Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01160-6

SPECIAL SECTION PAPER

Learning minimal automata with recurrent neural networks

Bernhard K. Aichernig[1] · Sandra König[2,3] · Cristinel Mateis[2] · Andrea Pferscher[1,4] · Martin Tappler[1,5]

Received: 17 February 2023 / Revised: 2 December 2023 / Accepted: 26 January 2024

Abstract
In this article, we present a novel approach to learning finite automata with the help of recurrent neural networks. Our goal is not only to train a neural network that predicts the observable behavior of an automaton but also to learn its structure, including the set of states and transitions. In contrast to previous work, we constrain the training with a specific regularization term. We iteratively adapt the architecture to learn the minimal automaton, in the case where the number of states is unknown. We evaluate our approach with standard examples from the automata learning literature, but also include a case study of learning the finite-state models of real Bluetooth Low Energy protocol implementations. The results show that we can find an appropriate architecture to learn the correct minimal automata in all considered cases.

Keywords Automata learning · Machine learning · Recurrent neural networks · Bluetooth Low Energy · Model inference

1 Introduction

Models are at the heart of any engineering discipline. They capture the necessary abstractions to master the complexity in a systematic design and development process. In software engineering, models are used for a variety of tasks, including specification, design, code-generation, verification, and testing. In formal methods, these models are given formal mathematical semantics to reach the highest assurance levels. This is achieved through (automated) *deduction*, i.e., the reasoning about specific properties of a general model.

With the advent of machine learning, there has been a growing interest in the *induction* of models, i.e., the learning of formal models from data. We have seen techniques to learn deterministic and non-deterministic finite state machines, Mealy machines, timed automata, and Markov decision processes. In this research, called automata learning [1], model learning [2], or model inference [3], specific algorithms have been developed that either start from given data (passive learning) [4] or actively query a system during learning (active learning) [5]. Two prominent libraries that implement such algorithms are AALpy [6] (implemented in Python) and LearnLib (implemented in Java) [7].

An alternative to specific algorithms is to map the automata learning problem to another domain. For example, it was shown that the learning problem can be encoded as SAT [8–11] or SMT [12, 13] problem, and then it is the task of the respective solver to find a model out of the given data.

In this work, we explore the question of whether machine learning can be harnessed for automata learning. That is, we research if and how the problem of automata learning can be mapped to a machine learning architecture. Our results show that a specific recurrent neural network (RNN) architecture is able to learn a Mealy machine from given data. Specifically, we approach the classic NP-complete problem of inducing an automaton with at most k states that is consistent with a finite

Communicated by Holger Schlingloff and Ming Chai.

✉ Andrea Pferscher
andreapf@ifi.uio.no

Bernhard K. Aichernig
aichernig@ist.tugraz.at

Sandra König
sandra.koenig@researchdrivensolutions.ie

Cristinel Mateis
cristinel.mateis@ait.ac.at

Martin Tappler
martin.tappler@ist.tugraz.at

1 Institute of Software Technology, Graz University of Technology, Graz, Austria

2 AIT Austrian Institute of Technology, Vienna, Austria

3 Research Driven Solutions Ltd., Greystones, Ireland

4 Department of Informatics, University of Oslo, Oslo, Norway

5 TU Graz - SAL DES Lab, Silicon Austria Labs, Graz, Austria

Published online: 21 March 2024

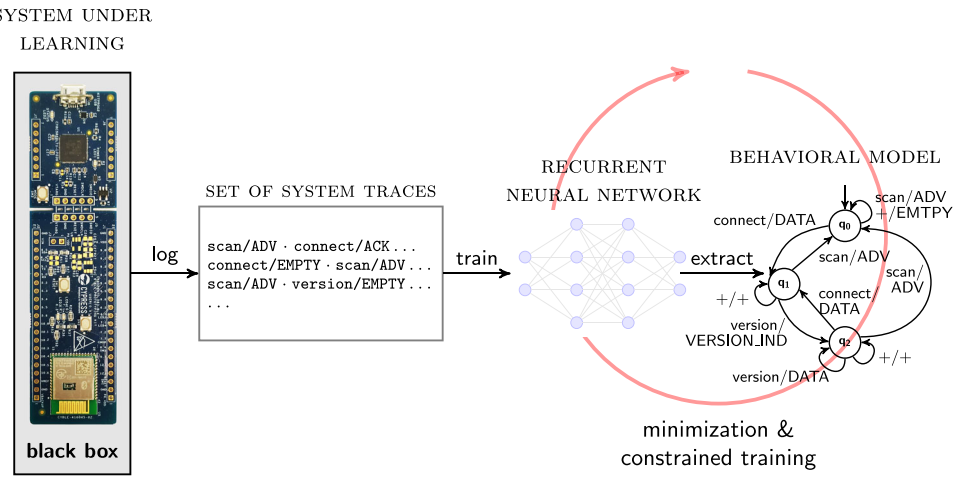

Fig. 1 Automata learning framework that is based on training a recurrent neural network (RNN) using a given set of traces. To learn a minimal automaton, we adapt the structure of the RNN iteratively

sample of a regular language [14]. Figure 1 depicts the basic procedure of the presented RNN-based learning technique. Given a set of traces from a black-box system, we train an RNN from which we extract an automaton that models the behavior of the system.

The main contributions of this work can be summarized as follows: (i) a novel architecture for automata learning by enhancing classical RNNs, (ii) a specific constrained training approach exploiting regularization, (iii) a systematic evaluation with standard grammatical inference problems and a real-world case study, and (iv) evidence that we can find an appropriate architecture to learn the correct automata in all considered cases.

This is an extended version of a conference paper presented at SEFM 2022, the 20th International Conference on Software Engineering and Formal Methods [15]. The new contributions comprise (i) a generalized algorithm for which the number of states of the automaton needs not be known, but can construct an automaton with minimal states, and (ii) the corresponding new evaluation.

This new generalization of our learning technique is non-trivial. Figure 1 illustrates the iterative procedure of our extension. The main idea is as follows: First, we determine an upper-bound of states that are necessary to capture the data in a Mealy machine. For this, we build a tree out of the given data traces and count the nodes. Then, we initialize an RNN with the size proportional to this upper bound. During training, we search for an automaton representation that captures the data. Once an automaton is found, we check with a standard minimization algorithm if a smaller automaton exists. If the minimized automaton has fewer states than previously found automata, we train an RNN with this smaller target number of states. This is done until we find a minimal automaton that is consistent with the training data. To sum up, we start with a tree-shaped model without any generalization and end with a minimal automaton. Our evaluation demonstrates that this is indeed possible.

The rest of the paper is structured as follows. Section 2 introduces preliminary work. In Sect. 3, we present our automata learning technique based on RNNs. Section 4 discusses the results of the conducted case studies. We compare to related work in Sect. 5, followed by concluding remarks in Sect. 6.

2 Preliminaries

2.1 Recurrent neural networks

Recurrent neural networks (RNNs) are a popular choice for modeling sequential data, such as time-series data [16]. The classical version of an RNN with feedback from a hidden layer to itself is known as vanilla RNN [17].

A *vanilla recurrent neural network* with input x and output y is defined as

$$h^{<t>} = f\left(W_{hx}x^{<t>} + W_{hh}h^{<t-1>} + b_h\right)$$
$$\hat{y}^{<t>} = g\left(W_y h^{<t>} + b_y\right)$$

where f and g are activation functions for the recurrent and the output layer, respectively. Popular activation functions for the recurrent layer are rectified linear unit (*ReLU*) and hyperbolic tangent (*tanh*), whereas the *softmax* or *hardmax* functions may be used for g when categorical output values shall be predicted. The activation functions for the output values are defined as

$$\text{softmax}(z)[i] = \frac{e^{z[i]}}{\sum_{n=1}^{N} e^{z[i]}},$$

and

$$\text{hardmax}(z)[i] = \begin{cases} 1, & \text{if } \ z[i] = max(z) \\ 0 & \text{otherwise} \end{cases}$$

for $i \in 1 \ldots N$ and $z = (z[1], \ldots, z[N]) \in \mathbb{R}^N$, where softmax$(z)$ provides a probability distribution over the values of vector z and hardmax(z) assigns the probability of one to the index in z with the highest value. The parameters, aka weights, $\Theta = (W_{hx}, W_{hh}, b_h, W_y, b_y)$ need to be learned. The input to the network at time step t is $x^{<t>}$, whereas $\hat{y}^{<t>}$ is the corresponding network's prediction. $h^{<t>}$ is referred to as the hidden state of the network and is used by the network to access information from past time steps or equivalently, pass relevant information from the current time step to future steps.

An RNN maps an input sequence $\mathbf{x}$ to an output sequence $\hat{\mathbf{y}}$ of the same length. It is trained based on training data $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$ containing m sequence pairs. While processing input sequences $\mathbf{x}_i = (x_i^{<1>}, \ldots, x_i^{<n>})$, values of the parameters Θ are learned to minimize the error between the true outputs $\mathbf{y}_i = (y_i^{<1>}, \ldots, y_i^{<n>})$ and the network's predictions $(\hat{y}_i^{<1>}, \ldots, \hat{y}_i^{<n>})$.

The error is measured through a predefined loss function. The most popular loss functions are the mean squared error for real-valued $y^{<t>}$, and the cross-entropy loss for categorical $y^{<t>}$.

Gradient-based methods are used to minimize the error by iteratively changing each weight in proportion to the derivative of the actual error with respect to that weight until the error falls below a predefined threshold for a fixed number of iterations.

2.2 Finite state machines

We consider finite-state machines (FSMs) in the form of Mealy machines:

Definition 1 A **Mealy machine** is a 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ where

- Q is a finite set of states containing the initial state q_0,
- I and O are finite sets of input and output symbols,
- $\delta : Q \times I \to Q$ is the state transition function, and
- $\lambda : Q \times I \to O$ is the output function.

Starting from a fixed initial state, a Mealy machine $\mathcal{M}$ responds to inputs $i \in I$, by changing its state according to δ and producing outputs $o \in O$ according to λ. Given a sequence of inputs $\mathbf{i} \in I^*$, $\mathcal{M}$ produces an output sequence $\mathbf{o} = \lambda^*(q_0, \mathbf{i})$, where $\lambda^*(q, \epsilon) = \epsilon$ for the empty sequence ϵ and $\lambda^*(q, i \cdot \mathbf{i}) = \lambda(q, i) \cdot \lambda^*(\delta(q, i), \mathbf{i})$, i is an input, $\mathbf{i}$ is an input sequence, and $\cdot$ denotes concatenation. Given input and output sequences $\mathbf{i}$ and $\mathbf{o}$ of the same length, we use $t(\mathbf{i}, \mathbf{o})$ to create a sequence of input–output pairs in $(I \times O)^*$. We call such a sequence of pairs a *trace*.

A Mealy machine $\mathcal{M}$ defines a regular language over $I \times O$: $L(\mathcal{M}) = \{t(\mathbf{i}, \mathbf{o}) \mid \mathbf{i} \in I^*, \mathbf{o} = \lambda^*(q_0, \mathbf{i})\} \subseteq (I \times O)^*$. The language contains the deterministic response to any input sequence and excludes all other sequences. We can now formalize the problem that we tackle in this paper: *Given a finite set of traces $S \subset (I \times O)^*$, we learn a Mealy machine $\mathcal{M}$ with at most n states such that $S \subseteq L(\mathcal{M})$, by training an RNN.* This is a classic NP-complete problem in grammatical inference [14]. Usually, it is stated for (DFAs), but any DFA can be represented by a Mealy machine with *true* and *false* as outputs, denoting whether a word (input sequence) is accepted.

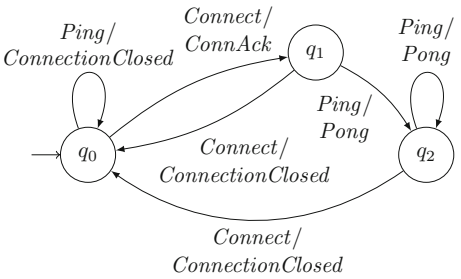

Fig. 2 Mealy machine of a ping server

Example 1 (Model of Ping Server) Figure 2 shows a Mealy machine of a simple ping server that responds to pings after a connection has been established. The model has three states that are connected with transitions labeled by pairs of inputs and outputs. For example, from the initial state q_0, the server responds with *ConnAck* to the input *Connect*. Any further *Connect* input leads to a closing of the connection with the corresponding output observation *ConnectionClosed*.

Next, we introduce auxiliary techniques that are related to automata learning. With the first, we compute a bound on the number of FSM states that are sufficient for a Mealy machine to produce a set of given traces. The second technique, FSM minimization, computes a Mealy machine $\mathcal{M}'$ from a Mealy machine $\mathcal{M}$, such that $\mathcal{M}'$ has the minimal number of states and its language is equivalent to $\mathcal{M}$.

2.2.1 Bounding FSM size

Let $S \subset (I \times O)^*$ be a finite set of traces and let $\ll$ be the reflexive prefix relation on traces. To compute an upper bound on the number of FSM states sufficient to produce S, we create a prefix-tree acceptor (PTA) from S and use its number of nodes as a bound for the number of FSM states that are sufficient to produce S. PTA creation is a common preprocessing step in automata learning algorithm [18], like RPNI [4]. We create input–output prefix tree acceptors (IOPTAs), which are a variation of PTAs similar to the PTAs used by IOALERGIA [19]. An IOPTA T is a tree that compactly represents a trace sample S with edges labeled by inputs and

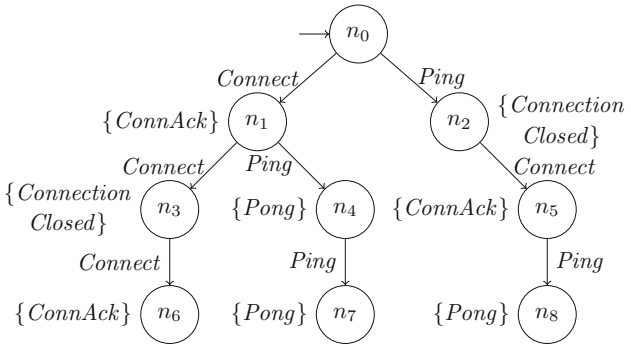

Fig. 3 IOPTA representing traces sampled from the ping server

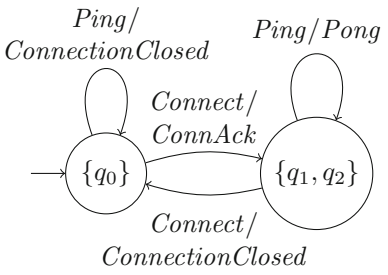

Fig. 4 Minimal Mealy machine of a ping server

nodes, except the root, labeled by outputs. Hence, a path from the root to a node of T is labeled by a trace $(I \times O)^*$.

An IOPTA T created for a trace sample S contains a path labeled by a trace t iff S contains a trace t' with $t \ll t'$, i.e., T contains a path for every trace prefix. Thus, T can be created from S by merging traces with common prefixes. Such an IOPTA T is a partial[1] Mealy machine that defines exactly the prefix-closure of S. We can deduce that *the number of nodes of T is a bound on the number of FSM states* sufficient to represent S. Since languages defined by a Mealy machine are prefix-closed, IOPTA computation does not introduce any generalization.

Example 2 (IOPTA of ping server) Suppose we sampled a set of three traces that includes:

- $Connect/ConnAck \cdot Connect/ConnectionClosed \cdot Connect/ConnAck$
- $Connect/ConnAck \cdot Ping/Pong \cdot Ping/Pong$
- $Ping/ConnectionClosed \cdot Connect/ConnAck \cdot Ping/Pong$

The corresponding IOPTA is shown in Fig. 3, where outputs are put in curly braces to distinguish them from inputs. The IOPTA has nine nodes; thus, we know that nine states are sufficient to model the ping server.

2.2.2 FSM minimization

Minimization of FSMs basically partitions the states Q of a Mealy machine $\mathcal{M}$ into blocks B that are equivalent w.r.t. λ^*; see Hopcroft et al. [20] for minimization of DFAs. That is, two states are grouped into a block if they produce the same outputs, and thus cannot be distinguished. Given such a partition B, a minimal Mealy machine $\mathcal{M}'$ can be constructed with states B, i.e., states given by blocks of indistinguishable states. Transitions between $b \in B$ and $b' \in B$ are created if there is a corresponding transition between $r \in b$ and $s \in b'$ in $\mathcal{M}$. Note that $\mathcal{M}'$ is unique up to a renaming. Active automata learning algorithms, like L^*, have minimality as an inherent property, whereas we apply minimization as an additional step. Efficient algorithms, such as Hopcroft's FSM minimization algorithm [21] have an $n \log n$ worst-case runtime complexity. Hence, the runtime overhead of the minimization step is negligible.

Example 3 (Minimization of Ping Model) The model shown in Fig. 2 is non-minimal. The states q_1 and q_2 are equivalent, as there is no sequence that distinguishes them. Hence, a minimization would create partition $\{\{q_0\}, \{q_1, q_2\}\}$. Based on that, we can create the minimal Mealy machine shown in Fig. 4.

2.3 Automata learning

Automata learning creates behavioral FSMs of black-box systems. Figure 5 illustrates the general framework for learning a reactive system model in the form of a Mealy machine. The goal of automata learning is to create a model $\mathcal{M}$ such that $L(\mathcal{M}) = L(\mathcal{M}_{\text{SUL}})$, where $\mathcal{M}_{\text{SUL}}$ is an unknown Mealy machine representing the System Under Learning (SUL).

We distinguish between active and passive learning algorithms. Passive learning creates a behavioral model from a given set of traces. To learn a Mealy machine $\mathcal{M}_P$, passive learning infers from a finite set of traces $S \subset (I \times O)^*$ a model $\mathcal{M}_P$ such that $S \subseteq L(\mathcal{M}_P)$, often restricting $\mathcal{M}_P$ to have at most k states. Given that $S \subseteq L(\mathcal{M}_{\text{SUL}})$, most algorithms guarantee $L(\mathcal{M}_P) = L(\mathcal{M}_{\text{SUL}})$ for large enough S and finite $\mathcal{M}_{\text{SUL}}$ [18]. One challenge in the application of passive learning is to provide a finite set of traces such that $L(\mathcal{M}_P) = L(\mathcal{M}_{\text{SUL}})$.

Active automata learning queries the SUL to create a behavioral model. Many active learning algorithms are based on the L^* algorithm [5] which is defined for different modeling formalisms like Mealy machines [22]. L^* queries the SUL to generate a finite set of traces $S \subset (I \times O)^*$ from which a hypothesis Mealy machine $\mathcal{M}_A$ is constructed that fulfills $S \subseteq L(\mathcal{M}_A)$. L^* guarantees that the $\mathcal{M}_A$ is minimal. The hypothesis $\mathcal{M}_A$ is then checked for equivalence to the language $L(\mathcal{M}_{\text{SUL}})$. Since $\mathcal{M}_{\text{SUL}}$ is unknown, checking the

[1] The transition and output functions are partial functions.

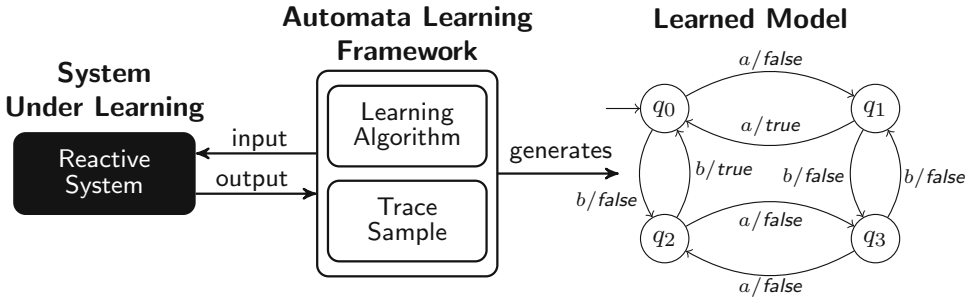

Fig. 5 The automata learning framework generates a Mealy machine from a sample of traces. The sample is generated from the executions of inputs on the reactive system

behavioral equivalence between $\mathcal{M}_{\text{SUL}}$ and $\mathcal{M}_A$ is generally undecidable. Hence, conformance testing is used to substitute the equivalence oracle in active learning. Model-based testing techniques generate a finite set of traces $S_{\mathcal{T}} \subset (I \times O)^*$ from executions on $\mathcal{M}_A$ and check if $S_{\mathcal{T}} \subset L(\mathcal{M}_{\text{SUL}})$. If $t(\mathbf{i}, \mathbf{o}) \notin L(\mathcal{M}_{\text{SUL}})$, a counterexample to the behavioral equivalence between $\mathcal{M}_{\text{SUL}}$ and $\mathcal{M}_A$ is found. Based on this trace, the set of traces $S \subset (I \times O)^*$ is extended by performing further queries. Again a hypothesis $\mathcal{M}_{\text{A}}$ is created and checked for equivalence. This procedure repeats until no counterexample to the equivalence between $L(\mathcal{M}_{\text{SUL}})$ and $L(\mathcal{M}_{\text{A}})$ can be found. The algorithm then returns the learned automaton $\mathcal{M}_{\text{A}}$. Note that L^* creates $\mathcal{M}_{\text{A}}$ such that $S \subset L(\mathcal{M}_{\text{A}})$. With access to a perfect behavioral equivalence check, which provides any differences between the languages defined by $\mathcal{M}_{\text{SUL}}$ and $\mathcal{M}_A$, we could guarantee that the generated finite set of traces S enables learning a model $\mathcal{M}_A$ such that $L(\mathcal{M}_A) = L(\mathcal{M}_{\text{SUL}})$.

3 Automata learning with RNNs

In this section, we first present the problem that we tackle and propose an RNN architecture as a solution. After that, we cover (i) the constrained training of the proposed RNN architecture with our specific regularization term, and (ii) the usage of the trained RNN to extract an appropriate automaton. Finally, we propose an iterative learning algorithm that uses the proposed RNN architecture and automaton extraction to learn a minimal automaton without knowing the minimal number of states.

3.1 Overview and architecture

It is well known that recurrent neural networks (RNNs) can be used to efficiently model time-series data, such as data generated from the interaction with a Mealy machine. Concretely, this can be done by using the machine inputs $x^{<t>}$ as inputs to the RNN and minimizing the difference between the machine's true outputs $y^{<t>}$ and the RNN's predictions $\hat{y}^{<t>}$. In other words, the RNN would predict the language $L(\mathcal{M})$ of a Mealy machine $\mathcal{M}$.

This optimization process can be performed via gradient descent. Even if such a trained RNN can model all interactions with perfect accuracy, one disadvantage compared to the native automaton representation as, e.g., a Mealy machine, is that it is much less interpretable. While each state in a Mealy machine can be identified by a discrete number, the hidden state of the RNN, which is the information passed from one time step to the next one, is a continuous real-valued vector. This vector may be needlessly large and contain mostly redundant information. Thus it would be useful if we could simplify such a trained RNN into a Mealy machine $\mathcal{M}_R$ that produces the language of a Mealy machine $\mathcal{M}$ that we want to learn, i.e., with $L(\mathcal{M}) = L(\mathcal{M}_R)$.

We approach the following problem. Given a sample $S \subset L(\mathcal{M})$ of traces $t(\mathbf{i}_j, \mathbf{o}_j)$ and the number of states k of $\mathcal{M}$, we train an RNN to correctly predict $\mathbf{o}_j$ from $\mathbf{i}_j$. To facilitate interpretation, we want to extract a Mealy machine $\mathcal{M}_R$ from the trained RNN with at most k states, modeling the same language. For $\mathcal{M}_R$, $S \subset L(\mathcal{M}_R)$ shall hold such that for large enough S we have $L(\mathcal{M}) = L(\mathcal{M}_R)$.

For this purpose, we propose an RNN architecture and learning procedure that ensure that the RNN hidden states can be cleanly translated into k discrete automata states. Compared to standard vanilla RNNs, the hidden states are transformed into an estimate of a categorical distribution over the k possible automaton states. This restricts the encoding of information in the hidden states since now all components need to be in the range [0, 1] and sum up to 1. Figure 6 shows our complete RNN cell architecture for a single hidden layer, implementing the following equations.

$$h^{<t>} = af\left(W_{hx}x^{<t>} + W_{hs}s^{<t-1>} + b_h\right), \quad af \in \{\text{ReLU}, \tanh\} \tag{1}$$

$$\hat{y}^{<t>} = \text{softmax}\left(W_y h^{<t>} + b_y\right) \tag{2}$$

$$\hat{s}^{<t>} = \text{softmax}\left(W_s h^{<t>} + b_s\right) \tag{3}$$

$$s^{<t>} = \begin{cases} \text{softmax}\left(W_s h^{<t>} + b_s\right) \\ \quad \ldots \text{if mode} = \texttt{“train”} \\ \text{hardmax}\left(W_s h^{<t>} + b_s\right) \\ \quad \ldots \text{else (i.e., mode} = \texttt{“infer”}) \end{cases} \tag{4}$$

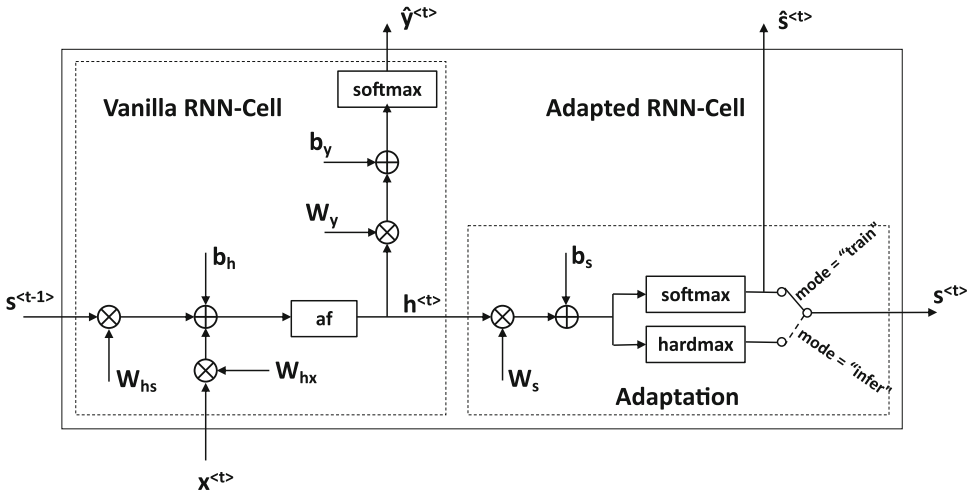

Fig. 6 RNN-cell architecture

In comparison with vanilla RNN cells, the complete hidden state $h^{<t>}$ is only an intermediate vector of values. Based on $h^{<t>}$, an output $\hat{y}^{<t>}$ is predicted using a softmax activation. A Mealy machine state $\hat{s}^{<t>}$ is predicted as well and passed to the next time step. It is computed via (i) softmax during RNN training, and (ii) via hardmax during inference. During training, see Algorithm 2, we also compute the cross-entropy of $\hat{s}^{<t>}$ with $hardmax(\hat{s}^{<t>})$ as a label, which serves as a regularization term. Inference refers to extracting an automaton from the trained RNN, which takes as input the current system state and an input symbol and gives as output the next system state and an output symbol. Hence, we use softmax to estimate a categorical distribution over possible states for training, whereas we use hardmax to concretely infer one state during inference.

Our algorithm for extracting a Mealy machine from a trained RNN, see Algorithm 3, is based on the idea that if the RNN achieves perfect accuracy when predicting the machine's true outputs, the hidden state $h^{<t>}$ encodes information corresponding to the state of a Mealy machine at time step t. Otherwise, the RNN would not be able to predict the expected outputs correctly, since those are a function of both the input and the current state. By adapting the RNN architecture, we enforce hidden states to correspond to discrete Mealy machine states.

3.1.1 Multiple hidden layers

The matrices W_{hx}, W_{hs}, W_y, and W_s and the corresponding bias vectors introduced before define the weights of an RNN with a single layer. It may be beneficial to add additional hidden layers to better predict certain complex behaviors. For the hidden layers, we use fully connected layers, where each layer i is defined by a pair W_{hhi} and b_{hi} containing the weights of that layer. Each layer performs an additional transformation of the hidden state $h^{<t>}$.

Concretely, the one-hot-encoded input $x^{<t>}$ and state $s^{<t-1>}$ are first mapped to $h^{<t>}$, that is, Eq. 1 is left unchanged. Let $h_0^{<t>} = h^{<t>}$, then every additional layer performs the transformation $h_i^{<t>} = af(W_{hhi}h_{i-1}^{<t>} + b_{hi})$. When we have multiple layers, we perform the state and output prediction on the result $h_k^{<t>}$ from the last hidden layer k, that is, we substitute $h^{<t>}$ by $h_k^{<t>}$ in Eqs. 2, 3 and 4. Hence, input processing is carried out only by the first layer, and state and output predictions, along with their corresponding regularization, are performed exclusively in the last layer. All transformations in between are not affected by regularization.

3.2 Training and automaton extraction

In the following, we first discuss how to train an RNN with the structure shown in Fig. 6 such that it will encode an automaton. Secondly, we show how to extract the automaton from a trained RNN. We start by illustrating the basic operation of such an RNN, i.e., the prediction of outputs and state transitions from an input sequence. This is called the *forward pass* and is used during training and automaton extraction.

Forward pass. Algorithm 1 implements the forward pass taking an input sequence $\mathbf{x}$ and a mode variable as parameters. The mode variable distinguishes between training (`train`) and automaton extraction (`infer`). The algorithm returns a pair $(\hat{\mathbf{y}}, \hat{\mathbf{s}})$ comprising the predicted output sequence and the sequence of hidden states visited by the forward pass. We want to learn the language of a Mealy machine, i.e., map $\mathbf{i} \in I^*$ to $\mathbf{o} \in O^*$ for sets I, O of input and output symbols. Therefore, we encode every $i \in I$ using a one-hot-encoding to yield input sequences $\mathbf{x}$ from $\mathbf{i} \in I^*$. In this encoding, every i is associated with a unique $|I|$-dimensional vector, where exactly one element is equal to one and all others are zero. We write x for a one-hot-encoded input i. Analogously, we encode outputs and the hidden state shall approach a one-hot-encoding in a k-dimensional vector space. For one-hot-

encoded outputs, we generally use the letter y and we use D to denote one-hot-encoded training datasets derived from a sample $S \subset L(\mathcal{M})$.

Algorithm 1 Model forward pass $M(x, mode)$

Require: Input sequence $\mathbf{x}$, Forward pass mode $\in$ {"train", "infer"}
Ensure: Pair $(\hat{\mathbf{y}}, \hat{\mathbf{s}})$ of predicted outputs and automaton states, resp.
1: $\hat{\mathbf{y}}, \hat{\mathbf{s}} \leftarrow [\,], [\,]$
2: $s \leftarrow one\_hot\_encoding(q_0)$
3: **for** $t \leftarrow 1$ **to** $\#steps(\mathbf{x})$ **do**
4: $\quad h = af(W_{hx}x^{<t>} + W_{hs}s + b_h)$
5: $\quad$ ▷ $af \in \{ReLU, \tanh\}$
6: $\quad \hat{y}^{<t>} \leftarrow softmax(W_y h + b_y)$
7: $\quad \hat{s}^{<t>} \leftarrow softmax(W_s h + b_s)$
8: $\quad$ **if** $mode =$ "$train$" **then**
9: $\quad\quad s \leftarrow \hat{s}^{<t>}$
10: $\quad$ **else** ▷ $mode =$ "$infer$"
11: $\quad\quad s \leftarrow hardmax(W_s h + b_s)$
12: $\quad$ **end if**
13: **end for**
14: **return** $(\hat{\mathbf{y}}, \hat{\mathbf{s}})$

Algorithm 1 initializes the output and state sequences $\hat{\mathbf{y}}$ and $\hat{\mathbf{s}}$ to the empty sequences and the hidden state s of the RNN to the one-hot encoding of the fixed initial state q_0. For every input symbol $x^{<t>}$, Lines 4–12 perform the equations defining the RNN, i.e., applying affine transformation using weights and an activation function. At each step t, we compute and store the predicted output $\hat{y}^{<t>}$ (Line 6) and the predicted state $\hat{s}^{<t>}$ (Line 7) in $\hat{\mathbf{y}}$ and $\hat{\mathbf{s}}$, respectively. In the "`train`" mode, we pass $\hat{s}^{<t>}$ as hidden state to the next time step (Line 9). In the "`infer`" mode used for automaton extraction, we apply a hardmax on the hidden state (Line 11) so that exactly one state is predicted.

Training. The architecture is trained by minimizing a prediction loss between $y^{<t>}$ and $\hat{y}^{<t>}$ along with a regularization loss: the cross-entropy of the state distribution $s^{<t>}$ w.r.t. to the state with the highest probability in $s^{<t>}$. On the one hand, our regularization design encourages the RNN to reuse a state at subsequent steps once it has been selected at the current step since this contributes to decreasing the regularization loss. On the other hand, it encourages using as few states as possible overall since any additional used state contributes to increasing the regularization loss. Minimizing our regularization of choice forces the RNN to increase the certainty about the predicted state. This ensures that the hidden states tend to be approximately one-hot-encoded vectors where the index of the maximal component corresponds to the state of a Mealy machine accepting the same language. Note that directly using a discrete state representation is not beneficial when training with gradient descent. Algorithm 2 implements the training in PyTorch-like [23] style. Its parameters are the training dataset D, a sample of the language to be learned, the learning epochs, and a regularization factor, which controls the influence of state regularization. The training is performed using the gradient descent-based Adam optimizer [24]. The algorithm performs up to $\#epochs$ loops over the training data. An epoch processes each trace in the training data referred to as an episode (Lines 4–21). For the actual training, we perform a forward pass in "`train`" mode and compute the overall loss from the prediction and state regularization losses (Lines 5–9). Lines 10 and 11 update the RNN parameters, i.e., the weights.

Training stops when the prediction accuracy of the RNN operated as an automaton reaches 100% or $\#epochs$ episodes have been performed. To calculate accuracy, we perform a forward pass in the "`infer`" mode in Line 15 and compute the average accuracy in Line 16. Upon finishing the training, Algorithm 2 returns a Boolean variable indicating if the prediction accuracy converged to 100% and the trained RNN.

Algorithm 2 RNN Training $train(M, D)$

Require: Initialized RNN model M, Training dataset $D = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$, $\#epochs$, Regularization factor C
Ensure: Pair of Boolean variable $converged$ indicating accuracy and trained RNN model M
1: $optimizer \leftarrow Adam(M)$
2: $converged \leftarrow \bot$
3: **for** $i \leftarrow 1$ **to** $\#epochs$ **do**
4: $\quad$ **for** $(\mathbf{x}, \mathbf{y}) \in D$ **do**
5: $\quad\quad \hat{\mathbf{y}}_{\text{tr}}, \hat{\mathbf{s}}_{\text{tr}} \leftarrow M(\mathbf{x}, \text{"}train\text{"})$
6: $\quad\quad loss_{\mathbf{y}} \leftarrow cross\_entropy(\mathbf{y}, \hat{\mathbf{y}}_{\text{tr}})$
7: $\quad\quad loss_{\mathbf{s}} \leftarrow cross\_entropy(hardmax(\hat{\mathbf{s}}_{\text{tr}}), \hat{\mathbf{s}}_{\text{tr}})$
8: $\quad\quad loss_{\mathbf{s}} \leftarrow C \times loss_{\mathbf{s}}$
9: $\quad\quad loss \leftarrow loss_{\mathbf{y}} + loss_{\mathbf{s}}$
10: $\quad\quad loss.backward()$
11: $\quad\quad optimizer.step()$
12: $\quad$ **end for**
13: $\quad acc_{\text{inf}} \leftarrow 0$
14: $\quad$ **for** $(\mathbf{x}, \mathbf{y}) \in D$ **do**
15: $\quad\quad \hat{\mathbf{y}}_{\text{inf}}, \hat{\mathbf{s}}_{\text{inf}} \leftarrow M(\mathbf{x}, \text{"}infer\text{"})$
16: $\quad\quad acc_{\text{inf}} \leftarrow acc_{\text{inf}} + accuracy(\mathbf{y}, \hat{\mathbf{y}}_{\text{inf}})/|D|$
17: $\quad$ **end for**
18: $\quad$ **if** $acc_{\text{inf}} = 100\%$ **then**
19: $\quad\quad converged \leftarrow \top$
20: $\quad\quad$ **break**
21: $\quad$ **end if**
22: **end for**
23: **return** $(converged, M)$

The purpose of the trained RNN model is not to predict outputs of new inputs, unseen during training, but to help with inferring an automaton that produces the training data. This automaton shall be used to predict the outputs corresponding to (new) inputs. Thus, we use all available data for training the RNN and aim at achieving perfect accuracy on the training data. Perfect accuracy on the training set gives us the confidence that the internal state representation of the learned RNN model corresponds to the true (partial) automaton that produced that data. In cases where the training data does not cover all states and transitions of the full true automaton, we might learn a partial automaton missing some states and

Algorithm 3 Automaton extraction from RNN $extract(M, D)$

Require: Trained RNN model M, Training data $D = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$
Ensure: Automaton transitions T
1: $T \leftarrow \{\}$
2: **for** $episode \leftarrow (\mathbf{x}, \mathbf{y}) \in D$ **do**
3: $\hat{\mathbf{y}}, \hat{\mathbf{s}} \leftarrow M(\mathbf{x}, \text{“}infer\text{”})$
4: $s\_from \leftarrow 0$
5: **for** $t \leftarrow 1$ **to** $\#steps(episode)$ **do**
6: $s\_to \leftarrow argmax(\hat{s}^{<t>})$
7: $in, out \leftarrow argmax(x^{<t>}), argmax(y^{<t>})$
8: **if** $out = argmax(\hat{y}^{<t>})$ **then**
9: $T \leftarrow T \cup \{(s\_from, s\_to, in/out)\}$
10: **else**
11: **break**
12: **end if**
13: $s\_from \leftarrow s\_to$
14: **end for**
15: **end for**
16: **return** T

transitions. Using all available data for training reduces the possibility to learn just a partial automaton.

Automaton extraction from a trained RNN. Given a trained RNN model, we extract the corresponding automaton with Algorithm 3. We represent the automaton of a Mealy machine by its set of transitions in the following form:

$$T = \{(s, s', i/o) \mid s, s' \in Q \wedge i \in I \wedge o \in O \wedge \delta(s, i) = s' \wedge \lambda(s, i) = o\}.$$

Algorithm 3 starts by initializing T to the empty set. Then, it iterates through all episodes, i.e., all traces, from the training set D. At each iteration (Lines 3–13) it first runs the RNN model M on the one-hot-encoded input sequence $\mathbf{x}$ of the current episode (Line 3) to obtain the corresponding predicted output symbols and transition state sequence $\hat{\mathbf{y}}$ and $\hat{\mathbf{s}}$, respectively. For this purpose, we perform the forward pass implemented by Algorithm 1 in the “`infer`” mode. This mode uses the *hardmax* operation to compute the encoded state in each step to ensure stability of the extraction process. A well-trained RNN with our architecture will traverse states that are close to being one-hot-encoded. The states are generally not perfectly one-hot-encoded due to numerical imprecisions and the nature of RNN training. Using *hardmax* to compute state suppresses the accumulation of such small imprecision. This is especially relevant when processing long training episodes during automaton extraction. Using the *softmax* operation, we might get ambiguous extraction results, which manifest as non-deterministic transitions.

Lines 6–13 iterate through all steps of the current episode. All episodes start from the initial state q_0 which, by construction, is assigned the label 0. Thus, we initialize the first state to 0 (Line 4). If the predicted output symbol matches the label at the current step (Line 8), then T is extended by a triple encoding a transition, which is built from the source/target states and the input/output symbols of the current step. By applying $argmax$ on the one-hot encoded input $x^{<t>}$ and output $y^{<t>}$ we get integer-valued discrete representations of them (Line 7). The actual corresponding input symbol, and respective output symbol, are obtained from the input, respective output, and symbol alphabet through an appropriate indexed mapping. For simplicity, we do not show this mapping here. If the predicted output does not match the expected value, the current and remaining steps of the current episode are ignored and the algorithm moves to the next episode (Line 2). An episode consists of a sequence of adjacent steps (or transitions) in the automaton, i.e., the next step starts from the state where the current step ended (Line 13). After processing all training data traces, Algorithm 3 returns the extracted automaton with transitions T. Note that the extracted automaton might not include all states that can be one-hot encoded with a vector with the length of $s^{<t>}$. Hence, the number of states of the extracted automaton might be smaller than the length of $s^{<t>}$.

3.3 Minimal automaton learning

In the following, we explain how the previously proposed algorithms can be used to learn a minimal automaton. In this case, we assume that we do not know in advance the exact number of states of a minimal automaton. Using an iterative approach, we adjust downward the upper bound on the maximal number of states allowed to create an automaton that correctly models the behavior of the given dataset. This iterative approach is repeated until the number of states cannot be reduced any further.

Algorithm 4 describes our iterative learning approach. For learning, we require a dataset D, which is a sample of the language to be learned, and a specific learning strategy. We distinguish between two learning strategies: *best effort* (“`bestEffort`”) and *exhaustive* (“`exhaustive`”). In the best-effort strategy, a learning iteration (Lines 5–22) ends upon the first successful attempt, if any, to learn an automaton which perfectly explains the training data (Line 19). In contrast, the exhaustive strategy performs all $\#runs$ attempts to learn an automaton that is correct to the dataset and selects among them the minimal one, if any, for further processing. Consequently, the exhaustive strategy tries within the given budget of $\#runs$ to find an even smaller automaton with the known upper bound of the current learning iteration, even if an automaton with the current upper bound has already been found. That is, the exhaustive strategy always executes all $\#runs$ iterations of the for-loop (Line 7), whereas the best-effort strategy at most $\#runs$. Algorithm 4 returns a triplet consisting of (i) the best learned automaton, (ii) a Boolean variable confirming that, at the last learning iteration, an automaton of the same size has been learned as in the

previous iteration, meaning that no smaller automaton could be further learned, and (iii) the number of performed learning iterations. The best learned automaton corresponds to an automaton with the fewest number of states learned in all iterations. The Boolean variable indicates whether the RNN model learned the same smallest automaton after setting the upper bound to the minimal number of states found so far.

Algorithm 4 Minimal automaton learning $fixpoint(D, strategy)$

Require: Trials budget $\#runs$, Training dataset $D = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_m, \mathbf{y}_m)\}$, Minimization strategy $\in$ {"bestEffort", "exhaustive"}
Ensure: Triplet of best learned automaton $A_{\min}$, and Boolean variable $approved_{\min}$ indicating whether a fixpoint has been reached, and number of learning iterations $\#it$

```
1: A_min ← IOPTA(D)
2: #it ← 0
3: repeat
4:    #it ← #it + 1
5:    states_min ← states_n(A_min)
6:    approved_min ← ⊥
7:    for i ← 1 to #runs do
8:       M ← RNN(states_min)
9:       converged, M ← train(M, D)
10:      if converged then
11:         A ← extract(M, D)
12:         A ← minimize(A)
13:         if states_n(A) < states_n(A_min) then
14:            A_min ← A
15:         else
16:            approved_min ← ⊤
17:         end if
18:         if strategy = "bestEffort" then
19:            break
20:         end if
21:      end if
22:   end for
23: until states_min = states_n(A_min)
24: return (A_min, approved_min, #it)
```

Algorithm 4 starts by creating an IOPTA from the given dataset D, as described in Sect. 2.2, and initializing the number of learning iterations $\#it$ to 0. The generated tree represents the initial automaton and provides the first upper bound on the number of states based on the number of nodes in the tree. We then start our iterative learning procedure. A learning iteration represents one iteration of the repeat-loop and includes the block from Lines 5–22.[2] We terminate the learning procedure if the learned automaton cannot be further minimized.

Let $states_n(A)$ be a function that returns the number of states of a given FSM A. In Line 5, we save in $states_{\min}$ the states number of the currently best learned automaton $A_{\min}$, i.e., the automaton with the fewest states learned so far. In the next step, we initialize a Boolean variable $approved_{\min}$ to *false*, represented by $\bot$. This variable indicates whether the RNN model learns again an automaton with the number of states equal to $states_{\min}$ after setting the upper bound of states to $states_{\min}$ (Line 8) at the current iteration. In the following (Lines 7–22), we attempt to reduce the number of states further to minimize the automaton $A_{\min}$ with our RNN-based learning technique.

The attempt to minimize the automaton is limited to a maximum number of runs. In Line 8, we initialize our RNN model M based on the current upper bound on the number of states. In terms of the RNN cell architecture we introduced in Sect. 3.1, the given number of states defines the size of the state vector $s^{<t>}$. We then train the RNN model M on the provided dataset as described in Algorithm 2 (Line 9). Hence, we obtain the trained RNN model M and a Boolean variable indicating whether the trained model achieved 100% accuracy in predicting the outputs of the provided dataset D. If the RNN model converges to 100% accuracy, from Lines 11–20, we (i) extract the automaton, (ii) check the automaton size, and (iii) stop trying to learn further automata with potentially fewer number of states if the best-effort strategy was selected.

(i) First, we extract the automaton A as described in Algorithm 3. Since we do not know the exact number of states, we may learn an automaton that has more states than the minimal automaton representing the dataset D. The RNN model might extract an automaton that has states that cannot be distinguished. As described in Sect. 2.2, we can further minimize an automaton by grouping indistinguishable states. For this, we merge all indistinguishable states of the extracted automaton A and create an equivalent automaton containing only distinguishable states in Line 12. Note that any kind of minimization on A does not affect the behavior learned by the RNN model.

(ii) We then compare the size of the minimized automaton A and the size of the current minimal automaton $A_{\min}$ (Line 13). If the newly extracted automaton A has fewer states than the previously learned minimal automaton $A_{\min}$, A becomes the new minimal automaton. In the case that we cannot further reduce the automaton size, we set $approved_{\min}$ to *true*, represented by $\top$, indicating that the fixpoint has been reached (Line 16).

(iii) If we use the best-effort strategy, we do not execute the remaining runs of the current learning iteration after learning the first automaton perfectly explaining the training data (Line 19). In the exhaustive strategy, on the other hand, we continue the current learning iteration until the entire budget $\#runs$ is consumed and train new RNN models with the minimum number of states from the previous learning iteration as upper bound for the size of the state vector $s^{<t>}$. After the maximum number of runs has been performed, $A_{\min}$ contains the best among all automata, if any, learned during the

[2] Note that one learning iteration of our learning algorithm may include several, at most $\#runs$, RNN trainings.

Table 1 Description of Tomita grammars

Grammar	Description	# States
Tomita 1	Strings of the form 1*	2
Tomita 2	Strings of the form (1 0)*	3
Tomita 3	Strings that do not include an odd number of consecutive 0 symbols following an odd number of consecutive 1 symbols	5
Tomita 4	Strings without more than 2 consecutive 0 symbols	4
Tomita 5	Even strings with an even number of 0 and 1 symbols	4
Tomita 6	Strings where the difference between the numbers of 0s and 1s is divisible by three	3
Tomita 7	Strings of the form 0*1*0*1*	5

$\#runs$ attempts to learn an automaton perfectly explaining the training data.

Finally, Algorithm 4 terminates if at the current learning iteration (i) either the fixpoint is reached, i.e., an automaton with the same number of states as in the previous learning iteration is learned, or (ii) no automaton at all perfectly explaining the training data can be learned due to insufficient budget. The algorithm returns the best learned automaton, if any, otherwise $IOPTA(D)$, along with the information whether the fixpoint has been reached and the number of performed learning iterations.

4 Case studies

4.1 Case study subjects

4.1.1 Tomita grammars

We use Tomita grammars [25] to evaluate our approach. These grammars are popular subjects in the evaluation of formal-language-related work on RNNs [26–28], as they possess various features, while they are small enough to facilitate manual analysis. All of the grammars are defined over the input symbols 0 and 1. We transformed the ground-truth DFAs into Mealy machines, thus the outputs are either *true* (string accepted) or *false* (string not accepted). Table 1 contains for each Tomita grammar a short description of the accepted strings and the number of states of the smallest Mealy machine accepting the corresponding language. For example, Tomita 5 accepts strings depending on parity of 0 and 1 symbols. The language described by Tomita 5 has been used to illustrate the L^* algorithm [5]. Automata accepting such languages are hard to encode using certain types of RNNs [29].

4.1.2 Bluetooth Low Energy (BLE)

To evaluate the applicability to practical problems, we consider the BLE protocol. BLE was introduced in the Bluetooth standard 4.0 as a communication protocol for low-energy devices. The BLE protocol stack implementation is different from the Bluetooth classic protocol. Pferscher and Aichernig [30] learned with L^* behavioral models of BLE devices. They presented practical challenges in the creation of an interface to enable the interaction required by active automata learning. Especially, the requirement of adequately resetting the device after each performed query raises the need for a learning technique that requires less interaction with the SUL. We selected three devices from their case study. The selected devices have a similarly large state space and show more stable deterministic behavior than other devices in the case study by Pferscher and Aichernig [30] which would have required advanced data processing that filters out non-deterministic behavior. Table 2 states the investigated devices, the used System-on-Chip, and the running application. In the following, we refer to the devices by the System-on-Chip names. The running application initially sends BLE advertisements and accepts a connection with another BLE device. If a connection terminates, the device again sends advertisements. The generated behavioral model should formalize the implemented connection procedure. Compared to existing work [30], we extended the considered nine inputs by another input that implements the termination request, which indicates the termination of the connection by one of the two devices. Since every input must be defined for every state, the complexity of learning increases with the size of the input alphabet. Hence, the BLE case study provides a first impression of the scalability of our presented learning technique.

Figure 7 depicts a behavioral model of the CYBLE-416045-02. For the illustration, some input and output labels have been simplified and combined by a '+'-symbol. The model shows that a connection can be established with a connection request and terminated by a scan or termination request. A version request is only answered once during an active connection. Pferscher and Aichernig [30] provide a link to complete models of all three considered examples.

Table 2 Investigated Bluetooth Low Energy (BLE) devices including the running application

Manufacturer (Board)	System-on-chip	Application	# States
Texas instruments (LAUNCHXL-CC2650)	CC2650	Project zero	5
Cypress (CY8CPROTO-063-BLE)	CYBLE-416045-02	Find me target	3
Nordic (decaWave DWM1001-DEV)	nRF52832	Nordic GATTS	5

The column #States indicates the number of states of the models created by active automata learning

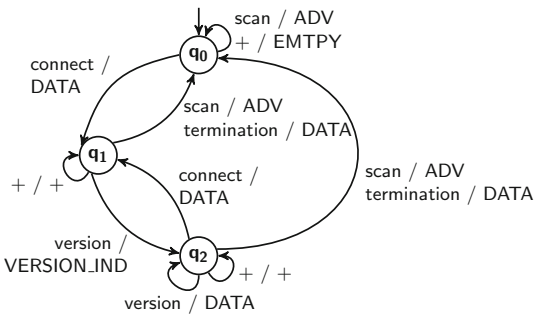

Fig. 7 Simplified model of the CYBLE-416045-02 ('+' abbreviates inputs/outputs)

4.2 Experimental setup

We demonstrate the effectiveness of our approach on both (i) the canonical Tomita grammars from the literature [26, 28, 31], and (ii) the physical BLE devices that were introduced in the previous section. Both evaluations (i) and (ii) are performed with AAL data and successively with randomly generated data.

We consider the automata learned with the active automata learning (AAL) algorithm L^* and the corresponding data produced by AAL as given. We call these the AAL automata and AAL data, respectively. In general, we do not require AAL to be executed in advance. AAL rather provides an outline for the evaluation of our proposed RNN architecture.

Our case study is based on the following four experimental setups: (i) We first evaluate the capability of our RNN architecture to learn the correct automaton when the number of states of the AAL automaton is known in advance. This number of states k is used to set the size of $s^{<t>}$ in the RNN architecture. The AAL automaton itself is only used as ground truth. It does not affect the RNN training procedure in any way other than defining the size of $s^{<t>}$. We say that the RNN learned the correct automaton if the automaton extracted from the trained RNN according to Algorithm 3 is isomorphic to the minimal ground-truth automaton.

(ii) We then evaluate the capability of our approach to learn the correct automaton without making any assumption on the number of automaton states other than a data-based upper bound, as shown in Algorithm 4. We fix $\#runs = 10$ and perform a statistical evaluation by running 10 times Algorithm 4 for each use case. For simplicity, we use the same values from (i) for the RNN architecture and training hyperparameters (e.g., number of hidden layers and neurons per layer, activation function, learning rate, regularization factor, etc.), except the size of the state vector $s^{<t>}$ which is now controlled by the algorithm itself. In practice, these hyperparameters can be tuned by running Algorithm 4 with different hyperparameter values and selecting those with better convergence properties.

(iii) Furthermore, we evaluate the effects of changing RNN hyperparameters.

(iv) Finally, we compare our proposed RNN-based learning technique with a classic passive learning technique from the literature. To enable a fair comparison, we again use the randomly generated data.

AAL Data. Firstly, we use the AAL data as RNN training data. This finite set of traces from AAL is complete in the sense that passive automata learning could learn a behavioral model with k states that conforms to the model learned by AAL. For AAL data generation, we used the active automata learning library AALpy [6], which implements state-of-the-art algorithms including the L^*-algorithm variant for Mealy machines by Shahbaz and Groz [22]. The logged data include all performed output queries and the traces generated for conformance testing during the equivalence check. The model-based testing technique used for conformance testing provides state coverage for the intermediate learned hypotheses.

For the BLE data generation, we use a similar learning framework as Pferscher and Aichernig [30]. To collect the performed output queries during automata learning, we logged the performed BLE communication between the learning framework and the SUL. The logged traces are then post-processed to exclude non-deterministic traces. Nondeterminism might occur due to packet loss or delayed packets. In this case, the active automata learning framework repeated the output query. To clean up the logged BLE traces, we execute all input traces on the actively learned Mealy machine. If the observed output sequence deviates from the Mealy machine output, the trace is removed from the considered learning dataset.

Random Data. Secondly, we use randomly generated data as training data. That is, we are not guided by any active learning procedure to generate the training data. Instead, we simply sample random inputs from the input alphabet and observe the outputs produced by the system, i.e., the Tomita grammars or the physical BLE devices. This corresponds to a more realistic real-world scenario where the data logged

during regular system operation is the only available training data. To speed up the experiments, we use the AAL automaton instead of the real system to generate random data. More precisely, we achieve this through random walks on the AAL automaton. Each random walk represents a trace in the training data. It always starts from the initial state of the AAL automaton and collects the sequence of input–output pairs obtained by running the AAL automaton on the randomly generated inputs. We set a value max_length for the maximal length of the generated traces and the number of traces to be generated.

- For Tomita grammars, in each iteration, we produce a trace through a random walk from the initial state with a length uniformly distributed within $[1, max\_length]$. We add the produced trace to the dataset if it was not already generated before.
- For the BLE devices, we generate traces that simulate BLE sessions between real-world devices. For this, each trace ends with a terminate request indicating the end of the connection.
  Hence, we can extract such traces from a long random walk by extracting the subtraces between two subsequent terminate requests. Thus, we start a random walk from the initial state. At each step, we sample an input request or force a terminate request to ensure a maximum individual trace length of max_length.
  Every time we return to the initial state, we add the corresponding generated trace to the dataset, if not already contained. We start a new random walk and iterate as long as the dataset does not contain the desired number of traces.
  Since each episode ends in the initial state due to the final terminate request, we exploit this knowledge during the RNN training by adding to the overall loss (Algorithm 2, Line 9) the term $cross\_entropy(q_0, s^{<last>})$ corresponding to the deviation of the last RNN state $s^{<last>}$ from the initial state of the learned automaton, which is fixed to q_0 by construction.

We start with a smaller random dataset and progressively generate bigger random datasets until the RNN learns the correct automaton or a predefined time budget is consumed.

All experiments were performed with PyTorch 1.8. The evaluation (i) was performed on a Dell Latitude 5501 laptop with Intel Hexa-Core I7-9850H, 32 GB RAM, 2.60 GHz, 512 GB SSD, NVIDIA GeForce MX150, and Windows-10 OS. The evaluations (ii) and (iii) were performed due to the increased amount of RNN training sessions on a scientific cluster based on Intel(R) Xeon(R) Gold 6230R CPU @ 2.1 GHz and Ubuntu 20.04. The evaluation (iv) was performed on an Apple MacBook Pro 2019 with an Intel Quad-Core i5 running at 2.4 GHz and with 8 GB memory.

4.3 Results and discussion

The following section presents the results for our four experimental setups, followed by a discussion on the generalizability of our presented learning technique to other case studies.

4.3.1 Minimal states number is given

In this evaluation, we assume that the number of states of a minimal automaton is given. This should demonstrate the basic feasibility of our proposed RNN architecture.

Tables 3 and 4 illustrate the experimental results obtained by applying our approach to learn the automata of Tomita grammars and BLE devices, respectively, from both AAL data and random data. Compared to the same tables from our conference paper [15], we present different numbers here since we repeated the experiments under slightly different conditions for the following reasons:

- We adapted the random data generation procedure to provide exactly the desired number of unique episodes, instead of just removing the duplicates after the episodes were generated and providing the remaining episodes.
- We enhanced some procedures to support the new Algorithm 4 which is introduced in this article.

The number of traces contained in the training data is given in the column *Size* for both AAL data and random data. For the random data, it is interesting to know how many traces from the AAL data were contained also in the random data. This information is shown in the column *AAL Data Coverage* as the ratio between the number of AAL traces contained in the random data and the overall number of traces in the AAL data.

The column *Episode Lengths* contains the means and standard deviations of the lengths of the traces in the training data. The column *RNN* contains (i) the RNN architecture parameters which possibly changed across the experiments (i.e., the activation function af and the number of hidden layers #hl in Table 3 and the regularization factor C in Table 4), and (ii) the number of epochs #e and the time t required by the RNN training to learn the correct automaton.

The values of other RNN architecture parameters, which were the same in all experiments, are mentioned in the table captions. For instance, it turned out that the values 0.001 and 256 for the learning rate and the number of neurons per hidden layer, respectively, worked for all considered case studies.

For the Tomita grammars (Table 3), the value of the regularization factor C was also fixed and equal to 0.001, whereas different activation functions and numbers of hidden layers were used across the different grammars. For the BLE

devices (Table 4), the number of the hidden layers and the activation function were also fixed and equal to 1 and *ReLU*, respectively, whereas different regularization factors were used across different devices.

The results show that we could find an appropriate architecture to learn the correct automata in all considered cases, meaning that the RNN accuracy was 100% in all cases. This was expected when learning from AAL data, as the AAL data were sufficient for the L^* algorithm to learn the underlying minimum automaton. More surprising is that we could learn for all examples the correct automaton also from relatively small random datasets with a low coverage of the AAL data. Even more surprising is that for all Tomita grammars, except Tomita 2 and 3, we could significantly shrink the random datasets compared to the AAL dataset and still learn correctly. The datasets became smaller in terms of both number of traces and average trace length. Moreover, only a small fraction of the AAL data happened to be included also in the random data. This suggests that the proposed RNN architecture and training may better generalize on sparser datasets than AAL. The good performance on Tomita grammars might be attributed to the small number of automaton transitions and input/outputs alphabets that only consist of '0' and '1' symbols.

For the BLE device CYBLE-416045-02, which has much larger input and output alphabets, we could still learn the correct automaton from a random dataset containing fewer traces than the AAL data. The other two BLE devices required larger random datasets due to the higher number of transitions to be covered.

For all case studies, except Tomita 7 and 6, the same RNN architecture worked for both AAL and random data. For Tomita 7, *ReLU* worked for the AAL data, whereas *tanh* worked better for the random data. For Tomita 6, two hidden layers worked better for the random data, as opposed to a single hidden layer. Typically, the tuning process of RNN hyperparameter values depends on the dataset, and each case study involves a distinct dataset. When learning from random data, we also attempted to learn from datasets as small as possible. In the case of Tomita 6 and 7, this necessitated different values for some hyperparameters compared to when learning from AAL data.

4.3.2 Minimal states number is learned

In the following, we present the results of the evaluation for learning the minimal ground-truth automaton from scratch with Algorithm 4, i.e., without knowing in advance the minimal number of states. In the following, we denote an automaton that is isomorphic to the minimal ground-truth automaton as *the minimal AAL automaton*.

We define an 'experiment' as the endeavor to learn a minimal automaton by executing Algorithm 4, with the following

Table 3 RNN automata learning of Tomita grammars

Grammar	AAL data			Random data			
	Size	Episode lengths {avg; std}	RNN {af; #hl} {#e; t}	Size	AAL data coverage	Episode lengths {avg; std}	RNN {af; #hl} {#e; t}
Tomita 1	41	{8.4; 4.5}	{relu; 1} {5; 3 s}	10	3/41	{6.2; 3.4}	{relu; 1} {9; 1 s}
Tomita 2	73	{9.5; 5.3}	{tanh; 1} {12; 14 s}	100	15/73	{6.7; 2.3}	{tanh; 1} {4; 5 s}
Tomita 3	111	{10.4; 4.9}	{relu; 2} {79; 233 s}	500	41/111	{7.8; 1.9}	{relu; 2} {28; 305 s}
Tomita 4	83	{10.4; 5.2}	{tanh; 1} {34; 53 s}	50	9/83	{5.8; 2.5}	{tanh; 1} {11; 6 s}
Tomita 5	91	{9.3; 4.7}	{relu; 1} {16; 26 s}	50	13/91	{6.2; 2.8}	{relu; 1} {24; 15 s}
Tomita 6	68	{8.6; 4.6}	{relu; 1} {5; 5 s}	20	6/68	{5.5; 2.5}	{relu; 2} {12; 3 s}
Tomita 7	115	{10.4; 5.1}	{relu; 1} {21; 44 s}	50	10/115	{6.5; 2.3}	{tanh; 1} {31; 21 s}

The correct automaton could be learned in all cases. Fixed parameters: #neurons per hidden layer = 256, learning rate = 0.001, C = 0.001

components: (i) a training dataset, such as the AAL dataset or randomly generated data, related to a Tomita grammar or a BLE device, (ii) an RNN architecture specified by parameters like the number of hidden layers, number of neurons, activation function, learning rate, and regularization factor (C), and (iii) a training budget that includes the maximum number of runs at each iteration of the fixpoint computation, the maximum number of training epochs, and, in the case of random data, the number of episodes to be randomly generated and the maximum allowed length for a generated episode. We say that an experiment is *approved* if the value of the variable $approved_{min}$ returned by Algorithm 4 is *true*, i.e., the last fixpoint iteration return as result an isomorphic automaton to the automaton learned in the previous iteration. In general, we follow the 'best-effort' strategy if not otherwise stated.

Our minimal automaton learning procedure is inherently stochastic. There are two sources of stochasticity: (i) RNN-related stochasticity (e.g., RNN parameters initialization, samples ordering during training, etc.), and (ii) random generation of training data. To account for the RNN-related stochasticity, we repeat an experiment with the same input ten times. When learning a minimal automaton from random data, the training data are always newly randomly generated in each of the ten experiments.

Tables 5, 6, 9 and 10 show the results individually for all experiments for the Tomita grammars and BLE devices, respectively. An entry in these tables reports following information related to the corresponding experiment of a Tomita grammar, resp. BLE device:

- overall result: (i) ✓ if a minimal automaton was learned, (ii) ✗ if an automaton was learned but this is not minimal, (iii) **−** if the fixpoint has not been reached within the given budget, i.e., $approved_{\min} = \perp$ in Algorithm 4, (iv) (✓) if an automaton with the minimal number of states was learned but this is not the same as the minimal AAL automaton due to incomplete randomly generated data,
- fixpoint convergence details: $\mathrm{N} \xrightarrow{\#it} \mathrm{n}$ where N is the upper bound size of the state vector $s^{<t>}$ (i.e., the maximal number of states that can be learned) initially estimated with IOPTA computation, n is either the states number of the learned automaton or n.a. (not applicable) if no automaton could be learned, and $\#it$ is the number of the learning iterations which is equal to the number of iterations of the repeat-loop in Algorithm 4,
- FSM minimization contribution: (i) 'min(I:i)': the maximum reduction of states achieved by the FSM minimization approach, as described in Sect. 2.2, during the learning iterations, where I and i (with I > i) are the numbers of states of the extracted automaton from the trained RNN before and after FSM minimization, respec-

Table 4 RNN automata learning of BLE devices

Device	AAL data			Random data			
	Size	Episode lengths {avg; std}	RNN {C; #e; t}	Size	AAL data coverage	Episode lengths {avg; std}	RNN {C; #e; t}
CYBLE-416045-02	272	{4.0; 1.0}	{0.01; 5; 12 s}	100	17/272	{11.5; 8.6}	{0.01; 29; 59 s}
CC2650	473	{5.1; 2.1}	{0.001; 162; 13 m:30 s}	1000	82/473	{11.8; 7.8}	{0.001; 42; 14 m:48 s}
nRF52832	447	{4.6; 1.1}	{0.001; 21; 1 m:30 s}	1500	100/447	{12.1; 8.1}	{0.001; 21; 11 m:18 s}

The correct automaton could be learned in all cases. Fixed parameters: #neurons per hidden layer = 256, #hl=1, af=relu, learning rate=0.001

Table 5 Experiments individual results: minimal RNN automata learning of Tomita grammars with **#epochs = 100** (strategy = best-effort)

Use case	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07	Exp08	Exp09	Exp10
Tomita 1										
AAL	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 28	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 13	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 18	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 08	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 08	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 01 : 31	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 08	✓ $207 \xrightarrow{2} 2$ – 00 : 00 : 48	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 50	✓ $207 \xrightarrow{2} 2$ $min(3:2)$ 00 : 01 : 32
Random	✓ $34 \xrightarrow{2} 2$ – 00 : 00 : 09	✓ $31 \xrightarrow{3} 2$ – 00 : 00 : 05	✓ $48 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 05	✓ $55 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 04	✓ $44 \xrightarrow{2} 2$ – 00 : 00 : 02	✓ $38 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 02	✓ $46 \xrightarrow{3} 2$ – 00 : 00 : 05	✓ $36 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 02	✓ $56 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 04	✓ $49 \xrightarrow{2} 2$ $min(3:2)$ 00 : 00 : 06
Tomita 2										
AAL	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 05 : 12	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 02 : 55	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 53	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 50	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 47	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 02 : 05	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 51	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 38	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 03 : 49	✓ $390 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 58
Random	✓ $193 \xrightarrow{2} 3$ $min(4:3)$ 00 : 02 : 20	✓ $250 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 03	✓ $231 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 24	✓ $231 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 33	✓ $239 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 21	✓ $218 \xrightarrow{2} 3$ – 00 : 00 : 17	✓ $231 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 12	✓ $214 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 37	✓ $244 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 28	✓ $214 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 12
Tomita 3										
AAL	✓ $545 \xrightarrow{2} 5$ $min(6:5)$ 00 : 59 : 45	✓ $545 \xrightarrow{2} 5$ $min(6:5)$ 01 : 22 : 45	✓ $545 \xrightarrow{2} 5$ $min(8:5)$ 01 : 00 : 44	✓ $545 \xrightarrow{2} 5$ – 00 : 36 : 41	✓ $545 \xrightarrow{2} 5$ – 00 : 32 : 40	✓ $545 \xrightarrow{2} 5$ $min(6:5)$ 00 : 43 : 46	✓ $545 \xrightarrow{2} 5$ $min(6:5)$ 00 : 50 : 28	✓ $545 \xrightarrow{2} 5$ $min(7:5)$ 00 : 17 : 09	– $545 \xrightarrow{1} n.a.$ $n.a.$ 00 : 49 : 50	✓ $545 \xrightarrow{2} 5$ $min(6:5)$ 00 : 07 : 12
Random	✓ $728 \xrightarrow{2} 5$ $min(6:5)$ 02 : 00 : 26	✓ $743 \xrightarrow{2} 5$ $min(8:5)$ 00 : 24 : 05	✓ $733 \xrightarrow{2} 5$ $min(9:5)$ 00 : 29 : 31	✓ $733 \xrightarrow{2} 5$ – 01 : 05 : 15	✓ $739 \xrightarrow{2} 5$ $min(6:5)$ 00 : 09 : 56	✓ $741 \xrightarrow{2} 5$ $min(13:5)$ 00 : 18 : 59	✓ $723 \xrightarrow{2} 5$ $min(7:5)$ 00 : 26 : 55	✓ $749 \xrightarrow{2} 5$ $min(7:5)$ 01 : 30 : 31	✓ $751 \xrightarrow{2} 5$ $min(7:5)$ 00 : 10 : 58	✓ $756 \xrightarrow{2} 5$ $min(9:5)$ 00 : 33 : 30
Tomita 4										
AAL	✓ $459 \xrightarrow{2} 4$ – 01 : 37 : 13	– $459 \xrightarrow{1} n.a.$ $n.a.$ 00 : 48 : 23	– $459 \xrightarrow{1} n.a.$ $n.a.$ 00 : 45 : 13	✓ $459 \xrightarrow{2} 4$ – 00 : 23 : 23	✓ $459 \xrightarrow{2} 4$ – 00 : 01 : 39	✓ $459 \xrightarrow{2} 4$ – 00 : 18 : 23	✓ $459 \xrightarrow{2} 4$ – 00 : 07 : 59	✓ $459 \xrightarrow{2} 4$ – 00 : 00 : 56	✓ $459 \xrightarrow{2} 4$ – 00 : 11 : 07	✓ $459 \xrightarrow{2} 4$ – 00 : 03 : 43
Random	✓ $121 \xrightarrow{2} 4$ $min(6:4)$ 00 : 05 : 15	✓ $140 \xrightarrow{2} 4$ $min(5:4)$ 00 : 10 : 03	✓ $145 \xrightarrow{2} 4$ $min(5:4)$ 00 : 08 : 27	✓ $133 \xrightarrow{2} 4$ – 00 : 02 : 51	✓ $132 \xrightarrow{2} 4$ $min(5:4)$ 00 : 04 : 29	✓ $133 \xrightarrow{2} 4$ $min(5:4)$ 00 : 09 : 25	✓ $139 \xrightarrow{2} 4$ $min(5:4)$ 00 : 02 : 32	✓ $129 \xrightarrow{2} 4$ $min(5:4)$ 00 : 00 : 30	– $139 \xrightarrow{1} n.a.$ $n.a.$ 00 : 08 : 52	– $141 \xrightarrow{1} n.a.$ $n.a.$ 00 : 08 : 40

Table 5 continued

Use case	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07	Exp08	Exp09	Exp10
Tomita 5										
AAL	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 01 : 38 : 21	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 51 : 00	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 45 : 34	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 05	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 33	✓ 436 $\xrightarrow{2}$ 4 – 00 : 03 : 42	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 03	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 18	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 12	– 436 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 29 : 16
Random	✓ 121 $\xrightarrow{2}$ 4 – 00 : 12 : 24	✓ 140 $\xrightarrow{2}$ 4 *min*(5 : 4) 00 : 04 : 02	✓ 145 $\xrightarrow{2}$ 4 – 00 : 03 : 52	✓ 133 $\xrightarrow{2}$ 4 – 00 : 03 : 55	✓ 132 $\xrightarrow{2}$ 4 *min*(5 : 4) 00 : 01 : 19	✓ 133 $\xrightarrow{2}$ 4 *min*(5 : 4) 00 : 00 : 47	✓ 139 $\xrightarrow{2}$ 4 – 00 : 07 : 55	✓ 129 $\xrightarrow{2}$ 4 *min*(5 : 4) 00 : 03 : 40	✓ 139 $\xrightarrow{2}$ 4 – 00 : 06 : 46	✓ 141 $\xrightarrow{2}$ 4 – 00 : 02 : 21
Tomita 6										
AAL	✓ 308 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 06 : 19	✓ 308 $\xrightarrow{2}$ 3 *min*(4 : 3) 00 : 01 : 15	✓ 308 $\xrightarrow{2}$ 3 *min*(6 : 3) 00 : 02 : 35	✓ 308 $\xrightarrow{2}$ 3 *min*(6 : 3) 00 : 02 : 11	✓ 308 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 01 : 50	✓ 308 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 00 : 28	✓ 308 $\xrightarrow{2}$ 3 *min*(7 : 3) 00 : 03 : 38	✓ 308 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 00 : 23	✓ 308 $\xrightarrow{2}$ 3 *min*(6 : 3) 00 : 00 : 24	✓ 308 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 01 : 30
Random	✓ 61 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 00 : 56	✓ 65 $\xrightarrow{2}$ 3 – 00 : 00 : 16	✓ 80 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 01 : 06	✓ 80 $\xrightarrow{2}$ 3 – 00 : 00 : 13	✓ 67 $\xrightarrow{2}$ 3 *min*(5 : 3) 00 : 00 : 22	✓ 59 $\xrightarrow{2}$ 3 – 00 : 00 : 14	✓ 70 $\xrightarrow{2}$ 3 – 00 : 00 : 13	✓ 67 $\xrightarrow{2}$ 3 *min*(4 : 3) 00 : 00 : 11	✓ 71 $\xrightarrow{2}$ 3 *min*(4 : 3) 00 : 00 : 17	✓ 79 $\xrightarrow{2}$ 3 – 00 : 00 : 12
Tomita 7										
AAL	– 564 $\xrightarrow{1}$ *n.a.* *n.a.* 02 : 00 : 55	– 564 $\xrightarrow{1}$ *n.a.* *n.a.* 01 : 07 : 59	✓ 564 $\xrightarrow{2}$ 5 – 00 : 38 : 28	✓ 564 $\xrightarrow{2}$ 5 – 00 : 25 : 36	✓ 564 $\xrightarrow{2}$ 5 – 00 : 39 : 47	– 564 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 44 : 26	✓ 564 $\xrightarrow{2}$ 5 – 00 : 17 : 15	– 564 $\xrightarrow{1}$ *n.a.* *n.a.* 00 : 44 : 24	✓ 564 $\xrightarrow{2}$ 5 – 00 : 50 : 47	✓ 564 $\xrightarrow{2}$ 5 – 00 : 23 : 38
Random	✓ 121 $\xrightarrow{2}$ 5 – 00 : 07 : 38	– 140 $\xrightarrow{2}$ *n.a.* – 00 : 16 : 57	✓ 145 $\xrightarrow{2}$ 5 *min*(6 : 5) 00 : 07 : 23	✓ 133 $\xrightarrow{2}$ 5 – 00 : 03 : 13	✓ 132 $\xrightarrow{2}$ 5 – 00 : 00 : 45	✓ 133 $\xrightarrow{2}$ 5 – 00 : 02 : 04	✓ 139 $\xrightarrow{2}$ 5 *min*(6 : 5) 00 : 02 : 18	– 129 $\xrightarrow{2}$ *n.a.* *min*(6 : 5) 00 : 07 : 09	✓ 139 $\xrightarrow{2}$ 5 *min*(6 : 5) 00 : 03 : 52	✓ 141 $\xrightarrow{2}$ 5 – 00 : 03 : 32

Table 6 Experiments individual results: minimal RNN automata learning of Tomita 5 and 7 grammars from AAL data with **#epochs = 200** (strategy = best-effort)

Use case	Exp11	Exp12	Exp13	Exp14	Exp15	Exp16	Exp17	Exp18	Exp19	Exp20
Tomita 5										
	❙	✓	✓	✓	✓	✓	✓	❙	✓	✓
AAL	$436 \xrightarrow{1} n.a.$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{1} n.a.$	$436 \xrightarrow{2} 4$	$436 \xrightarrow{2} 4$
	n.a.	$min(7:4)$	$min(5:4)$	$min(5:4)$	$min(5:4)$	$min(5:4)$	$min(9:4)$	*n.a.*	$min(5:4)$	$min(6:4)$
	01 : 01 : 00	02 : 09 : 26	00 : 28 : 15	00 : 14 : 08	00 : 33 : 50	00 : 41 : 38	00 : 14 : 06	01 : 15 : 37	00 : 02 : 18	00 : 35 : 14
Tomita 7										
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
AAL	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$	$564 \xrightarrow{2} 5$
	–	$min(6:5)$	$min(6:5)$	–	$min(6:5)$	–	–	$min(6:5)$	–	–
	00 : 44 : 46	00 : 24 : 59	00 : 06 : 50	00 : 43 : 51	01 : 08 : 26	00 : 23 : 02	00 : 36 : 53	00 : 33 : 07	00 : 08 : 30	00 : 21 : 39

tively, (ii) '−': if I = i, i.e., the FSM minimization could not further reduce the number of states of the automaton extracted from the trained RNN at any learning iteration, or (iii) 'n.a.': if no automaton could be learned,

- execution time hh:mm:ss representing the number of hours, minutes and seconds taken to run the experiment.

Tables 7, 8, 11 and 12 summarize the statistics averaged over the ten experiments performed for each Tomita grammar and BLE device with AAL and random training data. The meaning of the columns is as following:

1. "Autom. Learned": the percentage of experiments overall where an automaton could be learned, i.e., the percentage of approved experiments,
2. "Learned Autom. is min.": the percentage over the approved experiments, where the learned automaton is minimal,
3. "Minim. Boost": the percentage of the experiments overall where the extracted automaton from the trained RNN at some fixpoint iteration could be minimized with the FSM minimization, hence resulting in a reduction of the fixpoint iterations number,
4. "Initial State Vector Size": the mean and standard deviation over all experiments of the upper bound estimated with IOPTA computation which is used to set the size of the state vector $s^{<t>}$ in the RNN architecture (Fig. 6) for the first fixpoint iteration,
5. "#Iterations": the mean and standard deviation over the accepted experiments of the number of iterations required to converge,
6. "Non-min. Autom. #states": the mean and standard deviation of the learned automaton states number over all accepted experiments where a non-minimal automaton was learned,
7. "Learning Time": the mean and standard deviation over all accepted experiments of the execution time, both in form of hh:mm:ss.

Tomita grammars. First, a budget of 100 epochs has been employed for all Tomita grammars. As we can see in Tables 5 and 7, this budget was sufficient, indicated by ✓, to learn the minimal AAL automaton for most Tomita grammars in most experiments, except for Tomita 5 and Tomita 7 with AAL training data. For Tomita 5, we only have one approved experiment where it was also possible to learn the minimal automaton. In the remaining nine experiments, indicated by ❙, already at the first learning iteration, in none of the ten runs we could learn an automaton perfectly explaining the training data, i.e., training the model does not converge to 100% accuracy within the given budget. This strongly suggests that the maximum number of epochs of 100 was too low and stopped the RNN training too early. Similarly for Tomita

Table 7 Experiments summary: minimal RNN automata learning of Tomita grammars with **#epochs = 100** (strategy = best-effort)

Use case	Autom. learned (%)	Learned autom. is min. (%)	Minim. boost (%)	Initial state vector size {avg; std}	#Iterations {avg; std}	Non-min. autom. #states {avg; std}	Learning time {avg; std}
Tomita 1							
AAL	100	100	90	{207; 0}	{2; 0}	n.a	{00:00:36; 00:00:33}
Random	100	100	60	{44; 9}	{2.2; 0.4}	n.a	{00:00:04; 00:00:02}
Tomita 2							
AAL	100	100	100	{390; 0}	{2; 0}	n.a	{00:02:18; 00:01:22}
Random	100	100	90	{227; 17}	{2; 0}	n.a	{00:00:45; 00:00:43}
Tomita 3							
AAL	90	100	78	{545; 0}	{2; 0}	n.a	{00:43:28; 00:23:15}
Random	100	100	90	{740; 11}	{2; 0}	n.a	{00:43:01; 00:37:01}
Tomita 4							
AAL	80	100	0	{459; 0}	{2; 0}	n.a	{00:20:33; 00:31:59}
Random	80	100	88	{135; 7}	{2; 0}	n.a	{00:05:27; 00:03:31}
Tomita 5							
AAL	10	100	0	{436; 0}	{2; 0}	n.a	{00:03:42; 00:00:00}
Random	100	100	40	{135; 7}	{2; 0}	n.a	{00:04:42; 00:03:29}
Tomita 6							
AAL	100	100	100	{308; 0}	{2; 0}	n.a	{00:02:03; 00:01:50}
Random	100	100	50	{70; 8}	{2; 0}	n.a	{00:00:24; 00:00:20}
Tomita 7							
AAL	60	100	0	{564; 0}	{2; 0}	n.a	{00:32:35; 00:12:30}
Random	80	100	40	{135; 7}	{2; 0}	n.a	{00:03:51; 00:02:28}

Table 8 Experiments summary: minimal RNN automata learning of Tomita 5 and 7 grammars from AAL data with **#epochs = 200** (strategy = best-effort)

Use case	Autom. learned (%)	Learned autom. is min (%)	Minim. boost (%)	Initial state vector size {avg; std}	#Iterations {avg; std}	Non-min. autom. #states {avg; std}	Learning time {avg; std}
Tomita 5							
AAL	80	100	100	{436; 0}	{2; 0}	n.a	{00:37:22; 00:39:28}
Tomita 7							
AAL	100	100	40	{564; 0}	{2; 0}	n.a	{00:31:12; 00:18:29}

7, but less drastically than Tomita 5, there were four unapproved experiments in which no automaton could be learned. We thus increased the maximal number of epochs to 200 and repeated the ten experiments for Tomita 5 and Tomita 7 with the AAL training data. The results reported in Tables 6 and 8 show significant improvements with the increased budget.

All learned automata have the minimal number of states required to model the corresponding ground-truth system. After having increased the maximal training epochs number for Tomita 5 and 7, there were very few unapproved experiments overall. Remarkably, in 9 out of 16 Tomita training setups, the minimal AAL automaton could be learned in all conducted experiments. Moreover, the number of unapproved experiments can be further reduced, if necessary, by further increasing the budget. Importantly, Algorithm 4 directly provides information on whether it is necessary to address an unapproved experiment by increasing the budget. This information is conveyed through the $approved_{min}$ variable.

It is worth noting that for all Tomita grammars, except Tomita 3, learning a minimal automaton was more efficient with random data than AAL data. A randomly generated dataset which was smaller than the AAL dataset was sufficient to learn the minimal AAL automaton. This suggests that the AAL data, which is an optimal dataset for the L^* algorithm to learn the minimal automaton, is not necessarily also an optimal dataset for the RNN.

The FSM minimization contributed in most cases to speed up the fixpoint convergence by reducing the number of necessary iterations. In all cases (incl. the ones with no FSM minimization contribution), except one experiment for Tomita 1, the minimal AAL automaton could be learned in the minimum number of fixpoint iterations which is 2. This means that for most experiments, we could learn a minimal automaton within the first fixpoint iteration using a data-based upper bound on the number of states. The second iteration is only required to approve the previous learning result.

Finally, the above results were obtained by employing the best-effort strategy which was sufficient to learn the minimal AAL automaton for all Tomita grammars.

BLE devices. We first employed the best-effort strategy for all BLE devices in all experiments. This was sufficient to learn the minimal automaton in most cases, except the CC2650 device with AAL data, as we can see in Tables 9 and 11. Remarkably, only two unapproved experiments occurred overall. For the nRF52832 device in particular, all test setups delivered the minimum AAL automaton.

For the CYBLE-416045-02 and CC2650 devices with random data, there were a few experiments in which an automaton with the same number of states as the ground-truth automaton could be learned but with different transitions. This experiments are indicated by (✓). This suggests that

Table 9 Experiments individual results: minimal RNN automata learning of BLE devices with **strategy = best-effort** (#epochs = 200)

Use case	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07	Exp08	Exp09	Exp10
CYBLE-416045-02										
AAL	✓ $547 \xrightarrow{3} 3$ $min(6:5)$ 00 : 01 : 52	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 12	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 04	✓ $547 \xrightarrow{3} 3$ $min(5:4)$ 00 : 01 : 30	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 02 : 25	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 01 : 02	✓ $547 \xrightarrow{3} 3$ $min(5:4)$ 00 : 01 : 14	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 00 : 49	✓ $547 \xrightarrow{3} 3$ $min(6:5)$ 00 : 01 : 47	✓ $547 \xrightarrow{2} 3$ $min(4:3)$ 00 : 02 : 08
Random	✓ $970 \xrightarrow{3} 3$ $min(5:4)$ 00 : 51 : 53	✓ $930 \xrightarrow{2} 3$ $min(4:3)$ 00 : 09 : 13	(✓) $919 \xrightarrow{2} 3$ – 00 : 05 : 10	✓ $922 \xrightarrow{2} 3$ – 00 : 04 : 07	(✓) $814 \xrightarrow{3} 3$ – 01 : 53 : 27	✓ $985 \xrightarrow{2} 3$ – 00 : 06 : 13	(✓) $982 \xrightarrow{2} 3$ – 00 : 03 : 05	✓ $1020 \xrightarrow{3} 3$ – 00 : 14 : 10	– $979 \xrightarrow{4} n.a.$ – 02 : 03 : 28	✓ $940 \xrightarrow{3} 3$ – 00 : 13 : 16
CC2650										
AAL	✗ $1180 \xrightarrow{3} 6$ $min(11:8)$ 00 : 55 : 40	✗ $1180 \xrightarrow{4} 7$ $min(12:9)$ 00 : 45 : 13	✗ $1180 \xrightarrow{3} 7$ – 00 : 31 : 48	✗ $1180 \xrightarrow{4} 7$ $min(13:10)$ 00 : 32 : 32	✗ $1180 \xrightarrow{4} 6$ $min(10:8)$ 04 : 00 : 45	✗ $1180 \xrightarrow{3} 6$ $min(10:7)$ 01 : 07 : 22	✗ $1180 \xrightarrow{2} 7$ $min(10:7)$ 00 : 16 : 25	✗ $1180 \xrightarrow{3} 7$ $min(13:9)$ 00 : 12 : 30	✗ $1180 \xrightarrow{3} 7$ $min(13:10)$ 00 : 19 : 46	✓ $1180 \xrightarrow{4} 5$ $min(12:9)$ 00 : 49 : 23
Random	✓ $9539 \xrightarrow{2} 5$ $min(10:5)$ 03 : 34 : 56	(✓) $9490 \xrightarrow{3} 5$ $min(10:6)$ 26 : 47 : 00	✓ $9584 \xrightarrow{2} 5$ $min(13:5)$ 19 : 18 : 59	✓ $9499 \xrightarrow{2} 5$ $min(9:5)$ 09 : 37 : 45	✓ $9898 \xrightarrow{2} 5$ $min(9:5)$ 04 : 19 : 18	✗ $9802 \xrightarrow{2} 7$ $min(11:7)$ 34 : 51 : 04	✓ $9445 \xrightarrow{3} 5$ $min(11:6)$ 09 : 22 : 20	✓ $9474 \xrightarrow{3} 5$ $min(7:5)$ 03 : 23 : 54	– $9392 \xrightarrow{3} n.a.$ $min(7:6)$ 09 : 47 : 33	✓ $9514 \xrightarrow{4} 5$ $min(10:7)$ 06 : 10 : 26
nRF52832										
AAL	✓ $896 \xrightarrow{3} 5$ $min(8:7)$ 00 : 05 : 22	✓ $896 \xrightarrow{2} 5$ $min(7:5)$ 00 : 04 : 30	✓ $896 \xrightarrow{3} 5$ $min(8:6)$ 00 : 11 : 48	✓ $896 \xrightarrow{3} 5$ $min(7:6)$ 00 : 15 : 15	✓ $896 \xrightarrow{2} 5$ $min(5:5)$ 00 : 04 : 57	✓ $896 \xrightarrow{2} 5$ $min(7:5)$ 00 : 04 : 52	✓ $896 \xrightarrow{3} 5$ $min(8:6)$ 00 : 03 : 06	✓ $896 \xrightarrow{3} 5$ $min(8:7)$ 00 : 03 : 51	✓ $896 \xrightarrow{3} 5$ $min(8:6)$ 00 : 04 : 26	✓ $896 \xrightarrow{3} 5$ $min(9:7)$ 00 : 03 : 21
Random	✓ $14357 \xrightarrow{2} 5$ $min(6:5)$ 02 : 59 : 50	✓ $14118 \xrightarrow{3} 5$ $min(7:6)$ 03 : 35 : 51	✓ $13988 \xrightarrow{2} 5$ $min(7:5)$ 02 : 55 : 46	✓ $14442 \xrightarrow{2} 5$ $min(7:5)$ 02 : 31 : 46	✓ $14265 \xrightarrow{2} 5$ $min(7:5)$ 07 : 52 : 21	✓ $14118 \xrightarrow{2} 5$ – 02 : 46 : 51	✓ $13112 \xrightarrow{2} 5$ $min(8:5)$ 09 : 42 : 00	✓ $13877 \xrightarrow{2} 5$ $min(6:5)$ 02 : 15 : 25	✓ $14372 \xrightarrow{2} 5$ $min(6:5)$ 01 : 40 : 32	✓ $14442 \xrightarrow{2} 5$ $min(7:5)$ 03 : 21 : 12

Table 10 Experiments individual results: minimal RNN automata learning of CC2650 device from AAL data with **strategy = exhaustive** (#epochs=200)

Use case	Exp11	Exp12	Exp13	Exp14	Exp15	Exp16	Exp17	Exp18	Exp19	Exp20
CC2650										
AAL	✓ $1180 \xrightarrow{2} 5$ $min(11:5)$ 02 : 44 : 25	✓ $1180 \xrightarrow{3} 5$ $min(10:6)$ 04 : 13 : 15	✓ $1180 \xrightarrow{3} 5$ $min(10:6)$ 02 : 42 : 56	✓ $1180 \xrightarrow{3} 5$ $min(10:6)$ 04 : 20 : 44	✓ $1180 \xrightarrow{3} 5$ $min(11:6)$ 04 : 10 : 53	✓ $1180 \xrightarrow{3} 5$ $min(9:6)$ 04 : 00 : 26	✗ $1180 \xrightarrow{2} 6$ $min(10:6)$ 03 : 05 : 01	– $1180 \xrightarrow{3} n.a.$ $min(9:6)$ 05 : 07 : 04	– $1180 \xrightarrow{3} n.a.$ $min(12:6)$ 05 : 02 : 41	✓ $1180 \xrightarrow{3} 5$ $min(11:6)$ 04 : 09 : 11

Table 11 Experiments summary: minimal RNN automata learning of BLE devices with **strategy = best-effort** (#epochs = 200)

Use case	Autom. learned (%)	Learned autom. is min (%)	Minim. boost (%)	Initial state vector size {avg; std}	#Iterations {avg;std}	Non-min. autom. #states {avg; std}	Learning time {avg; std}
CYBLE-416045-02							
AAL	100	100	100	{547; 0}	{2.4; 0.5}	n.a	{00:01:30; 00:00:31}
Random	90	100	20	{946; 57}	{2.6; 0.7}	n.a	{00:24:30; 00:36:35}
CC2650							
AAL	100	10	90	{1180; 0}	{3.3; 0.7}	{6.7; 0.5}	{00:57:08; 01:06:56}
Random	90	89	100	{9564; 161}	{2.6; 0.7}	{7; 0}	{13:02:51; 11:22:21}
nRF52832							
AAL	100	100	100	{896; 0}	{2.7; 0.5}	n.a	{00:06:09; 00:04:02}
Random	100	100	90	{14109; 400}	{2.1; 0.3}	n.a	{03:58:09; 02:37:52}

Table 12 Experiments summary: minimal RNN automata learning of CC2650 device from AAL data with **strategy = exhaustive** (#epochs = 200)

Use case	Autom. learned (%)	Learned autom. is min (%)	Minim. boost (%)	Initial state vector size {avg; std}	#Iterations {avg; std}	Non-min. autom. #states {avg; std}	Learning time {avg; std}
CC2650							
AAL	80	88	100	{1180; 0}	{2.8; 0.4}	{6; 0}	{03:40:51; 00:42:21}

the coverage of the ground-truth automaton was not fully ensured by the randomly generated training data. Increasing the dataset with more randomly generated episodes could be beneficial to solve this issue.

For the CC2650 device, an automaton has been learned which is not minimal, indicated by ✗. Note that the learned automaton still conforms the provided dataset. Learning a non-minimal automaton was more frequently experienced with AAL data. We explain this phenomenon by the difference in the concepts of the used AAL algorithm and our learning technique. The L^* algorithm starts incrementally increasing the automaton size and stops when no further counterexample to the conformance between the learned automaton and SUL can be found. In contrast, our algorithm incrementally decreases the maximum allowed number of states of the automaton. Let S be the AAL dataset, M_A be the respective automaton learned by L^*, and M_R be the automaton learned by our RNN-based technique, then we experience the following relation for the CC2650 device: $S \subset L(M_A)$ but also $S \subset L(M_R)$. Since we assume that M_A correctly represents the SUL, our dataset S misses a trace that shows that $L(M_A) \neq L(M_R)$. This would be an issue in a generic application scenario where we do not know the ground truth automaton since we would learn a non-minimal automaton without knowing it is not minimal. In practice, we can circumvent this problem by testing conformance between the learned model and the SUL following a conformance testing approach as in active learning. Moreover, employing the more expensive exhaustive strategy improves significantly the probability to fix the non-minimality issue, as we can see by comparing Tables 9 and 10. In fact, with the exhaustive strategy, only in 1 out of 8 approved experiments the learned automaton was not minimal. However, it is worth noting that, whenever a non-minimal automaton was learned, the automaton size was not very different from the minimal automaton size, e.g., 6 or 7 states of the non-minimal automaton versus 5 states of the minimal automaton.

Compared to Tomita grammars, we find that the upper bounds estimated with the IOPTA calculation are much larger. This is due to the larger training datasets in the BLE devices, especially CC2650 and nRF52832 devices. Nevertheless, the fixpoint convergence is still reached in a low number of iterations, mostly 2 or 3 iterations. This is because our RNN architecture, thanks to the way how the regularization was designed, is capable of learning a number of automaton states which is close to the minimal automaton states number even if the size of the state vector $s^{<t>}$ would allow to learn many more states. However, even if having such large initial size values of the state vector $s^{<t>}$ does not affect much the functional capability of learning the minimal automaton, this is slowing down the training procedure since the RNN size increases considerably. Finding a better approximation than IOPTA for computing lower upper

bounds for the initial size of the state vector $s^{<t>}$, or simply making a good guess and trying with a much lower initial size, could significantly speed up the learning of the minimal automaton. For example, setting the initial size to 100 would represent a significant reduction (of up to 2 orders of magnitude) compared to IOPTA computation, yet it remains reasonable as it is much higher than the actual minimal number of states. Specifically, for the nRF52832 device, we found that the learning time of the minimal automaton could be reduced by approximately 21% from 00 h:05 m:22 s to 00 h:04 m:15 s with the AAL data and by around 94% from 02 h:59 m:50 s to only 00 h:11 m:03 s with the random data. However, in this paper, we adhere to IOPTA instead of defining ad-hoc fixed upper bounds since IOPTA provides a theoretical guarantee for the upper bound of the minimal number of states.

In summary, whether Algorithm 4 learns a minimal automaton equivalent to the ground truth depends on the training data quality and the ground truth automaton complexity in terms of number of states and transitions and input/output alphabet size. Unapproved experiments are not critical since they are directly signalized by the algorithm and can be mitigated by increasing the epochs budget. If a higher guarantee for automaton minimality is required, the more expensive exhaustive strategy should be used.

4.3.3 Effects of RNN hyperparameter changes

In the following, we illustrate how varying hyperparameter values can influence the results of our RNN-based learning technique.

The hyperparameter values provided in Tables 3 and 4 were determined through a tuning process until we reached a point where we were satisfied with the results. When dealing with AAL data, we explored different hyperparameter values until we could learn the minimal automaton in a reasonable timeframe. For random data, we additionally experimented with varying the size of the dataset, prioritizing learning from smaller datasets. The hyperparameter tuning process was repeated for each dataset size, starting from a size of 10 and gradually increasing it until the automaton could be successfully learned. This explains why we obtained different hyperparameter values for random data compared to AAL data in Table 3 for the Tomita 6 and Tomita 7 grammars. To discuss the influence of the hyperparameters, we examine how learning results change when altering the number of hidden layers and the activation function for Tomita 6 and Tomita 7, respectively. Tables 13, 14 and 15 illustrate the results with the new RNN architecture compared to the initial RNN architecture, utilizing the same seeds to ensure consistent non-deterministic starting conditions.

Tomita 6. In Table 3 for Tomita 6 with random data, we could successfully learn the minimal automaton from a ran-

Table 13 Change of RNN hyperparameters for Tomita 6 and Tomita 7

Grammar	AAL data			Random data			
	Size	Episode lengths {m; std}	RNN {af; #hl} {#e; t}	Size	AAL data coverage	Episode lengths {m; std}	RNN {af; #hl} {#e; t}
Tomita 6	68	{8.6; 4.6}	{relu; ~~12~~} ~~{5; 5 s}~~ {6; 9 s}	20	~~6~~/68	{5.5; 2.5}	{relu; 2} {12; 3 s}
Tomita 7	115	{10.4; 5.1}	{relu; 1} {21; 44 s}	~~50~~80	~~10~~18/115	{6.5; 2.3}	{~~tanh~~relu; 1} ~~{31; 21 s}~~ {37; 42 s}

Comparison with Table 3

Table 14 Change of RNN hyperparameters for Tomita 6 and Tomita 7

Use case	Exp01	Exp02	Exp03	Exp04	Exp05	Exp06	Exp07	Exp08	Exp09	Exp10
Tomita 6										
AAL	✓ – 308 $\xrightarrow{21}$ 3 *n.a.* ~~min(5:3)~~ *n.a.* 00 : ~~06:19~~22 : 31	✓ 308 $\xrightarrow{2}$ 3 ~~min(4:3)~~ – 00 : ~~01:15~~2 : 54	✓ 308 $\xrightarrow{2}$ 3 *min*(~~6~~7 : 3) 00 : ~~02:35~~5 : 24	✓ 308 $\xrightarrow{2}$ 3 *min*(6 : 3) 00 : ~~02:11~~11 : 37	✓ – 308 $\xrightarrow{21}$ 3 *n.a.* ~~min(5:3)~~ *n.a.* 00 : ~~01:50~~22 : 24	✓ 308 $\xrightarrow{2}$ 3 *min*(~~5~~4 : 3) 00 : ~~00:28~~6 : 40	✓ 308 $\xrightarrow{2}$ 3 *min*(~~7~~5 : 3) 00 : ~~03:38~~1 : 43	✓ 308 $\xrightarrow{2}$ 3 *min*(~~5~~4 : 3) 00 : 00 : ~~23~~21	✓ 308 $\xrightarrow{2}$ 3 ~~min(6:3)~~ – 00 : 00 : ~~24~~26	✓ 308 $\xrightarrow{2}$ 3 *min*(~~5~~4 : 3) 00 : ~~01:30~~13 : 54
Tomita 7										
Random	✓ ~~121~~172 $\xrightarrow{2}$ 5 – 00 : ~~07:38~~20 : 36	–3 ~~140~~216 $\xrightarrow{2}$ ~~n.a.~~5 – 00 : ~~16:57~~13 : 41	✓ ~~145~~199 $\xrightarrow{2}$ 5 ~~min(6:5)~~ – 00 : ~~07:23~~13 : 55	✓ ~~133~~195 $\xrightarrow{2}$ 5 – 00 : 03:~~13~~2 : 26	✓ ~~132~~199 $\xrightarrow{2}$ 5 – 00 : 00:~~45~~4 : 15	✓ ~~133~~194 $\xrightarrow{2}$ 5 – 00 : 02:~~04~~3 : 31	✓ ~~139~~185 $\xrightarrow{2}$ 5 ~~min(6:5)~~ – 00 : 02:~~18~~0 : 50	–3 ~~129~~176 $\xrightarrow{2}$ *n.a.* ~~min(6:5)~~ – 00 : ~~07:09~~11 : 08	✓ ~~139~~196 $\xrightarrow{2}$ 5 ~~min(6:5)~~ – 00 : 03 : ~~52~~17	✓ – ~~141~~192 $\xrightarrow{2}$ ~~5~~*n.a.* – 00 : 03:~~32~~16 : 53

Comparison with Table 5

Table 15 Change of RNN hyperparameters for Tomita 6 and Tomita 7

Use case	Autom. learned (%)	Learned autom. is min. (%)	Minim. boost (%)	Initial state vector size {avg; std}	#Iterations {avg; std}	Non-min. Autom. #states {avg; std}	Learning time {avg; std}
Tomita 6							
AAL	~~100%~~80%	100%	~~100%~~60%	{308;0}	{2;0}	n.a	~~{00:02:03;00:01:50}~~{00:05:22;00:05:06}
Tomita 7							
Random	~~80%~~90%	100%	~~40%~~0%	~~{135; 7}~~{192; 13}	{2; 0}	n.a	~~{00:03:51; 00:02:28}~~{00:08:12; 00:06:49}

Comparison with Table 7

dom dataset as small as containing 20 traces when increasing the number of hidden layers from 1 to 2. In contrast, with a single hidden layer, as used for AAL data, a random dataset including 50 traces would have been needed to learn the minimal automaton. In the following, we will therefore check if learning with two hidden layers also works for AAL data.

As can be seen in Table 13, when the minimum number of states is given, we could still successfully learn the minimal AAL automaton but the execution time almost doubles due to the additional hidden layer, while the number of training epochs remains almost constant. Learning the minimum number of states increases significantly for most experiments the learning time (see Table 15), especially for two experiments where convergence could not be reached (see Table 14). However, the RNN's performance improved independently, where FSM minimization helped only in 60% of the experiments, as opposed to 100% with the initial RNN architecture. This improvement can be attributed to the increased capacity of the RNN obtained through the additional hidden layer. Although there were two experiments where no convergence was reached at the first iteration, the overall convergence success rate of 80% remained satisfactory, and there was no need to repeat the experiments with an increased budget.

Tomita 7. In Table 3 for Tomita 7 with random data and the minimum states number given, we successfully learned the minimal automaton from a random dataset as small as including 50 traces when switching from the *ReLU* activation function, which first worked with the AAL data, to *tanh* (see Table 3). Attempting to learn the automaton from the AAL data with the *tanh* activation function did not complete within a set time budget of 10 min, which was a magnitude order higher than the learning time achieved with the *ReLU* activation function. In the following, we analyze the effects on learning the minimal automaton from random data when using the *ReLU* activation function, as with the AAL data, instead of *tanh*. Table 13 reveals an increase in the required random dataset size from 50 to 80 traces, which, in turn, explains the increase in AAL data coverage and learning time. When the minimal number of states is also learned (see Tables 14 and 15), there is an improvement in the convergence rate (90% instead of 80%), see Table 15, and in the RNN performance compared to external FSM minimization (0% instead of 40%), attributable to the larger training dataset. The larger training dataset is also responsible for the increased average learning time.

4.3.4 Comparison to a classic learning algorithm

We compared our RNN-based learning technique with a conventional passive learning algorithm from the literature. We chose a variant of the Regular Positive Negative Inference (RPNI) algorithm [4, 18] for learning Mealy machines. RPNI is a passive learning algorithm that is supported by many modern learning libraries such as LearnLib [7] or AALpy [6]. The RPNI variant starts by building an IOPTA from the given dataset, similar to our Algorithm 4. Then, the algorithm merges states based on the given dataset until no more merges are possible. Two states are merged if their future input/output behavior is equivalent. Originally, RPNI requires positive and negative episodes, where positive episodes are included in the language to be learned and negative ones are not. Negative traces are not required for learning Mealy machines, as the states can be distinguished due to the different output behavior for the same input sequences. For consistency with our active-learning-based data generation, we use the implementation of RPNI in AALpy for the performed comparison. Note that the chosen implementation merges states deterministically.

Table 16 presents the results of performing RPNI on the same randomly generated datasets as used for the RNN-based experiments in Tables 5 and 9. The numbers of episodes correspond to the numbers given in Tables 3 and 4. For comparison, we measure the conformance between the ground-truth automaton and the learned automaton checking bi-similarity. The conformance metric provides the average percentage of the number of common edges over the union of the edges from both automata. In addition, we report the average number of states of the learned automata, the percentage of automata that have the same number of states as the minimal ground-truth automaton, and the required execution time of the learning algorithm.

Remark. Owing to space constraints, the conformance metric values for the RNN-based experiments were not included in the tables presented in the earlier sections. To facilitate a more comprehensive comparison with the RPNI results showcased in Table 16, we now provide, in Table 17, the average and standard deviation of conformance values per use case. These values correspond to experiments conducted with the random data as outlined in 5 and 9. It is important to note that experiments where the convergence point was not reached, indicated by —, have been excluded from consideration.

The RPNI results indicate differences between the Tomita and the BLE case studies. For the Tomita case study, we find that RPNI achieves quite high conformance scores, with only a few exceptions where a slightly larger automaton is learned. This indicates that the generated random datasets sufficiently cover the behavior of the underlying Tomita case study. In contrast, the results for the BLE case study are different. The automata learned with RPNI achieve significantly lower conformance with the ground-truth automata. Moreover, RPNI failed to learn the minimal automaton in the majority of experiments. These results emphasize that randomly generated datasets might not be complete, meaning

Table 16 RPNI automata learning given the same random datasets used in the RNN-based experiments presented in Tables 5 and 9

Use case	#Episodes	Conformance (%) {avg; std}	#States {avg; std}	Learned autom. is min (%)	Time (s) {avg; std}
Tomita 1	10	{92.50; 23.72}	{2.10; 0.32}	90	{0.0008; 0.0002}
Tomita 2	100	{100.00; 0.00}	{3.00; 0.00}	100	{0.0025; 0.0002}
Tomita 3	500	{100.00; 0.00}	{5.00; 0.00}	100	{0.0104; 0.0007}
Tomita 4	50	{94.29; 18.07}	{4.20; 0.63}	90	{0.0015; 0.0002}
Tomita 5	50	{100.00; 0.00}	{4.00; 0.00}	100	{0.0018; 0.0004}
Tomita 6	20	{92.31; 24.32}	{3.20; 0.63}	90	{0.0008; 0.0001}
Tomita 7	50	{91.43; 18.07}	{5.20; 0.42}	80	{0.0019; 0.0002}
CYBLE-416045-02	100	{48.72; 15.11}	{6.40; 1.85}	0	{0.0121; 0.0021}
CC2650	1000	{71.43; 30.83}	{9.00; 5.01}	40	{0.1826; 0.1257}
nRF52832	1500	{84.51; 17.52}	{6.20; 1.87}	50	{0.1987; 0.0490}

Table 17 Conformance results from the RNN-based experiments with the random data presented in Tables 5 and 9

Use case	Tomita 1	Tomita 2	Tomita 3	Tomita 4	Tomita 5	Tomita 6	Tomita 7	CYBLE-416045-02	CC2650	nRF52832
Conformance (%) {avg; std}	{100; 0}	{100; 0}	{100; 0}	{100; 0}	{100; 0}	{100; 0}	{100; 0}	{95.24; 7.90}	{95.61; 11.78}	{100; 0}

that not every input and output is defined for every state. For the BLE case study, sparsity in the dataset is expected due to the larger size of the input and output alphabet. This shows that our RNN-based technique can generalize well enough on incomplete datasets and learns more conforming automata than RPNI. Regarding execution time, we observe that state merging requires less time than training an RNN.

For the case studies considered in this paper, a comparison of Tables 16 with Tables 7 and 11 indicates that our RNN-based learning approach is more favorable than a traditional *passive* automata learning algorithm like RPNI, when utilizing random data or data collected during the regular operation of the SUL. If there is the option of a reliable online interaction with the SUL, it is advisable to opt for a classical *active* automata learning algorithm such as L^* for effectively learning the minimal automaton of the SUL.

4.3.5 Generalizability

We evaluated the feasibility of learning minimal FSMs of relatively small sizes from samples of Tomita regular languages and BLE devices. It remains to be proven whether our results generalize to more complex languages that require larger FSMs. However, our main goal was to propose an architecture that can accurately capture discrete dynamics. The languages from our case studies possess various different features, like modular arithmetic in Tomita 5. Hence, we argue that the architecture generalizes well to different languages of comparable discrete complexity. While interesting, it is not our foremost goal to scale to large discrete systems. Since RNNs are generally well-suited to tasks, like time-series forecasting, we rather plan to extend RNNs with our architecture to be able to additionally model continuous dynamics. Thus, we aim to pave the way toward end-to-end learning of hybrid system models.

5 Related work

In this research, we address the challenge of learning an FSM from a given sample by employing constrained training of an RNN. The literature has presented various techniques for extracting an FSM from sample data, including state-merging-based approaches [4, 18, 32, 33], search-based techniques [34–37], and methods relying on SAT [9–11] and SMT [12, 13] solving. Similar to state-merging methods, we construct an IOPTA, with the distinction that we do not merge its states. The IOPTA solely establishes an upper bound on the maximum number of states of the ultimately learned FSM. Our findings in Sect. 4 demonstrate that our RNN-based method can learn more accurate FSMs for systems with more than two inputs and outputs compared to a well-established state-merging technique [18, 32].

Early work on the relationship between finite automata and neural networks examined the capacity of neural networks to simulate the behavior outlined by a finite automaton. Pioneering this investigation, Kleene [38] was among the first to demonstrate the suitability of neural networks for such simulations. Subsequently, Minsky [39] provided a comprehensive construction of neural networks capable of simulating finite automata. In contrast, we do not simulate a known automaton, but rather we learn an FSM from a sample of a regular language.

The relationship between neural networks and automata has also been exploited to explain the behavior of neural networks, by extracting automata from trained neutral networks. The process of extracting an automaton from a neural network is commonly referred to as *knowledge distillation* [40]. The literature comprises various techniques for knowledge distillation: Giles et al. [41] demonstrated the extraction of deterministic finite automata (DFAs) from a particular type of RNNs trained on regular languages, a method later refined by Omlin and Giles [42] to be applicable to a broader range of RNN architectures. In a more general context, Wang et al. [43] conducted a benchmark study on different RNN architectures for DFA extraction. Furthermore, Wang et al. [44] presented an empirical evaluation, validating the reliability of the approach introduced by Giles et al. [41]. The basis for their approach is that hidden states of RNNs form clusters; thus, automata states can be identified by determining such clusters. This property was recently also used to learn DFAs [26]. Tino and Sajada [45] used self-organizing maps to identify clusters for modeling Mealy automata. Michalenko et al. [46] empirically analyzed this property and found that there is a correspondence between hidden-state clusters and automata states, but some clusters may overlap, i.e., some states may be indistinguishable. Hong et al. [47] and Dong et al. [31] utilized clustering to find an adequate state abstraction and subsequently apply a state-merging algorithm on the abstracted traces to learn DFAs and Markov chains, respectively. The use of these techniques relies on the assumption that the nodes of neural networks form distinct clusters.

In contrast to relying on clustering, which may not be perfect, we enforce a clustering of the hidden states through regularization. Closest to our work in this regard is the work by Oliva and Lago-Fernández [27]. They enforce neurons with sigmoid activation to operate in a binary regime, thus leading to very dense clusters, by introducing Gaussian noise prior to applying the activation function during training. The identified clusters correspond to states of the FSM, where the number of clusters is not necessarily minimal. Similarly to our approach, they apply minimization algorithms to minimize the extracted automaton. However, a key distinction from our technique lies in the fact that the definition of state transitions relies on the inference of clusters in their

approach, whereas our proposed RNN architecture directly predicts the next states.

An additional method for knowledge distillation of neural networks involves treating the trained neural network as an oracle that can be queried. Using an active learning approach, the learning algorithm constructs a behavioral model based on the responses obtained from these queries. Several approaches have been recently proposed based on or related to the L^* algorithm [5]. Weiss et al. proposed automata-learning-based approaches to extract DFAs [26], weighted automata [48], and subsets of context-free grammars from RNNs [49]. Mayr and Yovine [50] applied the L^* algorithm to extract automata from RNNs, where they provide probabilistic guarantees. Khmelnitsky et al. [51] propose a property-directed verification approach for RNNs. They use the L^* algorithm to learn automata models from RNNs and analyze these models through model checking. Muškardin et al. [28] examine the effect of different equivalence-query implementations in L^*-based learning of models from RNNs. Barbot et al. [52] use an A^*-based technique to extract push-down automata from RNNs simulating context-free grammars. In addition to L^*-related approaches, there are also query-based techniques utilizing Hankel matrices [53]: Both, Eyraud and Ayache [54], and Lacroce et al. [55] create weighted automata by populating a Hankel matrix through queries to an RNN. Motivated by the recent popularity of knowledge distillation, the TAYSIR competition [56] has been initiated, with the aim of creating models that provide *simpler* representations of trained RNN and transformers. Muškardin et al. [57] emerged as the winners of the competition, employing a learning-based testing approach following their earlier work [28] and utilizing the automata learning library AALpy [6]. It is important to note that these active techniques focus on extracting an automaton from a trained neural network, whereas our approach involves a constrained training technique for RNNs to extract automata from given samples.

Another approach to extract an FSM from a trained neural network is to additionally train an autoencoder that encodes a neural network as a finite state representation. Koul et al. [58] introduce quantization through training of *quantized bottleneck networks* into RNNs that encode policies of autonomous agents. This allows them to extract FSMs in order to understand the memory usage of recurrent policies. Carr et al. [59] use quantized bottleneck networks to extract finite-state controllers from recurrent policies to enable formal verification.

6 Conclusion

In this work, we presented a new machine learning technique for learning minimal finite-state models in the form of Mealy machines. Our automata learning approach builds upon a specialized RNN architecture together with a constrained training method in order to construct a fixed-size Mealy machine from given training data. Starting from an upper bound on the number of states, the approach iteratively creates models of decreasing size. This iterative process enables learning of minimal models. In common with classical passive automata learning methods, like RPNI [4] and ALERGIA [32], we start from a tree-shaped representation of the training data. However, instead of explicit state merging, we employ RNN training to search for a smaller representation of the data.

We evaluated our method on example grammars from the literature as well as on a Bluetooth protocol implementation. In almost all cases, we were able to learn a minimal automaton representing correctly the ground truth. Where this was not the case, it was due to missing training data. Nevertheless, the learned Mealy machine was correct with respect to the training data. A clear advantage compared to our previous work is that the user does not need to know the number of states in advance.

We see the encouraging results as a step toward learning more complex models comprising discrete and continuous behavior, as found in many control applications. Control applications commonly have only a few modes (discrete states) but may possess complex continuous behavior. For this reason, we focus on small automata in this work. As a next step, we see the integration of continuous behavior into our models as the most promising avenue for future work. That is, we plan to learn the discrete behavior with the regularized training described in this article while learning continuous behavior through conventional RNN training. Having finite-state models of hybrid systems will especially help in the explainability and interoperability of decisions of hybrid system controllers. We leave these investigations for future work. We will also apply our approach to case studies with larger numbers of states. However, other automata learning techniques focused on discrete behavior may be more suitable.

Finally, we dare to express the hope that this work might contribute to bridging the gap between the research communities in machine learning and automata learning ultimately leading to more trustworthy AI systems.

Acknowledgements We thank Dominik Schmidt for his contribution to the conference paper that built the basis for this work. We would like to thank also the anonymous reviewers for their comments on the earlier drafts of this article. This research was partly supported by (i) the LearnTwins project funded by FFG (Österreichische Forschungsförderungsgesellschaft) under grant 880852, (ii) the TU Graz LEAD project Dependable Internet of Things in Adverse Environments project, (iii) the "University SAL Labs" initiative of Silicon Austria Labs (SAL) and its Austrian partner universities for applied fundamental research for electronic based systems, (iv) the AIDOaRt project (grant agreement No 101007350) from the ECSEL Joint Undertaking (JU), where the JU receives support from the European Union's Horizon 2020 research and

innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, and Spain, and (v) the University of Oslo (UiO) sustainability project "Coastal Ecosystems Dynamics Under Anthropogenic Pressures" which is funded by UiO and the Norwegian Ministry of Education and Research.

Funding Open access funding provided by University of Oslo (incl Oslo University Hospital)

References

1. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits, pp. 123–148. Springer, Cham (2018)
2. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Machine Learning for Dynamic Software Analysis: Potentials and Limits—International Dagstuhl Seminar 16172, pp. 74–100. Springer, Cham (2018)
3. Irfan, M.N., Oriat, C., Groz, R.: Model inference and testing. In: Advances in Computers, vol. 89, pp. 89–139. Elsevier (2013)
4. Oncina, J., Garcia, P.: Identifying regular languages in polynomial time. In: Advances in Structural and Syntactic Pattern Recognition. Machine Perception and Artificial Intelligence, vol. 5, pp. 99–108. World Scientific (1992). https://doi.org/10.1142/9789812797919_0007
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
6. Muskardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: an active automata learning library. Innov. Syst. Softw. Eng. **18**(3), 417–426 (2022). https://doi.org/10.1007/s11334-022-00449-3
7. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib—a framework for active automata learning. In: CAV 2015. Part I, pp. 487–495. Springer, Cham (2015)
8. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In: Furbach, U., Shankar, N. (eds.) Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17–20, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4130, pp. 483–497. Springer, Berlin (2006). https://doi.org/10.1007/11814771_40
9. Heule, M., Verwer, S.: Software model synthesis using satisfiability solvers. Empir. Softw. Eng. **18**(4), 825–856 (2013)
10. Avellaneda, F., Petrenko, A.: FSM inference from long traces. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E.P. (eds.) Formal Methods—22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15–17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10951, pp. 93–109. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_6
11. Wallner, F., Aichernig, B.K., Burghard, C.: It's not a feature, it's a bug: fault-tolerant model mining from noisy data. In: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), pp. 316–328. IEEE Computer Society, Los Alamitos, CA, USA (2024). https://doi.ieeecomputersociety.org/
12. Smetsers, R., Fiterau-Brostean, P., Vaandrager, F.W.: Model learning as a satisfiability modulo theories problem. In: LATA 2018. Ramat Gan, Israel, April 9–11, 2018, Proceedings, pp. 182–194. Springer, Cham (2018)
13. Tappler, M., Aichernig, B.K., Lorber, F.: Timed automata learning via SMT solving. In: NFM 2022. Pasadena, CA, USA, May 24–27, 2022, Proceedings, pp. 489–507. Springer, Cham (2022)
14. Gold, E.M.: Complexity of automaton identification from given data. Inf. Control **37**(3), 302–320 (1978). https://doi.org/10.1016/S0019-9958(78)90562-4
15. Aichernig, B.K., König, S., Mateis, C., Pferscher, A., Schmidt, D., Tappler, M.: Constrained training of recurrent neural networks for automata learning. In: Schlingloff, B., Chai, M. (eds.) Software Engineering and Formal Methods—20th International Conference, SEFM 2022, Berlin, Germany, September 26–30, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13550, pp. 155–172. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17108-6_10
16. Elman, J.L.: Finding structure in time. Cognit. Sci. **14**(2), 179–211 (1990). https://doi.org/10.1207/s15516709cog1402_1
17. Ma, Y., Principe, J.C.: A taxonomy for neural memory networks. IEEE Trans. Neural Netw. Learn. Syst. **31**(6), 1780–1793 (2020). https://doi.org/10.1109/TNNLS.2019.2926466
18. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York (2010)
19. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning Markov decision processes for model checking. In: Fahrenberg, U., Legay, A., Thrane, C.R. (eds.) Proceedings Quantities in Formal Methods, QFM 2012, Paris, France, 28 August 2012. EPTCS, vol. 103, pp. 49–63 (2012). https://doi.org/10.4204/EPTCS.103.6
20. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. In: Pearson International Edition, 3rd edn. Addison-Wesley, USA (2007)
21. Hopcroft, J.: An n log n algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press, (1971). https://doi.org/10.1016/B978-0-12-417750-5.50022-1
22. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: FM. LNCS, vol. 5850, pp. 207–222. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-05089-3_14
23. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: an imperative style, high-performance deep learning library. In: NeurIPS, pp. 8024–8035. Curran Associates Inc. (2019)
24. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: ICLR (2015)
25. Tomita, M.: Dynamic construction of finite automata from examples using hill-climbing. In: Conference of the Cognitive Science Society, pp. 105–108 (1982)
26. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. In: ICML. Proceedings of Machine Learning Research, vol. 80, pp. 5244–5253. PMLR (2018)
27. Oliva, C., Lago-Fernández, L.F.: Stability of internal states in recurrent neural networks trained on regular languages. Neurocom-

puting **452**, 212–223 (2021). https://doi.org/10.1016/j.neucom.2021.04.058
28. Muskardin, E., Aichernig, B.K., Pill, I., Tappler, M.: Learning finite state models from recurrent neural networks. In: IFM. LNCS, vol. 13274, pp. 229–248. Springer, Cham (2022)
29. Goudreau, M.W., Giles, C.L., Chakradhar, S.T., Chen, D.: First-order versus second-order single-layer recurrent neural networks. IEEE Trans. Neural Netw. **5**(3), 511–513 (1994). https://doi.org/10.1109/72.286928
30. Pferscher, A., Aichernig, B.K.: Fingerprinting Bluetooth Low Energy devices via active automata learning. In: FM. LNCS, vol. 13047, pp. 524–542. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_28
31. Dong, G., Wang, J., Sun, J., Zhang, Y., Wang, X., Dai, T., Dong, J.S., Wang, X.: Towards interpreting recurrent neural networks through probabilistic abstraction. In: ASE, pp. 499–510. IEEE (2020). https://doi.org/10.1145/3324884.3416592
32. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Carrasco, R.C., Oncina, J. (eds.) Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21–23, 1994, Proceedings. Lecture Notes in Computer Science, vol. 862, pp. 139–152. Springer, Berlin (1994). https://doi.org/10.1007/3-540-58473-0_144
33. Verwer, S., Hammerschmidt, C.A.: flexfringe: a passive automaton learning package. In: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17–22, 2017, pp. 638–642. IEEE Computer Society (2017). https://doi.org/10.1109/ICSME.2017.58
34. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A. (eds.) Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11–13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4445, pp. 114–124. Springer (2007). https://doi.org/10.1007/978-3-540-71605-1_11
35. Lai, Z., Cheung, S.C., Jiang, Y.: Dynamic model learning using genetic algorithm under adaptive model checking framework. In: Sixth International Conference on Quality Software (QSIC 2006), 26–28 October 2006, Beijing, China, pp. 410–417. IEEE Computer Society (2006). https://doi.org/10.1109/QSIC.2006.25
36. Lefticaru, R., Ipate, F., Tudose, C.: Automated model design using genetic algorithms and model checking. In: Kefalas, P., Stamatis, D., Douligeris, C. (eds.) 2009 Fourth Balkan Conference in Informatics, BCI 2009, Thessaloniki, Greece, 17–19 September 2009, pp. 79–84. IEEE Computer Society (2009). https://doi.org/10.1109/BCI.2009.15
37. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn - learning timed automata from tests. In: André, É., Stoelinga, M. (eds.) Formal Modeling and Analysis of Timed Systems—17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27–29, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11750, pp. 216–235. Springer (2019). 10.1007/978-3-030-29662-9_13
38. Kleene, S.C.: Representation of Events in Nerve Nets and Finite Automata. RAND Corporation, Santa Monica (1951)
39. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Inc, Hoboken (1967)
40. Hinton, G.E., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. In: CoRR (2015). arXiv:1503.02531
41. Giles, C.L., Miller, C.B., Chen, D., Chen, H., Sun, G., Lee, Y.: Learning and extracting finite state automata with second-order recurrent neural networks. Neural Comput. **4**(3), 393–405 (1992). https://doi.org/10.1162/NECO.1992.4.3.393
42. Omlin, C.W., Giles, C.L.: Extraction of rules from discrete-time recurrent neural networks. Neural Netw. **9**(1), 41–52 (1996). https://doi.org/10.1016/0893-6080(95)00086-0
43. Wang, Q., Zhang, K., Liu, X., Giles, C.L.: Verification of recurrent neural networks through rule extraction. In: CoRR. (2018) arXiv:1811.06029
44. Wang, Q., Zhang, K., II., A.G.O., Xing, X., Liu, X., Giles, C.L.: An empirical evaluation of rule extraction from recurrent neural networks. Neural Comput. **30**(9), 2568–2591 (2018). https://doi.org/10.1162/NECO_A_01111
45. Tino, P., Sajda, J.: Learning and extracting initial Mealy automata with a modular neural network model. Neural Comput. **7**(4), 822–844 (1995). https://doi.org/10.1162/neco.1995.7.4.822
46. Michalenko, J.J., Shah, A., Verma, A., Baraniuk, R.G., Chaudhuri, S., Patel, A.B.: Representing formal languages: a comparison between finite automata and recurrent neural networks. In: ICLR. OpenReview.net (2019)
47. Hong, D., Segre, A.M., Wang, T.: Adaax: explaining recurrent neural networks by learning automata with adaptive states. In: Zhang, A., Rangwala, H. (eds.) KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14–18, 2022, pp. 574–584. ACM (2022). https://doi.org/10.1145/3534678.3539356
48. Weiss, G., Goldberg, Y., Yahav, E.: Learning deterministic weighted automata with queries and counterexamples. In: NeurIPS, pp. 8558–8569 (2019)
49. Yellin, D.M., Weiss, G.: Synthesizing context-free grammars from recurrent neural networks. In: TACAS. LNCS, vol. 12651, pp. 351–369. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72016-2_19
50. Mayr, F., Yovine, S.: Regular inference on artificial neural networks. In: CD-MAKE. LNCS, vol. 11015, pp. 350–369. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99740-7_25
51. Khmelnitsky, I., Neider, D., Roy, R., Xie, X., Barbot, B., Bollig, B., Finkel, A., Haddad, S., Leucker, M., Ye, L.: Property-directed verification and robustness certification of recurrent neural networks. In: ATVA. LNCS, vol. 12971, pp. 364–380. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_24
52. Barbot, B., Bollig, B., Finkel, A., Haddad, S., Khmelnitsky, I., Leucker, M., Neider, D., Roy, R., Ye, L.: Extracting context-free grammars from recurrent neural networks using tree-automata learning and A* search. In: Chandlee, J., Eyraud, R., Heinz, J., Jardine, A., van Zaanen, M. (eds.) Proceedings of the 15th International Conference on Grammatical Inference, 23–27 August 2021, Virtual Event. Proceedings of Machine Learning Research, vol. 153, pp. 113–129. PMLR (2021). https://proceedings.mlr.press/v153/barbot21a.html
53. Berstel, J., Reutenauer, C.: Rational series and their languages. In: EATCS Monographs on Theoretical Computer Science, vol. 12. Springer (1988). https://www.worldcat.org/oclc/17841475
54. Eyraud, R., Ayache, S.: Distillation of weighted automata from recurrent neural networks using a spectral approach. In: CoRR (2020). arXiv:2009.13101
55. Lacroce, C., Panangaden, P., Rabusseau, G.: Extracting weighted automata for approximate minimization in language modelling. In: Chandlee, J., Eyraud, R., Heinz, J., Jardine, A., van Zaanen, M. (eds.) Proceedings of the 15th International Conference on Grammatical Inference, 23–27 August 2021, Virtual Event. Proceedings of Machine Learning Research, vol. 153, pp. 92–112. PMLR (2021). https://proceedings.mlr.press/v153/lacroce21a.html
56. Eyraud, R., Lambert, D., Joutei, B.T., Gaffarov, A., Cabanne, M., Heinz, J., Shibata, C.: TAYSIR competition: transformer+RNN: algorithms to yield simple and interpretable representations. In: Coste, F., Ouardi, F., Rabusseau, G. (eds.) International Conference on Grammatical Inference, ICGI 2023, 10–13 July 2023, Rabat, Morocco. Proceedings of Machine Learning Research, vol. 217,

pp. 275–290. PMLR (2023). https://proceedings.mlr.press/v217/eyraud23a.html
57. Muskardin, E., Tappler, M., Aichernig, B.K.: Testing-based black-box extraction of simple models from RNNs and transformers. In: Coste, F., Ouardi, F., Rabusseau, G. (eds.) International Conference on Grammatical Inference, ICGI 2023, 10–13 July 2023, Rabat, Morocco. Proceedings of Machine Learning Research, vol. 217, pp. 291–294. PMLR (2023). https://proceedings.mlr.press/v217/muskardin23a.html
58. Koul, A., Fern, A., Greydanus, S.: Learning finite state representations of recurrent policy networks. In: ICLR. OpenReview.net (2019)
59. Carr, S., Jansen, N., Topcu, U.: Verifiable RNN-based policies for POMDPs under temporal logic constraints. In: IJCAI, pp. 4121–4127. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/570

Bernhard K. Aichernig is a tenured associate professor (ao. Univ. -Prof.) at Graz University of Technology, Austria. With his research group, he investigates the foundations of software engineering for realizing dependable computer-based systems. Bernhard is an expert in formal methods and testing. His research covers a variety of areas combining falsification, verification, and abstraction techniques. Current topics include the Internet of Things, test-case generation, automata learning, and statistical model checking. Since 2020, he leads the TU Graz-SAL Dependable Embedded Systems Lab (DES Lab) with fundamental research in zero-bug software and dependable AI. He participated in five European projects. From 2004 to 2016, Bernhard served as a board member of Formal Methods Europe, the association that organizes the Formal Methods symposia. From 2002 to 2006, he had a faculty position at the United Nations University in Macao S.A.R., China. Bernhard holds a habilitation in Practical Computer Science and Formal Methods, a doctorate, and a diploma engineer degree from Graz University of Technology.

Sandra König is a researcher at the Austrian Institute of Technology in the Center for Digital Safety & Security since 2014. She received her bachelor's and master's degree in mathematics at ETH Zurich and her PhD with distinction in Mathematics at Alpen Adria University Klagenfurt. Her core focus lies on probabilistic risk models and game theoretic risk mitigation methods. She has been involved in several Austrian and EU projects dealing with protection of critical infrastructures as well as safety and security of autonomous systems. Beyond this, she is interested in application of machine learning techniques and domains such as logistics. She is a lecturer at the University of Zurich.

Cristinel Mateis is a machine learning professional affiliated with AIT Austrian Institute of Technology GmbH in Vienna. His focus lies in pioneering solutions for intricate business and technical challenges through the application of cutting-edge machine learning techniques, drawing upon his extensive industry experience and research expertise in artificial intelligence. Cristinel holds a master's degree in Computer Engineering from the University of Calabria and a Ph.D. in Knowledge Representation and Reasoning from Vienna University of Technology.

Andrea Pferscher is a postdoctoral researcher at the Department of Informatics at the University of Oslo. She received her PhD from Graz University of Technology in 2023. In her PhD thesis, she investigated automata learning techniques for analyzing and testing the safety-critical behavior of network components. In her current position, she has broadened her focus to the application of formal methods for digital twins. She has been a member of different Austrian, European and Norwegian research projects.

Martin Tappler is a postdoctoral researcher at the Institute of Software Technology at Graz University of Technology. He obtained a PhD in computer science in 2019 at Graz University of Technology on the topic of automata learning. Subsequently, he worked as a postdoctoral researcher at the Schaffhausen Institute of Technology, Switzerland. His research interests include model-based testing, automata learning, and verification of learning-enabled systems.