]. The user needs a good understanding of how to identify states within the automaton, and where to link, to create specific control flow. The approach proposed in this paper attempts to alleviate this by letting the system detect automatically when parts of the programmed structure start to overlap. When that happens, the system can simplify the structure by itself. The detection of overlap is based on a marking function on state pairs which describes how well the structures starting from these states fit onto each other, or whether there is some contradiction that distinguishes them.
In the following, we give a brief overview over works related to the approach. In Section 3, the marking function is formally defined. Algorithms for generating and utilizing the marking function are presented in Section 4. In Section 5, we describe our experiments. Finally, Section 6 contains our conclusion.

2 Related Work

Automata as a basic concept are used in robotics in varying contexts and abstraction levels [4,5,6,7,8,9]. There is also a number of approaches where users can explicitly generate and edit robot programs in the form of (different variants of) automata [10,11,12]. However, all of these use graphical editors. Our research, in contrast, is trying to gauge the validity of an approach without a GUI, as motivated in Section 1.
Because the formalism of *Extended Robot State Automata* (ERSA) is unique to our research, there is no previous work on automatic simplification of structures within them. Some existing approaches for structural synthesis in a more general context use Version Spaces [13,14,15], for *Programming by Demonstration* (PbD) with applications from text editing macros to robot programs. Other works use planning systems for this goal [16,17]. There are also classical PbD approaches that, if they synthesize structure, ususally do so from repeated demonstrations [8,18,19]. In contrast, the proposed approach operates directly on the automata, and requires only a small amount of overlap with an existing part of a program to close a loop (as opposed to multiple complete demonstrations). Finally, there is a number of works on structured robot programs (including loops) that employ visual programming languages, which we explicitly want to avoid for our system [20,21,22,23].

3 Marking formalism

The goal of our method is to simplify programs when they start *overlapping* (i.e. when the user starts to program a part of the task that has been programmed before) and generate more complex structure (e.g. closed loops) within the program from that. To do this, we track whether state pairs are *conflicting*, meaning there is some information that lets us rule out that two states would represent a shared state in a more concise representation of the task. The opposite term is *unifiable*, i.e. states that could possibly represent a shared logical task state and to which unification might be applied in the future. For unifiable state pairs, the system also calculates how much overlap the two states provide in the structure of their successors. This idea of marking state pairs according to unifiability is directly inspired by unification of states in finite state automata, used for automata minimization (see e.g. [24,25]).
In [3] we defined the model of ERSA, a variant of finite state automata. An ERSA is given as a tuple $M = (Q, \Sigma, P, D, \delta, u, q_s)$, where (with the set of states Q, input alphabet Σ, transition function δ, and initial state q_s as usual) the additional components are the space of poses P, the space of pose variables $D = P^n$, and the update function u. This u is the general update function, which represents the action executed by the robot when the system progresses along a transition as defined by $\delta(q, \sigma) = q'$. We defined the restricted update functions $u^{q,\sigma}$ (to a specific q and σ) as satisfying one of a small number of possible forms. This formulation ensures that the semantics of a logical state of the system do not differ e.g. when the robot is at two different poses in its workspace. The definition, for completeness, is $u^{q,\sigma} : P \times D \times P \mapsto P \times D$, where

$$u^{q,\sigma} \in \left\{ \begin{array}{l} (p, d, p') \mapsto (\bar{p}, d), \\ (p, d, p') \mapsto (p \cdot \bar{p}, d), \\ (p, d, p') \mapsto (p' \cdot \bar{p}, d), \\ (p, d, p') \mapsto (p, d_1, \dots, d_{k-1}, p, d_{k+1}, \dots, d_n), \\ (p, d, p') \mapsto (d_k, d) \end{array} \right\}, 1 \leq k \leq n, \bar{p} \in P \qquad (1)$$

For more details on ERSA, we refer to [3].
In marking state pairs, we need to consider the updates along pairs of outgoing transitions from the two states. If the successor states (the targets of the two transitions) are not conflicting, but a different update is performed (e.g. movement in one case and saving the current pose, or a different movement, in the other), the preceding states themselves cannot be considered unifiable. Substituting one by the other would lead to a different action being performed.
Formally: There is a function

$$M_{trans} : \{\{q, q'\} \mid q \in Q, q' \in Q, q \neq q'\} \times (\Sigma \cup \{\varepsilon\}) \mapsto \{-1, 0\},$$

$$M_{trans}(\{q, q'\}, \sigma) = \begin{cases} -1 \text{ if } u^{q,\sigma} \text{ and } u^{q',\sigma} \text{ are from different cases in (1),} \\ \quad \text{or the parameters } k \text{ or } \bar{p} \text{ are different,} \\ 0 \text{ otherwise} \end{cases}$$

We will use this in the definition of the marking function. A final piece of notation: we can express $Q = Q_b \uplus Q_v \uplus Q_t$ as a disjoint partition, where

- $Q_b = \{q \in Q \mid \exists \sigma \in \Sigma : \delta(q, \sigma) \text{ defined}\}$ are *branching states* (states with at least one transition that requires a specific input)
- $Q_v = \{q \in Q \mid \delta(q, \varepsilon) \text{ defined}\}$ are *via states* (states with a spontaneous transition)
- $Q_t = Q \setminus \{Q_b \cup Q_v\}$ are states for which no transitions are defined so far

In branching states, the automaton can also branch into several possible transitions. A branching state and a via state are automatically conflicting, because unifying them would create a state with non-deterministic behaviour (following the spontaneous transition or gathering an input).

The *marking function* for state pairs $\{q, q'\}$ is then defined as follows:

$$M : \{\{q, q'\} \mid q \in Q, q' \in Q, q \neq q'\} \mapsto \mathbb{N} \cup \{-1\},$$

$$M(\{q,q'\}) = \begin{cases} -1 & \text{if } ((\text{w.l.o.g. } q \in Q_b \wedge q' \in Q_v) \\ & \quad \vee(\exists \sigma \in \Sigma \cup \{\varepsilon\} : M(\{\delta(q,\sigma), \delta(q',\sigma)\}) = -1) \\ & \quad \vee(\exists \sigma \in \Sigma \cup \{\varepsilon\} : M_{trans}(\{q,q'\},\sigma) = -1)), \\ \multicolumn{2}{l}{\min\left(\max\limits_{\sigma \in \Sigma \cup \{\varepsilon\}} \{M(\{\delta(q,\sigma), \delta(q',\sigma)\}) + 1\}, |Q|\right) \text{ otherwise}} \end{cases}$$

I.e., $M(\{q, q'\}) = -1$ iff the two states are of conflicting type, or there is a conflicting successor pair under some input σ, or the outgoing transitions under some input σ are conflicting. Otherwise, it is one more than the highest marking of a successor pair (under any input σ), defaulting to zero. We will call a non-conflicting marking of value $n \in \mathbb{N}$ *supporting* (to depth n).

4 Algorithms

The table-filling algorithm used to determine the correct markings is presented in pseudocode in Fig. 1. Note that calculating values of M_{trans} does not require any iteration or recursion, only simple lookups within the automaton.

If values of M higher than a threshold of m are calculated for any pair of states, this pair is subjected to *unification*, which denotes merging the two states into a single state. The unification procedure is presented in Fig. 2. It makes use of a disjoint-set (or union-find) data structure [26,27] for tracking the central information: which sets of states will be unified into which *representative*.

We first note the pairs with sufficient support, and set one of both states as the representative. Then, we unify states with their representatives, i.e. we note their successor pairs for unification. This may entail further unification in transitive successors. When there are no more updates, we set all transition targets as well as the initial and current state to their correct representatives. Finally, we remove any states no longer reachable from either the initial or the current state; this can be done by simple graph reachability tests.

```
Data: Table $M$ for $\{\{q,q'\} \mid q \in Q, q' \in Q, q \neq q'\}$ (e.g. lower diagonal matrix)
if $M$ was newly generated then
    for $\{\{q,q'\} \mid q \in Q, q' \in Q, q \neq q'\}$ do
        if w.l.o.g. $q \in Q_b \wedge q' \in Q_v$ then
            $M(\{q,q'\}) \leftarrow -1$
        else
            $M(\{q,q'\}) \leftarrow 0$
else if a new state $q^*$ was just generated then
    for $\{q \in Q \mid q \neq q^*\}$ do
        $M(\{q,q^*\}) \leftarrow 0$
repeat
    for $\{\{q,q'\} \mid q \in Q, q' \in Q, q \neq q'\}$ do
        if $\exists \sigma \in \Sigma \cup \{\varepsilon\} : M(\{\delta(q,\sigma), \delta(q',\sigma)\}) = -1 \vee M_{trans}(\{q,q'\},\sigma) = -1$ then
            $M(\{q,q'\}) \leftarrow -1$
        else
            $M(\{q,q'\}) \leftarrow \min(\max_{\sigma \in \Sigma \cup \{\varepsilon\}} \{M(\{\delta(q,\sigma), \delta(q',\sigma)\}) + 1\}, |Q|)$
until no entry of $M$ changed in this iteration;
```

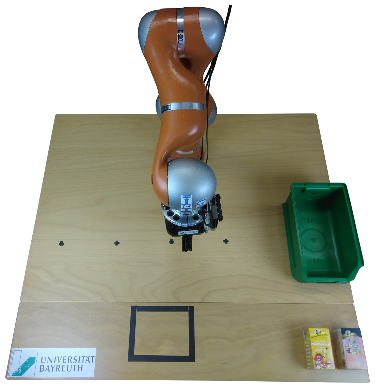
Fig. 1: Table-filling algorithm to generate correct markings

```
Data: Disjoint-set data structure $I$ (e.g. disjoint-set forest) for states
for $\{\{q,q'\} \mid q \in Q, q' \in Q, q \neq q', M(\{q,q'\}) \geq m\}$ do
    $I.union(q,q')$
repeat
    for $\{q \in Q \mid I.find(q) \neq q\}$ do
        for $\{\sigma \in \Sigma \cup \{\varepsilon\}\}$ do
            $I.union(\delta(q,\sigma), \delta(I.find(q),\sigma))$
until no previously distinct state sets were unified in this iteration;
for $\{q \in Q\}$ do
    for $\{\sigma \in \Sigma \cup \{\varepsilon\}\}$ do
        $\delta(q,\sigma) \leftarrow I.find(\delta(q,\sigma))$
Set initial and current state to their representatives; Remove states no longer reachable
  from initial or current state
```

Fig. 2: Unification procedure when sufficient support has been detected.

5 Experiments

The above methods for calculating markings and unifying state pairs were implemented into the system described in our previous work [3]. In this system, the user kinesthetically guides a lightweight robot arm with a mounted gripper and camera. New states are generated by different haptic interactions (specific movements the user executes with the robot). Each new state is connected to the automaton by a new transition. The update along this transition represents the programmed action (e.g., movement to a target pose to which the robot was guided). The table-filling algorithm for calculating the markings is run after each action. When sufficient support (overlap) is detected, unification takes place.
Experiments consisted of the same task from [3], programmed here without any explicit linking. Briefly summarized, it includes all parts of the ERSA formalism: Distinguishing two kinds of objects in an outer loop with two branches, where in the first branch objects of one kind are picked up and placed again after some movement. In the second branch, objects of the other kind are placed in a row by iterating over target positions in a second small loop. The overall setup for this task is depicted in Fig. 3 (left). Here, unification was applied for closing the outer loop (of identifying an arriving object), as well as the inner loop (of iterating over target positions).
The parameter m was set to 3 for the experiments. This is the lowest appropriate value: The task involves overlap of two states after the inital branching state (moving to a grasp pose above the object, and grasping it, in both branches).
Fig. 4 shows an automaton before and after unification of the first branch into



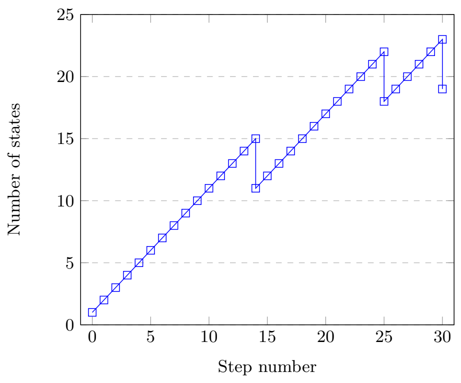
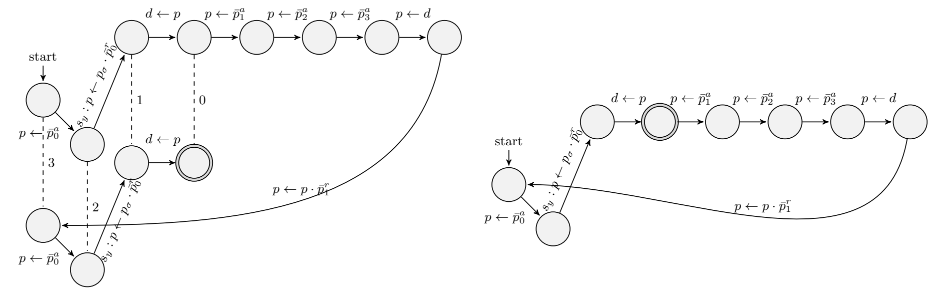


Fig. 3: Left: Setup, reproduced from [3]. Objects arrive in the black rectangle and have to be dipped into the container and put back, or placed at the first free spot marked with a black dot. Right: number of states over number of programmed steps, in one run. Each step adds a new state. On steps 14, 25 and 30, a new state is generated, but unification afterwards reduces the number of states.

a closed loop. It illustrates how only partial overlap of the two instances of the loop body is sufficient for the system to suggest unification.

The approach of marking and unification was verified on the task by applica-


start
$d \leftarrow p$
$p \leftarrow \vec{p}^a_1$
$p \leftarrow \vec{p}^a_2$
$p \leftarrow \vec{p}^a_3$
$p \leftarrow d$
$p \leftarrow \vec{p}^a_0$
$s_y : p \leftarrow p_e \cdot \vec{p}^r_0$
$p \leftarrow p \cdot \vec{p}^r_1$
0
1
2
3

(a) Before unification. Support depth is printed on the dashed edges.

(b) After unification.

Fig. 4: Partial automaton before and after the first unification. The leftmost state pair has a support depth of 3. From there, successor states are unified. Current state after unification is the unified version of the previously current state. (Current states marked by double boundary.)

tion by one of the authors as well as two non-expert participants. These followed the same procedure as the study depicted in [3]. In all three cases, unification was applied successfully. As Fig. 3 (right) shows for a single run, the number of states progresses linearly up to detection of sufficient support, at which point the number of states drops by unification.

6 Conclusion

In this paper we proposed an approach to automatically close structures within ERSA from overlap in the programmed state chains. We presented the formalism of the underlying marking function on state pairs, and the algorithms for calculating and utilizing the marking function. We also outlined our experimental verification. In conclusion, the approach generates the intended structures without any explicit linking operation required from the user. This facilitates the programming of structured tasks by non-experts.

References

1. Perzylo, A., et al: SMErobotics: Smart robots for flexible manufacturing. IEEE Robot. Autom. Mag. 26, 78–90 (2019). https://doi.org/10.1109/MRA.2018.2879747

2. Sauer, L., Henrich, D., Martens, W.: Towards Intuitive Robot Programming Using Finite State Automata. Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). 11793 LNAI, 290–298 (2019). https://doi.org/10.1007/978-3-030-30179-8_25
3. Sauer, L., Henrich, D.: Extended state automata for intuitive robot programming. Mech. Mach. Sci. 102, 61–68 (2021). https://doi.org/10.1007/978-3-030-75259-0_7
4. Brooks, R.A.: A Robust Layered Control System For A Mobile Robot. IEEE J. Robot. Autom. RA-2, 14–23 (1986). https://doi.org/10.1109/JRA.1986.1087032
5. Grollman, D.H., Jenkins, O.C.: Can we learn finite state machine robot controllers from interactive demonstration? Stud. Comput. Intell. 264, 407–430 (2010). https://doi.org/10.1007/978-3-642-05181-4_17
6. König, L., Mostaghim, S., Schmeck, H.: Decentralized Evolution of Robotic Behavior Using Finite State Machines. Int. J. Intell. Comput. Cybern. 2, 1–33 (2009). https://doi.org/10.1108/17563780911005845
7. Marino, A., Parker, L., Antonelli, G., Caccavale, F.: Behavioral control for multi-robot perimeter patrol: A finite state automata approach. Proc. - IEEE Int. Conf. Robot. Autom. 831–836 (2009). https://doi.org/10.1109/ROBOT.2009.5152710
8. Orendt, E.M., Henrich, D.: Control flow for robust one-shot robot programming using entity-based resources. 18th Int. Conf. Adv. Robot. 68–74 (2017). https://doi.org/10.1109/ICAR.2017.8023498
9. Riano, L., Mcginnity, T.M.: Automatically composing and parameterizing skills by evolving finite state automata. Robotics and Autonomous Systems (2012). https://doi.org/10.1016/j.robot.2012.01.002
10. Brunner, S.G., Steinmetz, F., Belder, R., Dömel, A.: RAFCON: A graphical tool for engineering complex, robotic tasks. IEEE Int. Conf. Intell. Robot. Syst. 3283–3290 (2016). https://doi.org/10.1109/IROS.2016.7759506
11. Steinmetz, F., Wollschlager, A., Weitschat, R.: RAZER—A HRI for Visual Task-Level Programming and Intuitive Skill Parameterization. IEEE Robot. Autom. Lett. 3, 1362–1369 (2018). https://doi.org/10.1109/LRA.2018.2798300
12. Thomas, U., Hirzinger, G., Rumpe, B., Schulze, C., Wortmann, A.: A new skill based robot programming language using UML/P Statecharts. Proc. - IEEE Int. Conf. Robot. Autom. 461–466 (2013). https://doi.org/10.1109/ICRA.2013.6630615
13. Lau, T.A., Weld, D.S.: Programming by Demonstration Using Version Space Algebra. Mach. Learn. 53, 111–156 (2003). https://doi.org/10.1023/A:1025671410623
14. Lau, T., Domingos, P., Weld, D.S.: Learning programs from traces using version space algebra. Proc. 2nd Int. Conf. Knowl. Capture. 36–43 (2003). https://doi.org/10.1145/945645.945654
15. Pardowitz, M., Glaser, B., Dillmann, R.: Learning repetitive robot programs from demonstrations using version space algebra. Proc. 13th IASTED Int. Conf. Robot. Appl. RA 2007 Proc. IASTED Int. Conf. Telemat. 394–399 (2007).
16. Hu, Y., Levesque, H.: Planning with loops: Some new results. ICAPS Work. Gen. Plan. 35–42 (2009).
17. Levesque, H.J.: Planning with loops. IJCAI Int. Jt. Conf. Artif. Intell. 509–515 (2005).
18. Eker, S., Lee, T.J., Gervasio, M.: Iteration learning by demonstration. AAAI Spring Symp. - Tech. Rep. SS-09-01, 40–47 (2009).
19. Friedrich, H., Dillmann, R.: Robot programming based on a single demonstration and user intentions. Proc., 3rd Eur. Work. Learn. Robot. (1995).
20. Alexandrova, S., Cakmak, M., Hsiao, K., Takayama, L.: Robot Programming by Demonstration with Interactive Action Visualizations. Robotics: Science and Systems Foundation (2015). https://doi.org/10.15607/rss.2014.x.048

21. Alexandrova, S., Tatlock, Z., Cakmak, M.: RoboFlow: A flow-based visual programming language for mobile manipulation tasks. Proc. IEEE Int. Conf. Robot. Autom. 5537–5544 (2015). https://doi.org/10.1109/ICRA.2015.7139973
22. Huang, J., Lau, T., Cakmak, M.: Design and evaluation of a rapid programming system for service robots. ACM/IEEE Int. Conf. Human-Robot Interact. 295–302 (2016). https://doi.org/10.1109/HRI.2016.7451765
23. Sefidgar, Y.S., Agarwal, P., Cakmak, M.: Situated Tangible Robot Programming. ACM/IEEE Int. Conf. Human-Robot Interact. Part F1271, 473–482 (2017). https://doi.org/10.1145/2909824.3020240
24. Hopcroft, J.E., Motwani, R, Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley Publishing Company (1979).
25. Watson, B.: A taxonomy of finite automata minimization algorithms. Eindhoven University of Technology, Department of Mathematics and Computing Science, Computing Science Section (1993).
26. Hopcroft, J.E., Ullman, J.D.: Set merging algorithms. SIAM J. Comput. 2, 294–303 (1973). https://doi.org/10.1137/0202024
27. Galil, Z., Italiano, G.F.: Data structures and algorithms for disjoint set union problems. ACM Comput. Surv. 23, 319–344 (1991). https://doi.org/10.1145/116873.116878