Interpreting Finite Automata for Sequential Data

Christian Albert Hammerschmidt
SnT
University of Luxembourg
christian.hammerschmidt@uni.lu

Sicco Verwer
Department of Intelligent Systems
Delft University of Technology
s.e.verwer@tudelft.nl

Qin Lin
Department of Intelligent Systems
Delft University of Technology
q.lin@tudelft.nl

Radu State
SnT
University of Luxembourg
radu.state@uni.lu

Abstract

Automaton models are often seen as *interpretable* models. Interpretability itself is not well defined: it remains unclear what interpretability means without first explicitly specifying objectives or desired attributes. In this paper, we identify the key properties used to interpret automata and propose a modification of a state-merging approach to learn variants of finite state automata. We apply the approach to problems beyond typical grammar inference tasks. Additionally, we cover several use-cases for prediction, classification, and clustering on sequential data in both supervised and unsupervised scenarios to show how the identified key properties are applicable in a wide range of contexts.

1 Introduction

The demand for explainable machine learning is increasing, driven by the spread of machine learning techniques to sensitive domains like cyber-security, medicine, and smart infrastructure among others. Often, the need is abstract but nevertheless a requirement, e.g., in the recent EU regulation [Goodman and Flaxman, 2016].

Approaches to explanations range from post-hoc explanation systems like Turner [2015], which provide explanations of decisions taken by black-box systems to the use of linear or specialized white-box systems [Fiterau et al., 2012] that generate models seen as simple enough for non-expert humans to interpret and understand. Lipton [2016] outlines how different motivations and requirements for interpretations lead different notions of interpretable models in supervised learning.

Automata models, such as (probabilistic) deterministic finite state automata ((P)DFA) and timed automata (RA) have long been studied (Hopcroft et al. [2013]) and are often seen as interpretable models. Moreover, they are learnable from data samples, both in supervised and unsupervised (see Higuera [2010]) fashion. But which properties make these models interpretable, and how can we get the most benefit from them? We argue that—especially in the case of unsupervised learning—automata models have a number of properties that make it easy for humans to understand the learned model and project knowledge into it: a graphical representation, transparent computation, generative nature, and our good understanding of their theory.

2 Preliminaries

Finite automata. The models we consider are variants of deterministic finite state automata, or finite state machines. These have long been key models for the design and analysis of computer systems [Lee and Yannakakis, 1996]. We provide a conceptual introduction here, and refer to Section B in the appendix for details. An automaton consists of a set of states, connected by transitions labeled over an alphabet. It is said to accept a word (string) over the alphabet in a computation if there exists a path of transitions from a predefined start state to one of the predefined final states, using transitions labeled with the letters of the word. Automata are called deterministic when there exists exactly one such path for every possible string. Probabilistic automata include probability values on transitions and compute word probabilities using the product of these values along a path, similar to hidden Markov models (HMMs).

Learning approaches. As learning finite state machines has long been of interest in the field of grammar induction, different approaches ranging from active learning algorithms [Angluin, 1987] to algorithms based on the method of moments [Balle et al., 2014] have been proposed. Process mining [Van Der Aalst et al., 2012] can also be seen as a type of automaton learning, focusing on systems that display a large amount of concurrency, such as business processes, represented as interpretable Petri-nets. We are particularly interested in state-merging approaches, based on Oncina and Garcia [1992]. While Section C in the appendix provides formal details, we provide a conceptual introduction here.

The starting point for state-merging algorithms is the construction of a tree-shaped automaton from the input sample, called augmented prefix tree acceptor (APTA). It contains all sequences from the input sample, with each sequence element as a directed labeled transition. Two samples share a path if they share a prefix. The state-merging algorithm reduces the size of the automaton iteratively by reducing the tree through merging pairs of states in the model, and forcing the result to be deterministic. The choice of the pairs, and the evaluation of a merge is made heuristically: Each possible merge is evaluated and scored, and the highest scoring merge is executed. This process is repeated and stops if no merges with high scores are possible. These merges generalize the model beyond the samples from the training set: the starting prefix tree is already an a-cyclic finite automaton. It has a finite set of computations, accepting all words from the training set. Merges can make the resulting automaton cyclic. Automata with cycles accept an infinite set of words. State-merging algorithms generalize by identifying repetitive patterns in the input sample and creating appropriate cycles via merges. Intuitively, the heuristic tries to accomplish this generalization by identifying pairs of states that have similar future behaviors. In a probabilistic setting, this similarity might be measured by similarity of the empirical probability distributions over the outgoing transition labels. In grammar inference, heuristics rely on occurrence information and the identity of symbols, or use global model selection criteria to calculate merge scores.

3 Flexible State-Merging

The key step in state-merging algorithms is the identification of good merges. A merge of two states is considered good if the possible futures of the two states are very similar. By focusing on application-driven notions of *similarity* of sequences and sequence elements, we modify the state-merging algorithm as follows: For each possible merge, a heuristic can evaluate an application-driven similarity measure to obtain the merge score. Optionally, each symbol of the words is enriched with addition data, e.g. real values. This information can be aggregated in each state, e.g. by averaging. It is used to guide the heuristic in reasoning about the similarity of transitions and therefore the inferred latent states, or guide a model selection criterion, e.g. by using mean-squared-error minimization as an objective function. It effectively separates the symbolic description connecting the latent variables from the objective (function) used to reason about the similarity. An implementation in C++ is available[1]. The importance of combining data with symbolic data is getting renewed attention, c.f. recent works such as Garnelo et al. [2016]. In Section 5, we outline regression automata as a use case of our approach.

---

[1] https://bitbucket.org/chrshmmmr/dfasat

4 Aspects of Interpretability in Automata

To understand how a particular model works as well as how to go beyond the scope of the model and combing foreground knowledge about the application with the model itself, we now focus on which aspects of automata enable interpretations:

1. Automata have an easy *graphical representation* as cyclic, directed, labeled graphs, offering a hierarchical view of sequential data.

2. *Computation is transparent.* Each step of the computation is the same for each symbol w_i of an input sample w. It can be verified manually (e.g. visually), and compared to other computation paths through the latent state space. This makes it possible to analyze training samples and their contribution to the final model.

3. Automata are *generative.* Sampling from the model helps to understand what it describes. Tools like model checkers to query properties in a formal way, e.g. using temporal logic, can help to analyze the properties of the model.

4. Automata are *well studied* in theory and practice, including composition and closure properties, sub-classes and related equally expressive formalisms. This makes it easy for humans to transfer their knowledge onto it: The model is frequently used in system design as a way to describe system logic, and are accessible to a wide audience.

The intention behind using either of these aspects depends on the purpose of the interpretation, e.g. trust, transparency, or generalizing beyond the input sample. Especially for unsupervised learning, we believe that knowledge transfer and exploratory knowledge discovery are common motivations, e.g. in (software) process discovery.

5 Application Case Studies

In the following we present some use cases of automata models and how they are interpreted and how the properties identified in Section 4 contribute to it. While this is by no means an exhaustive literature study, we hope that it helps to illustrate how the different aspects of interpretability are used in practice. In unsupervised learning, the data is observations without labels or counter-examples to the observed events. Often, there is no ground-truth to be used in an objective function. Methods for learning such systems typically use statistical assumptions to compute state similarity.

Software systems. Automata models are often used to infer models of software in an unsupervised fashion, e.g. Walkinshaw et al. [2007]. In these cases, the generative property of automaton models is see as interpretable: It is possible to ask queries using a temporal logic like LTL [Clarke et al., 2005] to answer questions regarding the model, e.g. whether a certain condition will eventually be true, or analyze at what point a computation path deviated from the expected outcome. In Smetsers et al. [2016], the authors use this property to test and validate properties of code by first fuzzing the code to obtain execution paths and the associated inputs and then learn a model to check LTL queries on. Additionally, we can transfer human expert knowledge on system design [Wagner et al., 2006] to inspect the model, e.g. to identify the function of substructures identified. An example can be found in Smeenk et al. [2015], where the authors infer a state machine for a printer via active learning. Through visual inspection alone it is possible to identify deadlocks that are otherwise hard to see in the raw data. The visual analysis helped to identify bugs in the software the developers were unaware of. The appendix shows the final model in Figure 3.

Biological systems. In Schmidt and Kramer [2014], timed automata are used to infer the cell cycle of yeast based on sequences of gene expression activity. The graphical model obtained (c.f. Figure 4) can be visually compared to existing models derived using other methods and combined with a-priori knowledge in biology.

Driver behavior. In our ongoing work using the RTI+ state-merging algorithm for timed automata [Verwer et al., 2008], we analyze car following behavior of human drivers. The task is to relate driver actions like changes of vehicle speed (and thus distance and relative speed to a lead vehicle) to a response action, e.g. acceleration. The inferred automaton model is inspected visually like a software controller. Different driving behaviors are distinguished by clustering the states of the automaton, i.e. the latent state space. The discovered distinct behaviors form control loops within the model. Figure 2 in the appendix shows an example with the discovered clusters highlighted.

Wind speed prediction. In our own work, we applied automata learning in different ways to a problem not related to grammar inference, predicting short-term changes in wind speeds. We take two different approaches to obtain models that tell us more about the data than just a minimizer of the objective function: In one approach, [Pellegrino et al., 2016], we **discover structure** in the data by using by inferring transition guards over a potentially infinite alphabet, effectively discovering a clustering as transition labels from the sequences automatically. The only constraint and objective used here is the similarity of future behaviors. The learned model can be seen as a structure like a decision tree built in a bottom-up fashion, but allowing loops for repetitive patterns. Figure 1 in the appendix shows an example of such an automaton. In another approach [Lin et al., 2016], we use our flexible state-merging framework to **impose structure** through parameter choices. We derive discrete events from the physical wind speed observations by using clustering approaches to obtain a small alphabet of discrete events as a symbolic representation that fits the underlying data well. Using a heuristic that scores merges based on minimizing a mean squared error measure, the automata model has a good objective function for regression, as well as discovers latent variables in terms of the given discretization. In practice, other choices of discretization can be used. By using thresholds of the turbine used in wind mills, e.g. the activation point, one could infer a model whose latent states relate to the physical limitations of the application. We are planning to analyze this approach in future work. As with the previous example, the learned model can be seen as a description of decision paths taken in the latent state-space. If the model makes predictions that are difficult to believe for human experts, the computation and the model prediction can be visually analyzed to see which factors contributed to it, and how the situation relates to similar sets of features.

6   Discussion

The applications surveyed in Section 5 show that *interpreting* finite state automata as models takes many forms and serves different goals. As such, this interpretation is not a feature inherent in the models or the algorithms themselves. Rather, interpretations are defined by the need and intention of the user. But yet, **interpretations of automata models draw from a core set of properties** as identified in Section 4: graphical representations, transparent computation, generative nature, and our understanding of their theory.

We note that **automata models are particularly useful in unsupervised learning:** Applications of automata models often aim at easing the transfer of knowledge about the subject, or related subjects, to the data generating process. In this case, machine learning serves as a tool for exploration, to deal with epistemic uncertainty in observed systems. The goal is not only to obtain a more compact view of the data, but learn how to generalize from the observed data. Often, it is often unclear what the a-priori knowledge is as users rely on experience. This makes it very difficult to formalize a clear objective function. A visual model with a traceable computation helps to guide the users, and helps to iterate over multiple models.

Flexible state-merging allows to obtain automata models in new domains: in our flexible state-merging framework presented in Section 3, we try to separate the symbolic representation from the objective function and heuristic. We hope that this will help to guide discovery by stating the model parameters, e.g. the symbols, independently form the heuristic that guides the discovery of latent variables. In this fashion, it is possible to learn models with interpretable aspects without having to sacrifice the model performance on the intended task.

Future work. We hope that this discussion will help to build a bridge between practitioners and experts in applied fields on one side, and the grammar inference and machine learning community on other side. As probabilistic deterministic automata models are almost as expressive as HMMs, the models and techniques described here are applicable to a wide range of problems with decent performance metrics. We see a lot of promise in combining symbolic data with numeric or other data via a flexible state-merging approach to bring automata learning to fields beyond grammatical inference.

References

Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2): 87–106, 1987. URL `http://dx.doi.org/10.1016/0890-5401(87)90052-6`.

Borja Balle, Xavier Carreras, Franco M. Luque, and Ariadna Quattoni. Spectral Learning of Weighted Automata. *Machine Learning*, 96(1-2):33–63, 2014. URL `http://link.springer.com/article/10.1007/s10994-013-5416-x`.

Edmund Clarke, Ansgar Fehnker, Sumit Kumar Jha, and Helmut Veith. Temporal Logic Model Checking. In *Handbook of Networked and Embedded Control Systems*, Control Engineering, pages 539–558. Birkhäuser Boston, 2005. ISBN 978-0-8176-3239-7 978-0-8176-4404-8. URL `http://link.springer.com/chapter/10.1007/0-8176-4404-0_23`.

Madalina Fiterau, Artur Dubrawski, Jeff Schneider, and Geoff Gordon. Trade-offs in Explanatory Model Learning. 2012. URL `http://www.ml.cmu.edu/research/dap-papers/dap_fiterau.pdf`.

Marta Garnelo, Kai Arulkumaran, and Murray Shanahan. Towards Deep Symbolic Reinforcement Learning. *arXiv:1609.05518 [cs]*, September 2016. URL `http://arxiv.org/abs/1609.05518`. arXiv: 1609.05518.

Bryce Goodman and Seth Flaxman. European Union Regulations on Algorithmic Decision-making and a "Right to Explanation". *arXiv:1606.08813 [cs, stat]*, June 2016. URL `http://arxiv.org/abs/1606.08813`. arXiv: 1606.08813.

Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, April 2010. ISBN 978-0-521-76316-5.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, Harlow, Essex, Pearson New International Edition edition, November 2013. ISBN 978-1-292-03905-3.

D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines-a Survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996. ISSN 0018-9219. doi: 10.1109/5.533956.

Qin Lin, Christian Hammerschmidt, Gaetano Pellegrino, and Sicco Verwer. Short-term Time Series Forecasting with Regression Automata. In *MiLeTS Workshop at KDD 2016*, 2016. URL `http://www-bcf.usc.edu/~liu32/milets16/paper/MiLeTS_2016_paper_17.pdf`.

Zachary C. Lipton. The Mythos of Model Interpretability. *arXiv:1606.03490 [cs, stat]*, June 2016. URL `http://arxiv.org/abs/1606.03490`. arXiv: 1606.03490.

Jose Oncina and Pedro Garcia. Identifying Regular Languages In Polynomial Time. In *Advances in Structural and Syntactic Pattern Recognition*, pages 99–108. World Scientific, 1992.

Gaetano Pellegrino, Christian Albert Hammerschmidt, Qin Lin, and Sicco Verwer. Learning Deterministic Finite Automata from Infinite Alphabets. In *Proceedings of The 13th International Conference on Grammatical Inference*, Delft, October 2016.

Jana Schmidt and Stefan Kramer. Online Induction of Probabilistic Real-Time Automata. *Journal of Computer Science and Technology*, 29(3):345–360, May 2014. ISSN 1000-9000, 1860-4749. doi: 10.1007/s11390-014-1435-8. URL `http://link.springer.com/article/10.1007/s11390-014-1435-8`.

Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N. Jansen. Applying Automata Learning to Embedded Control Software. In *Formal Methods and Software Engineering*, number 9407 in Lecture Notes in Computer Science, pages 67–83. November 2015. URL `http://link.springer.com/chapter/10.1007/978-3-319-25423-4_5`.

Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing Model Learning with Mutation-Based Fuzzing. *arXiv:1611.02429 [cs]*, 2016. URL `http://arxiv.org/abs/1611.02429`. arXiv: 1611.02429.

Ryan Turner. A Model Explanation System. In *Black Box Learning and Inference NIPS Workshop*, 2015. URL `http://www.blackboxworkshop.org/pdf/Turner2015_MES.pdf`.

Wil Van Der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter van den Brand, Ronald Brandtjen, Joos Buijs, and others. Process Mining Manifesto. In *Business Process Management Workshops*, pages 169–194. Springer, 2012. URL `http://link.springer.com/chapter/10.1007/978-3-642-28108-2_19`.

Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. Efficiently Learning Simple Timed Automata. *Induction of Process Models*, pages 61–68, 2008. URL `http://www.cs.ru.nl/~sicco/papers/ipm08.pdf`.

Ferdinand Wagner, Ruedi Schmuki, Thomas Wagner, and Peter Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, May 2006. ISBN 978-1-4200-1364-1.

N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse Engineering State Machines by Interactive Grammar Inference. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.
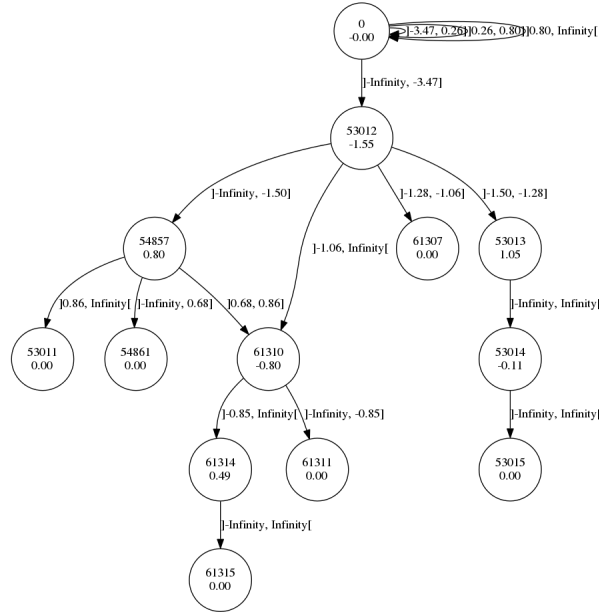
A   Visualizations of Automata

Figure 1: Auto-regressive model of short-term average wind speeds using the approach presented by Lin et al. [2016]. The ranges on the transitions indicate for which speed they activate, the number in the nodes is the predicted change in speed. The top node models persistent behavior (i.e. no change in speed for the next period), whereas the lower part denotes exceptions to this rule.
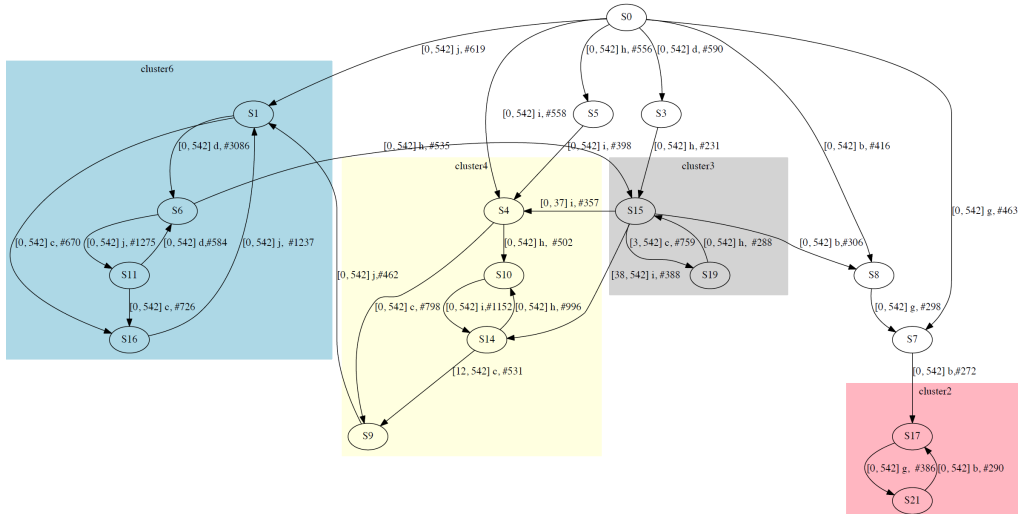

Figure 2: Car-following controller model with highlighted clusters. Each color denotes one distinct cluster. The loop in cluster 6 (colored light blue), e.g. state sequence: 1-6-11-16-1 with symbolic transitions loop: d (keep short distance and slower than lead vehicle)-j (keep short distance and same speed to lead vehicle)-c (keep short distance and faster than lead vehicle))-j, can be interpreted as the **car-following behavior at short distances**, i.e. adapting the speed difference with the lead vehicle around 0 and bounding the relative distance in small zone. Similarly interesting and significant loops can be also seen in cluster 2 (colored pink) and cluster 4 (colored light yellow), which are **long distance** and **intermediate distance car-following** behaviors respectively. The intermediate state like $S15$ in cluster 3 (colored light grey) explains how to switch between clusters.
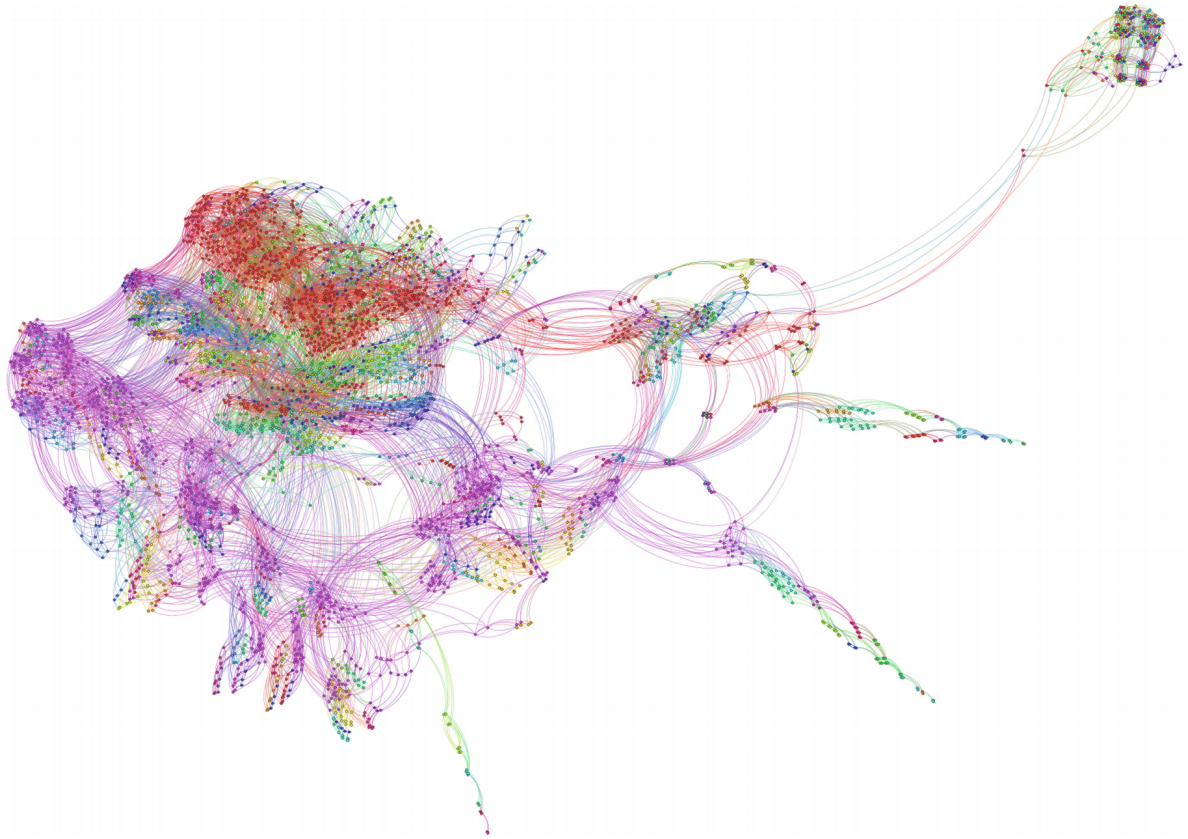
6

Figure 3: Visualization of a state machine learned from a printer controller using an active learning approach. Each state, represented as a dot, represents a state in the controller software and each transition, represened as an edge, a state-change upon receiving input. Using knowledge about the controller design, it is possible to identify the sparsely connected protrusions of states at the bottom as deadlock situations. Taken from Smeenk et al. [2015]
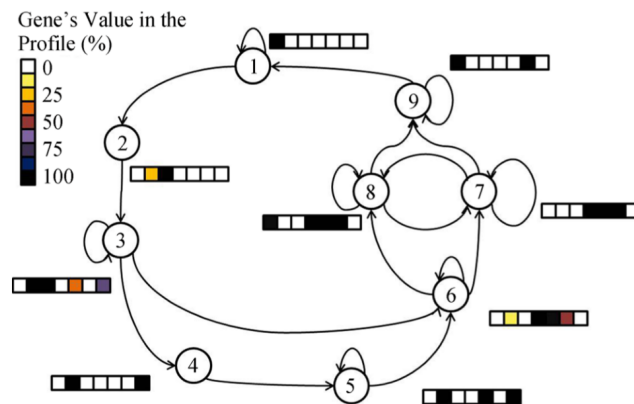.

Figure 4: The yeast cell cycle learned in Schmidt and Kramer [2014] using a passive online learning algorithm for timed automata. The colored bars indicate gene activities. The model corresponds to the well-known yeast cycle.

B Finite State Automata

In this section, we give a formal introduction to deterministic finite state automata as the most basic model considered in the related work. For other variants like probabilistic automata, timed and real time automata, and regression automata, we refer to the cited papers for formal introductions.

A *deterministic finite state automaton* (DFA) is one of the basic and most commonly used finite state machines. Below, we provide a concise description of DFAs, the reader is referred to Hopcroft et al. [2013] for a more elaborate overview. A DFA $A = \langle Q, T, \Sigma, q_0, Q_+ \rangle$ is a directed graph consisting of a set of *states* Q (nodes) and labeled *transitions* T (directed edges). An example is shown in Figure 6. The *start state* $q_0 \in Q$ is a specific state of the DFA and any state can be an *accepting state* (final state) in $Q_+ \subseteq Q$. The labels of transitions are all members of a given *alphabet* Σ. A DFA A can be used to *generate* or *accept* sequences of symbols (strings) using a process called *DFA computation*. This process begins in q_0, and iteratively *activates* (or *fires*) an outgoing transition $t_i = \langle q_{i-1}, q_i, l_i \rangle \in T$ with label $l_i \in \Sigma$ from the *source state* it is in q_{i-1}, moving the process to the *target state* q_i pointed to by t_i. A computation $q_0 t_1 q_1 t_2 q_2 \ldots t_n q_n$ is *accepting* if the state it ends in (its last state) is an accepting state $q_n \in Q_+$, otherwise it is *rejecting*. The labels of the activated transitions form a string $l_1 \ldots l_n$. A DFA accepts exactly those strings formed by the labels of accepting computations, it rejects all others. Since a DFA is *deterministic* there exists exactly one computation for every string, implying that for every state q and every label l there exists at most one outgoing transition from q with label l. A string s is said to *reach* all the states contained in the computation that forms s, s is said to *end* in the last state q_n of such a computation. The set of all strings accepted by a DFA A is called the *language* $L(A)$ of A.
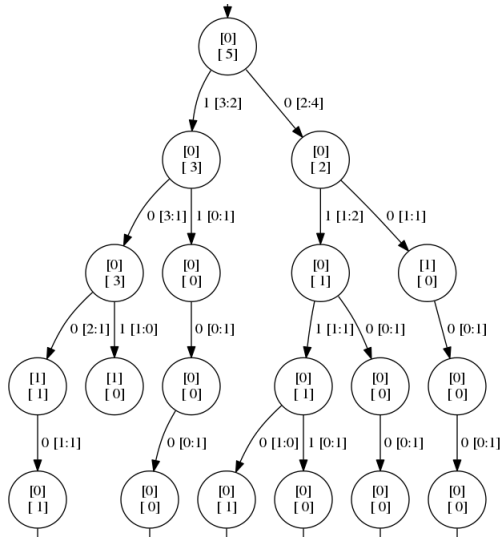

Figure 5: The initial part of the prefix tree, built from the input sample as shown in Figure 7. The square brackets indicate occurrence counts of positive data.
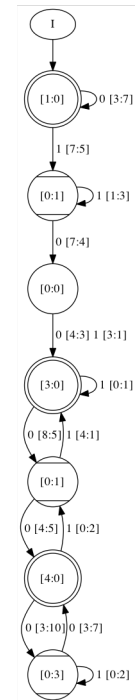
Figure 6: An automaton learned from the input sample. The numbers in square brackets indicate occurrence counts of positive and negative samples.

8

C State-Merging Algorithms

The idea of a state-merging algorithm is to first construct a tree-shaped DFA A from the input sample S, and then to merge the states of A. This DFA A is called an *augmented prefix tree acceptor* (APTA). An example is shown in Figure 5. For every state q of A, there exists exactly one computation that ends in q. This implies that the computations of two strings s and s' reach the same state q if and only if s and s' share the same prefix until they reach q. Furthermore, an APTA A is constructed to be *consistent* with the input sample S, i.e., $S_+ \subseteq L(A)$ and $S_- \cap L(A) = \emptyset$. Thus a state q is accepting only if there exists a string $s \in S_+$ such that the computation of s ends in q. Similarly, it is rejecting only if the computation of a string $s \in S_-$ ends in q. As a consequence, A can contain states that are neither accepting nor rejecting. No computation of any string from S ends in such a state. Therefore, the rejecting states are maintained in a separate set $Q_- \subseteq Q$, with $Q_- \cup Q_+ = \emptyset$. Whether a state $q \in Q \setminus (Q_+ \cup Q_-)$ should be accepting or rejecting is determined by merging the states of the APTA and trying to find a DFA that is as small as possible.

```
1 9 1 0 0 0 0 0 0 0 0
1 15 0 1 1 0 0 0 1 0 1 0 1 0 1 0 0
0 5 0 1 0 0 0
1 2 0 0
0 12 1 1 0 0 0 0 0 0 0 1 0 0
1 5 1 0 1 0 0
1 3 1 0 0
0 6 0 0 0 0 0 1
1 3 1 0 1
0 20 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 1 0 0 0
0 13 0 1 1 1 0 1 0 0 0 0 0 0 0
1 3 1 0 1
1 7 1 0 0 0 0 0 0
```

Figure 7: Input sample for the APTA in Figure 5 and the learned model in Figure 6. The first column indicates whether the sample is positive or negative (i.e. a counter-example), the second column indicates the length of the word. The following symbols form the word, with each symbol separated by a space. This input format is commonly used in tools in the grammar induction community.

A *merge* of two states q and q' combines the states into one: it creates a new state q'' that has the incoming and outgoing transitions of both q and q', i.e., replace all $\langle q, q_t, l \rangle, \langle q', q_t, l \rangle \in T$ by $\langle q'', q_t, l \rangle$ and all $\langle q_s, q, l \rangle, \langle q_s, q', l \rangle \in T$ by $\langle q_s, q'', l \rangle$. Such a merge is only allowed if the states are *consistent*, i.e., it is not the case that q is accepting while q' is rejecting or vice versa. When a merge introduces a non-deterministic choice, i.e., q'' is now the source of two transitions $\langle q'', q_1, l \rangle$ and $\langle q'', q_2, l \rangle$ in T with the same label l, the target states of these transitions q_1 and q_2 are merged as well. This is called the *determinization* process (c.f. the while-loop in Algorithm 2), and is continued until there are no non-deterministic choices left. However, if this process at some point merges two inconsistent states, the original states q and q' are also considered inconsistent and the merge will fail. The result of a successful merge is a new DFA that is smaller than before, and still consistent with the input sample S. A state-merging algorithm iteratively applies this state merging process until no more consistent merges are possible. The general algorithm is outlined in Algorithm 1. Figure 6 shows an automaton obtained from the input given in Figure 7, which is also depicted as an APTA in Figure 5.

Algorithm 1 State-merging in the red-blue framework

Require: an input sample S
Ensure: A is a DFA that is consistent with S
  $A = \mathsf{apta}(S)$ {construct the APTA A}
  $R = \{q_0\}$ {color the start state of A red}
  $B = \{q \in Q \setminus R \mid \exists \langle q_0, q, l \rangle \in T\}$ {color all its children blue}
  while $B \neq \emptyset$ **do** {while A contains blue states}
    if $\exists b \in B$ s.t. $\forall r \in R$ holds $merge(A, r, b) = \text{FALSE}$ **then** {if there is a blue state inconsistent with every red state}
      $R := R \cup \{b\}$ // color b red
      $B := B \cup \{q \in Q \setminus R \mid \exists \langle b, q, l \rangle \in T\}$ {color all its children blue}
    else
      for all $b \in B$ and $r \in R$ **do** {forall red-blue pair of states}
        compute the $\mathsf{evidence}(A, q, q')$ of $merge(A, r, b)$ {find the best performing merge}
      end for
      $A := merge(A, r, b)$ with highest $\mathsf{evidence}$ {perform the best merge}
      let q'' be resulting state
      $R := R \cup \{q''\}$ {color the resulting state red}
      $R := R \setminus \{r\}$ {uncolor the merged red state}
      $B := \{q \in Q \setminus R \mid \exists r \in R \text{ and } \langle r, q, l \rangle \in T\}$ {recompute the set of blue states}
    end if
  end while
  return A

Algorithm 2 Merging two states: $\mathsf{merge}\,(A, q, q')$

Require: an augmented DFA $A = \langle Q, T, \Sigma, q_0, Q_+, Q_- \rangle$ and two states $q, q' \in Q$
Ensure: if q and q' are inconsistent, return FALSE; else return A with q and q' merged.
  if $(q \in Q_+ \text{ and } q' \in Q_-)$ or $(q \in Q_- \text{ and } q' \in Q_+)$ **then**
    return FALSE {return FALSE if q is inconsistent with q'}
  end if
  let $A' \langle Q', T', \Sigma, q_0', Q_+', Q_-' \rangle$ be a copy of A {initialize the result A'}
  create a new state q'', and set $Q' := Q' \cup q''$ {add a new state q'' to A'}
  if $q \in Q_+$ or $q' \in Q_+$ **then**
    set $Q_+' := Q_+' \cup \{q''\}$ {q'' is accepting if q or q' is accepting}
  end if
  if $q \in Q_-$ or $q' \in Q_-$ **then**
    set $Q_-' := Q_-' \cup \{q''\}$ {q'' is rejecting if q or q' is rejecting}
  end if
  for all $t = \langle q_s, q_t, l \rangle \in T'$ with $q_s \in \{q, q'\}$ **do** {forall transitions with source state q or q'}
    $T' := T' \setminus \{t\}$ // remove the transition
    $T' := T' \cup \{\langle q'', q_t, l \rangle\}$ {add a new transition with q'' as source}
  end for
  for all $t = \langle q_s, q_t, l \rangle \in T'$ with $q_t \in \{q, q'\}$ **do** {forall transitions with target state q or q'}
    $T' := T' \setminus \{t\}$ {remove the transition}
    $T' := T' \cup \{\langle q_s, q'', l \rangle\}$ {add a new transition with q'' as target}
  end for
  set $Q' := Q' \setminus \{q, q'\}$ {remove q and q' from A'}
  while $\langle q_f, q_1, l \rangle, \langle q_f, q_2, l \rangle \in T'$ with $q_1 \neq q_2$ **do** {while non-deterministic choices exist}
    $A'' := \mathsf{merge}(A', q_1, q_2)$ {determinize the targets}
    if A'' equals FALSE **then**
      return FALSE {return FALSE if the targets are inconsistent}
    else
      $A' := A''$ {else keep the merge and continue determinizing}
    end if
  end while
  return A'
