Learning and Model-Checking Networks of I/O Automata

Hua Mao HUAMAO@CS.AAU.DK
Manfred Jaeger JAEGER@CS.AAU.DK
Department of Computer Science, Aalborg University, Denmark

Editor: Steven C.H. Hoi and Wray Buntine

Abstract

We introduce a new statistical relational learning (SRL) approach in which models for structured data, especially network data, are constructed as networks of communicating finite probabilistic automata. Leveraging existing automata learning methods from the area of grammatical inference, we can learn generic models for network entities in the form of automata templates. As is characteristic for SRL techniques, the abstraction level afforded by learning generic templates enables one to apply the learned model to new domains. A main benefit of learning models based on finite automata lies in the fact that one can analyse the resulting models using formal model-checking techniques, which adds a dimension of model analysis not usually available for traditional SRL modeling frameworks.

Keywords: Automata Learning, Relational Learning, Probabilistic Model Checking, Network Data

1. Introduction

In this paper we introduce a new type of probabilistic model that combines three distinct lines of research: grammatical inference, formal verification by model checking, and statistical relational learning (SRL). As a result, we obtain a new SRL framework that can be used to learn models for typical SRL domains (e.g., sensor networks, social networks, multi-agent systems, biological systems), and which enables us to perform (model-checking) types of analyses on the learned model that so far have not been utilized in SRL.

Grammatical inference (de la Higuera, 2010) is concerned with learning language specifications in the form of grammars or automata from data consisting of strings over some alphabet Σ. Starting with Angluin's seminal work (Angluin, 1987), methods have been developed for learning deterministic, non-deterministic and probabilistic grammars and automata. An efficient algorithm exists for learning probabilistic automata that are deterministic in the sense that from every state q of the automaton, and for every symbol $s \in \Sigma$, there is at most one possible (i.e. non-zero probability) transition from q with label s (Carrasco and Oncina, 1994). For this algorithm theoretical consistency results establish that the data-generating automaton is identified in the large-sample limit (Carrasco and Oncina, 1994).

Model Checking is a verification technique to determine whether a system model complies with a specification provided in a formal language (Baier and Katoen, 2008). In the simplest case, system models are given by finite non-deterministic or probabilistic automata, but model-checking techniques have also been developed for more sophisticated system models, e.g., timed automata (Kwiatkowska et al., 2004). Powerful software tools that are available for model checking include UPPAAL (Behrmann et al., 2011) and PRISM (Kwiatkowska et al., 2011). Traditionally, models used in model-checking are manually constructed, either in the development phase as system designs, or for existing hard- or software systems from known specifications and documentation. Since such model-construction can be extremely time-consuming, or even infeasible in the case of insufficient documentation for an existing system, there is an increasing interest in *model learning* (or *specification mining*) for formal verification (Ammons et al., 2002; Sen et al., 2004; Mao et al., 2011; Chen et al., 2012). The learning methods in this area often are adapted grammatical inference techniques.

Models for complex systems are typically constructed in the form of multiple interacting components, each of which is a finite automaton itself. To model such individual system components, one needs an automaton model that allows us to condition the probabilistic, observable output of a system component on the inputs it receives from other components. The standard learning approach for finite probabilistic automata has been extended to learning of such I/O automata in (Mao et al., 2012). Moreover, complex systems can be composed of multiple essentially identical components. Typical benchmark models from the field of model checking that have this structure are for systems of multiple connected processors (Itai and Rodeh, 1990; Herman, 1990).

Learning probabilistic models for complex systems composed of multiple identical components can also be said to be the general theme of *statistical relational learning*. Even though here a great variety of different modeling approaches is used – they all give rise to probabilistic models for structured domains consisting of a number of objects, or entities, which are described by attributes, and connected by relations. Furthermore, they all construct such models out of instantiations of generic modeling templates for the individual domain objects.

In the SRL approach we introduce in this paper, generic object (entity, component, ...) models consist of finite probabilistic input/output (I/O) automata templates. Complex domains are modeled as networks of instantiations of these automata templates, typically involving various types of objects described by different templates. Relations between the objects determine how the inputs of one object are defined by the outputs of other objects. Compared to other SRL models, the resulting framework has the following distinguishing features:

- Probabilistic dependencies between related objects are induced by explicit (and observable) communication of I/O symbols.
- The models are inherently dynamic (temporal): they define a probability distribution over discrete time series of observations of the relational system state.
- The models support analysis of the system dynamics by formal model checking of properties expressed in linear time temporal logic.

This paper is intended as a proof-of-concept: we introduce the necessary conceptual framework of networks of I/O automata in Section 2. In Section 3 we show how the learning method for I/Oautomata can be used to learn automata templates. Section 4 illustrates our approach on a small toy example for a social network model and the Ceará dataset. The main contribution is to show the feasibility of learning SRL models consisting of interacting

I/O automata, and to demonstrate the usefulness of model checking approaches for model analysis.

2. IODPFA Networks

We begin by defining I/O deterministic finite automata, which will become the basic building blocks of our relational models.

Definition 1 *(IODPFA) An input/output deterministic probabilistic finite automaton (IODPFA) is a tuple $M = (Q, \Sigma^{in}, \Sigma^{out}, q^s, \mathbf{P}, L)$ where*

- *Q is a finite set of states.*
- *Σ^{in} is a finite input alphabet.*
- *Σ^{out} is a finite output alphabet, and $L : Q \rightarrow \Sigma^{out}$ is a labeling function.*
- *q^s is the initial state.*
- *$\mathbf{P} : Q \times \Sigma^{in} \times Q \rightarrow [0,1]$ is the transition probability function such that for all $q \in Q$ and all inputs $\alpha \in \Sigma^{in}$, $\sum_{q' \in Q} \mathbf{P}(q, \alpha, q') = 1$.*
- *For all $q \in Q$, $\sigma \in \Sigma^{out}$, and $\alpha \in \Sigma^{in}$: there exists at most one $q' \in Q$ with $L(q') = \sigma$ and $\mathbf{P}(q, \alpha, q') > 0$. We then also write $\mathbf{P}(q, \alpha, \sigma)$ instead of $\mathbf{P}(q, \alpha, q')$.*

The existence of a unique initial state, and the last condition in the definition, together make this model deterministic. If we omit the dependence on an input symbol from Σ^{in}, one obtains what is known as a deterministic labeled Markov Chain (DLMC).

Example 1 *(IODPFA)*
In Figure 1 (a), two IODPFA T_1 and T_2 are given. For T_1, the initial state is labeled by 0, $\Sigma_1^{in} = \{a, b\}$, and $\Sigma_1^{out} = \{0, 1\}$. For T_2, the initial state is labeled by a, $\Sigma_2^{in} = \{0,1\} \times \{0,1\}$, and $\Sigma_2^{out} = \{a, b\}$. The edge label $(1,0)/(0,1) : 1$, for example, means that this transition is taken for the two inputs $(1,0)$ and $(0,1)$ with probability 1.

We construct networks out of IODPFA components. A network will usually consist of multiple identical IODPFAs, but it may also contain several distinct types of IODPFAs. In the following, we refer to an IODPFA in the sense of Definition 1 also as an IODPFA *template* to emphasize the fact that our final models contain multiple copies, or instantiations, of these basic building blocks. Apart from specifying how many instantiations of which templates are contained in the nework, one has to specify how the outputs of some components provide the inputs of other components. This leads to the following definition.

Definition 2 *(IODPFA Network)*
Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be a set of IODPFA templates, i.e.,

$$T_i = (Q_i, \Sigma_i^{in}, \Sigma_i^{out}, q_i^s, \mathbf{P}_i, L_i)$$

is an IODPFA in the sense of Definition 1. Furthermore, for each T_i, there exists $k_{i,1}, \ldots, k_{i,m} \in \mathbb{N} \cup \{0\}$ such that

$$\Sigma_i^{in} = \mathop{\times}_{j=1}^{m} \left(\Sigma_j^{out}\right)^{k_{i,j}}$$

An IODPFA Network for $\mathcal{T}$ ($\mathcal{T}$-Network for short) is given by

- *Sets of automata $\mathcal{C}_i = \{C_i^1, \ldots, C_i^{u_i}\}$ for each IODPFA template T_i ($u_i \in \mathbb{N}$).*
- *An I/O mapping $\mathcal{R}_{I/O}$:*

$$C_i^k \to \mathop{\times}_{j=1}^{m} \left\{C_j^1, \ldots, C_j^{u_j}\right\}^{k_{i,j}}$$

Example 2 *(IODPFA Networks)*
In the network of Figure 1 (b), there are two templates $\mathcal{T} = \{T_1, T_2\}$. T_1 receives the input from one component of T_2, i.e., $k_{1,1} = 0$, $k_{1,2} = 1$. T_2 receives the input from two components of T_1, i.e., $k_{2,1} = 2$, $k_{2,2} = 0$. There are two instance for T_1, i.e., $\mathcal{C}_1 = \{C_1^1, C_1^2\}$, and one instance for T_2, i.e., $\mathcal{C}_2 = \{C_2^1\}$. Both C_1^1 and C_1^2 receive inputs from C_2^1, and $\mathcal{R}_{I/O}(C_1^1) = \mathcal{R}_{I/O}(C_1^2) = (\emptyset, C_2^1)$. C_2^1 receives inputs from C_1^1, C_1^2, and $\mathcal{R}_{I/O}(C_2^1) = ((C_1^1, C_1^2), \emptyset)$.

Semantically, IODPFA networks are intended to model discrete time stochastic systems, whose components make transitions simultaneously and independently (given the system state at the previous time step). The overall system then can be seen just as a labeled Markov Chain, formally defined as the synchronous product in the sense of the following definition.

Definition 3 *(Synchronous Product)*
Let $\mathcal{C} := \bigcup_{i=1}^{m} \mathcal{C}_i = \{C_1, \ldots, C_u\}$ be the set of all components in the system, $C_h = (Q_h, \Sigma_h^{in}, \Sigma_h^{out}, q_h^s, \mathbf{P}_h, L_h)$. Each component may come from different IODPFA templates, and here we omit the index of IODPFA type. Given the I/O relation of these components $\mathcal{R}_{I/O} : h \to \{1, \ldots, u\}^{k_h}$ (i.e. $k_h = \sum_{j=1}^{m} k_{i,j}$ if $C_h \in \mathcal{C}_i$), the synchronous product *of components is defined as : $M = \bigotimes_{h=1}^{u} C_h = (\mathop{\times}_{h=1}^{u} Q_h, \mathop{\times}_{h=1}^{u} \Sigma_h^{out}, q^s, \mathbf{P}, L)$, where*

- *$q^s = (q_1^s, \ldots, q_u^s)$*
- *the transition matrix $\mathbf{P}$ is defined by the following rule:*

$$(q_1, \ldots, q_u) \xrightarrow{\mathop{\times}_{h=1}^{u} \mathbf{P}_h(q_h, \alpha_h, q_h')} (q_1', \ldots, q_u')$$

where $q_h \in Q_h$, and $\alpha_h = \mathop{\times}_{j \in \mathcal{R}_{I/O}(h)} L(q_j) \in \Sigma_h^{in}$.

- *$L(q_1, \ldots, q_u) = (L(q_1), \ \ldots, \ L(q_u))$*

Example 3 *(Synchronous Product)*

Figure 1 (c) shows the synchronous product of the network in (b) with 3 components, i.e., C_1^1, C_1^2 and C_2^1. There are 8 states and 23 transitions in the product, the initial state is labeled with symbols $(0, 0, a)$. Each state is labeled by combined symbols from 3 components. From the initial state q^s a transition to the state labeled by $(0, 1, a)$ is made with probability $P = P_1^1(0, a, 0) \times P_1^2(0, a, 1) \times P_2^1(a, 00, a) = 0.125$.

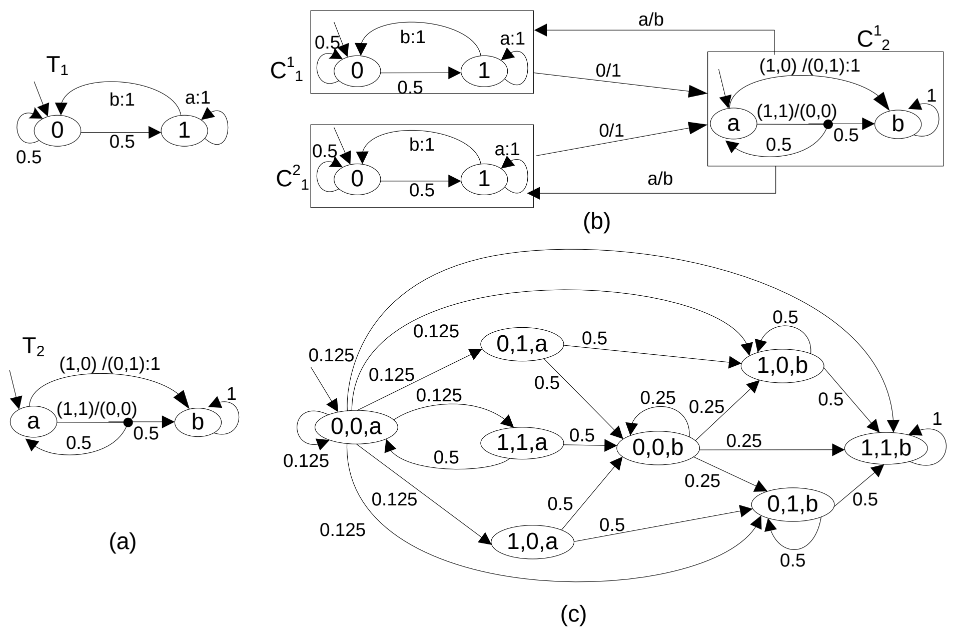

Figure 1: (a) Two IODPFA templates: T_1 and T_2. (b) A IODPFA network with 3 components: C_1^1, C_1^2 and C_2^1. (c) The synchronous product of 3 components.

Apart from defining the semantics of an IODPFA network, the synchronous product is also constructed in practice when exact model checking is performed for an IODPFA network. Note that the size of the synchronous product is exponential in the number of components of the IODPFA network, which limits the applicability of methods that require a compilation into the synchronous product.

2.1. Probabilistic LTL

Linear time temporal logic (LTL) (Baier and Katoen, 2008) is a temporal logic that expresses properties about the relation between the state labels in executions. The basic ingredients of LTL-formulae are atomic propositions (state labels $a \in \Sigma^{out}$), the Boolean connectors conjunction $\wedge$, and negation $\neg$, and two basic temporal modalities $\bigcirc$ ("next") and U ("until"). Linear time temporal logic (LTL) over Σ^{out} is defined by the syntax

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathsf{U} \varphi_2 \qquad a \in \Sigma^{out}$$

Let φ be an LTL formula over Σ^{out}. For $s = \sigma_0\sigma_1 \ldots \in (\Sigma^{out})^\omega$, $s[j \ldots] = \sigma_j\sigma_{j+1}\sigma_{j+2}\ldots$ is the suffix of s starting with the $(j+1)$st symbol σ_j. Then the LTL semantics for infinite words over Σ^{out} are as follows:

- $s \models true$
- $s \models a$, iff $\sigma_0 = a$
- $s \models \varphi_1 \wedge \varphi_1$, iff $s \models \varphi_1$ and $s \models \varphi_2$
- $s \models \neg\, \varphi$, iff $s \not\models \varphi$
- $s \models \bigcirc\, \varphi$, iff $s[1 \ldots] \models \varphi$
- $s \models \varphi_1 \mathsf{U} \varphi_2$, iff $\exists j \geq 0$ s.t. $s[j \ldots] \models \varphi_2$ and $s[i \ldots] \models \varphi_1$, for all $0 \leq i < j$

The syntax of probabilistic LTL (PLTL) is:

$$\phi ::= P_{\bowtie r}(\varphi) \quad (\bowtie \in \geq,\ \leq,\ =;\ r \in [0,1];\ \varphi \in \text{LTL})$$

We use $(\Sigma^{out})^\omega$ to denote the set of infinite strings over Σ^{out}. A LMC M defines a probability distributionP_M over $(\Sigma^{out})^\omega$. A labeled Markov chain M satisfies the PLTL formula $P_{\bowtie}(\varphi)$ iff $P_M(\varphi) \bowtie r$, where $P_M(\varphi)$ is short for $P_M(\{s \mid s \models \varphi, s \in (\Sigma^{out})^\omega\})$.

3. Learning IODPFA Networks

3.1. Data

The data we learn from is generated by observing a running IODPFA network. At each time step the output of all system components will be observed. Thus, data generated by a run of the system shown in Figure 1 would be of the form

$$(0(C_1^1), 0(C_1^2), a(C_2^1)), (0(C_1^1), 1(C_1^2), b(C_2^1)), (1(C_1^1), 0(C_1^2), b(C_2^1)), \ldots$$

From this sequence we can extract multiple training sequences for learning the component IODPFAs. For this it is required that we know the I/O relation in the network. Assuming this to be as shown in Figure 1, one obtains from the network sequence above two training sequences for the IODPFA T_1:

$$0, a, 0, b, 1, b, \ldots$$
$$0, a, 1, b, 0, b, \ldots$$

and one training sequence for the the IODPFA T_2:

$$a, (0,0), b, (0,1), b, (1,0), \ldots$$

These sequences, now, consist of alternating output and input symbols in the vocabularies of the given IODPFA types. We note that due to the independence of the transitions taken in the indivdual components of the network (more precisely: conditional independence given the inputs), multiple training sequences for one component type obtained from a single network sequence can be treated as statistically independent samples, even if the input sequences are the same, as they are here in the two training sequences for T_1.

Our learning algorithm for IODPFAs requires as input independently sampled I/O sequences $S_1, S_2, \ldots, S_n$. In order to guarantee that the data-generating automaton is correctly identified in the limit $n \to \infty$, the S_i have to satisfy certain "richness" conditions:

first, the S_i must contain sequences of sufficient length, so as to guarantee that all states of the data-generating automaton are represented in the data. In our data-generation protocol for synthetic data we ensure this condition by sampling sequences whose length is randomly determined according to an exponential distribution. This ensures that the length of sequences is unbounded, even though sequences of much larger length than the expected value of the given exponential distribution will become very rare. Second, the input sequences must be sufficiently diverse (a property related to what is known as *fairness* in the verification literature) so that data is collected for all possible inputs at all states.

3.2. IOalergia

This algorithm, named IOalergia, for learning IODPFA is an adapted version of the Alergia algorithm (Carrasco and Oncina, 1994; de la Higuera, 2010). The algorithm starts with the construction of the *input and output frequency prefix tree acceptor* IOFPTA which is the representation of the set of strings $\mathcal{S}$ in the data. Each node in the tree is labeled by an output symbol $\sigma \in \Sigma^{out}$, and each edge is labeled by an input action $\alpha \in \Sigma^{in}$. Every path from the root to a node corresponds to a string $s \in \text{prefix}(\mathcal{S})$, where $\text{prefix}(\mathcal{S})$ is the set of all prefixes of all strings in the dataset. The node s is associated with the frequency function $f(s,\alpha,\sigma)$ which is the number of strings in $\mathcal{S}$ with the prefix $s\alpha\sigma$, and $f(s,\alpha) = \sum_{\sigma\in\Sigma^{out}} f(s,\alpha,\sigma)$. An IOFPTA can be transformed to IODPFA by normalizing frequencies $f(s,\alpha,\cdot)$ to $\mathbf{P}(s,\alpha,\cdot)$. The IOFPTA in Figure 2 (a) is constructed from a small sample dataset generated by C_1^1 and C_1^2 in Figure 1 (b).

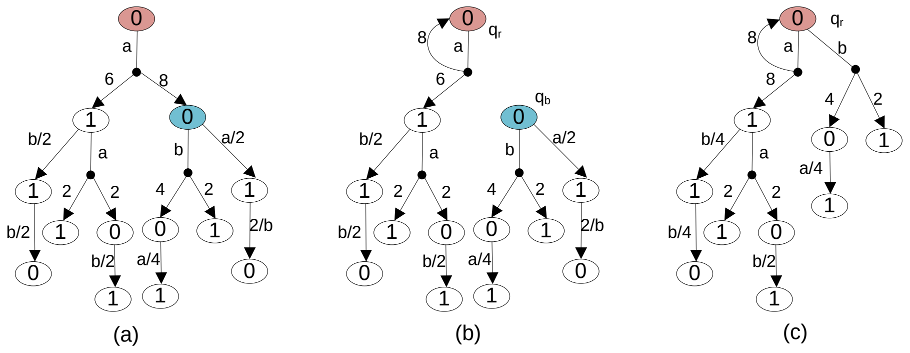

Figure 2: (a) The IOFPTAconstructed from the dataset generated by C_1^1 and C_1^2. (b) and (C) illustrate the procedure of merging node q_b to node q_r.

The basic idea of this learning algorithm is to approximate the generating model by grouping together nodes in IOFPTA which can be mapped to the same state in the generating model. Nodes which can be mapped to the same state in the generating model will pass the compatible test based on the Hoeffding bound parameterized by ϵ. Formally, two nodes q_r and q_b are ϵ-compatible ($1 \geq \epsilon > 0$), if it holds that:

1. $L(q_r) = L(q_b)$
2. For all $\alpha \in \Sigma^{in}$, and $\sigma \in \Sigma^{out}$,

$$\left|\frac{f(q_r,\alpha,\sigma)}{f(q_r,\alpha)}-\frac{f(q_b,\alpha,\sigma)}{f(q_b,\alpha)}\right|<\left(\sqrt{\frac{1}{f(q_r,\alpha)}}+\sqrt{\frac{1}{f(q_b,\alpha)}}\right)\cdot\sqrt{\frac{1}{2}\ln\frac{2}{\epsilon}}$$

3. Nodes $q_r\alpha\sigma$ and $q_b\alpha\sigma$ are ϵ-compatible, for all $\alpha\in\Sigma^{in}$, and $\sigma\in\Sigma^{out}$

Condition 1) requires two nodes in the tree to have the same label. Condition 2) defines the compatibility between each outgoing transition with the same input action respectively from nodes q_r and q_b. If two nodes in IOFPTA are compatible, then distributions for all input actions should pass the *compatibility* test. The last condition requires the compatibility to be recursively satisfied for every pair of successors of q_r and q_b.

If two nodes in the tree pass the *compatibility* test which means they can be mapped to the same state in the generating model, then they will be merged, as well as their successor nodes. Figure 2 (b) and (c) show the procedure of merging node q_b with q_r. In (b), the transition from the node q', i.e., q_r, to q_b is redirected to q_r, then q_r has a selfloop with the action a. In (c), the subtree with the a action from q_b is fold to the subtree of q_r. For another action b of the node q_b, there is not b action from q_r, so the subtree of the action b from q_r is added as a subtree of q_r.

In our experiments, we tune the choice of ϵ so as to obtain the best approximation to the real model by maximizing the *Bayesian Information Criterion (BIC)* score of the model, or by *cross-validation*.

4. Experiments

In this section, we apply our approach on a small toy example for a social network model, and the real Ceará dataset.

4.1. Aggregation Components

The definition of IODPFAs and IODPFA networks is somewhat rigid in the sense that each IODPFA template has a fixed interface with other templates. In particular, the number of inputs that a particular template receives is fixed. However, in many applications one may want to be able to model a dependence on a varying number of inputs. For example, in a social network model, where each person is represented by an IODPFA component, it is natural that a person receives as input the output from all his or her neighbors in the network. Such dependencies based on one-to-many relationships are quite commonly dealt with in SRL frameworks by means of *aggregation* or *combination* operators.

To model dependencies on an arbitrary number of inputs in the IODPFA framework, one can extend the definition of IODPFAs to allow infinite input alphabets, which then, in particular, could be of the form $(\Sigma^{out})^*$, thus allowing sequences of arbitrary length of inputs from a basic (finite) output alphabet Σ^{out}. Also, the definition of the I/O mapping $\mathcal{R}_{I/O}$ for IODPFA networks can be generalized to allow a component whose input alphabet is of the form $(\Sigma^{out})^*$ to be mapped to an arbitrary number of input-providing components. The resulting generalized form of IODPFA network still compiles as a synchronous product into a simple labeled Markov Chain, and therefore does not give rise to fundamental new issues in terms of semantics, or for the model checking algorithms. However, the IOalergia learning algorithm crucially relies on the fact that the input alphabets for IODPFAs are finite, and, moreover, it will only work in practice for input alphabets of moderate size.

To combine the flexibility of modeling with aggregation operations with our learning algorithms, we introduce *aggregation components*, which are IODPFAs in the generalized sense, but which are assumed to be defined and fixed, not learned. As an example, Fig 3 shows an aggregation component we use for our toy social network model. This component has as input alphabet $(\Sigma^{out})^*$, where Σ^{out} contains the symbol *share*. The transitions of the aggregation component only depend on how many components of its input string are equal to *share*. In the figure, this is represented, for example, with $s = 2 : 1$, meaning that if the input contains two *share* symbols, then this transition is taken with probability 1. The output alphabet of the aggregation component is $\Sigma^{out}_{aggr} = \{N = 0, 3 > N > 0, N \geq 3\}$ which represents the total count of *share* symbols seen in the past, binned into the three intervals $[0], [1, 2], [3, \infty]$. We can then model users in the network by IODPFAs with input alphabet Σ^{out}_{aggr}, and which are connected to a single aggregation component, that serves as an interface between users and their varying number of neighbors.

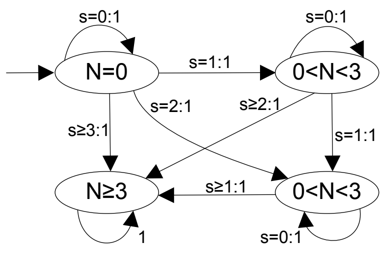

Figure 3: The aggregation model

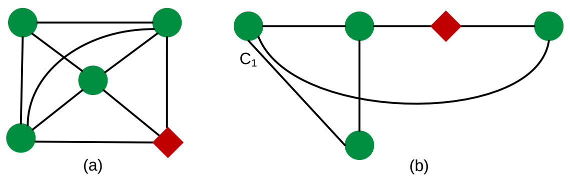

Figure 4: Structures of social networks

4.2. Social Networks

In this section, we are going to show the applicability of learning algorithm by a case study on the message spread in a social network. We will learn 2 types of user models in the network, and use the learned models to built a new network. PLTL properties of networks built by the real model and learned model will be checked by the probabilistic model checking tool PRISM (Kwiatkowska et al., 2011).

We manually construct two types of users in the network represented by IODPFA templates T_p and T_u. The user from T_p has 6 possible observable behaviors, i.e., outputs: *log in*, *log out*, *update* his own profile, *view* others' profiles; *comment* on the message or *share* it. When a specific message enters the network, a user in the network can view this message only if at least one of his connected friends has shared it. When the user views the message, the probability for sharing or only commenting on it depends on the number N of friends who has shared the message, according to $N = 0, 0 < N < 3$, or $N \geq 3$. The user who only comments on the message may change his mind as the number N increases. The user will not share the message again if he has already shared it before, but he can still update his own profile, log in or log off the system.

Users of T_u will only update their own profiles or comment on the message from neighbors, but never share it. The probability that they give comments on the message also

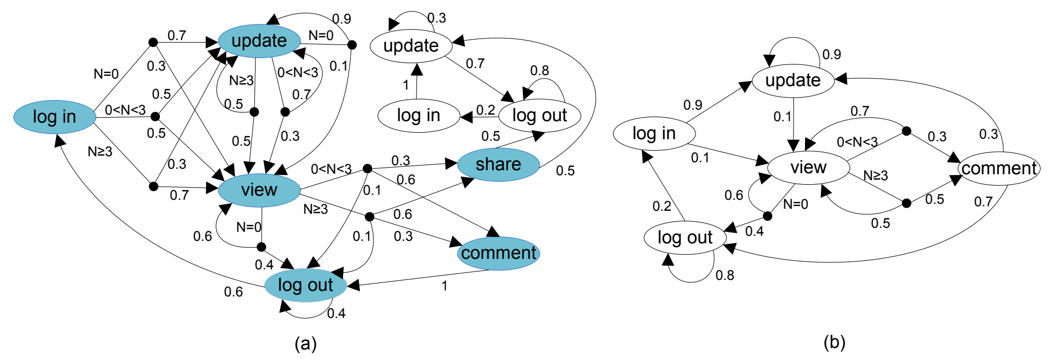

Figure 5: (a) The model T_p. (b) The model T_u.

Table 1: Learning results for T_u

$\lvert\mathcal{S}\rvert$	$\lvert Seq\rvert$	Time	$\lvert Q\rvert$	$\lvert Tran\rvert$
121	3	0.4	6	10
404	8	0.7	5	16
627	15	1.0	5	22
809	21	1.2	5	25
1000	26	1.3	5	26
4000	91	3.6	5	27
10000	249	7.7	5	28
20000	506	14	5	28

Table 2: Learning results for T_p

$\lvert\mathcal{S}\rvert(\times 10^3)$	$\lvert Seq\rvert$	Time	$\lvert Q\rvert$	$\lvert Tran\rvert$
10	262	7.3	8	49
20	497	12.2	10	56
40	1001	26.4	15	82
80	2004	50.6	21	98
100	2523	52.2	12	66
200	4995	99.0	10	56
400	9946	268.1	10	56
600	15101	397.4	10	56

depends on N. These two types of users are represented as the IODPFAs shown in Figure 5. There are 3 inputs for T_p and T_u, i.e., $\Sigma_p^{in} = \Sigma_u^{in} = \{N = 0, 3 > N > 0, N \geq 3\}$. For T_p, $\Sigma_p^{out} = \{$*log in, update, view, log out, share, comment*$\}$, and for T_u, $\Sigma_u^{out} = \{$*log in, update, log out, comment*$\}$.

The inputs for the user IODPFAs are provided by aggregation components from the outputs of all neighbors, as described in Section 4.1. The structure of the network we use to generate data is shown in Figure 4 (a). The 4 green circles are users from T_p and 1 red diamond is the user from T_u. For clarity, the aggregation components are not shown separately in the figure. For the template T_u, the initial state is the one labeled with the symbol *log in*, and there are 5 states and 28 transitions. For the template T_p, the initial state is a special state not shown in the figure, from which there are transitions to the blue nodes in Figure 5 (a). In this way we are effectively obtaining an initial state distribution, rather than a unique start state. Note, though, that for the automaton to be deterministic, we can only have an initial distribution that does not place nonzero probabilities on two distinct states with the same label.

There are 10 states and 56 transitions in T_p. Learning results for two templates are in table 1 and table 2: $\lvert\mathcal{S}\rvert$ is the number of symbols in the dataset, $\lvert Seq\rvert$ is the number of sample sequences, 'Time' is learning time (in seconds), including the time for constructing IOFPTA and the time for optimizing the BIC score, $\lvert Q\rvert$ is the number of states in the learned model, and $\lvert Tran\rvert$ is number of transitions. The structure of the model T_u was correctly identified from datasets of at least 10×10^3 symbols, whereas exact learning of T_p required 200×10^3 symbols.

Using both the original and the learned templates, we construct a new network with the structure in Figure 4 (b). We then model check the PLTL property: $P(true \, \mathsf{U}_{\leq L} \, C_1 = share)$, i.e., the probability that the user C_1 shares the message in L steps. Figure 6 shows the model checking results for templates learned from different sample sizes, and the original ones of Figure 5. One observes that models learned from smaller amounts of data that do not exactly match the structure of the true model already provide reasonable approximations for these PLTL queries. The model learned from 200×10^3 symbols almost exactly matches the true model (the two curves are overlaid).

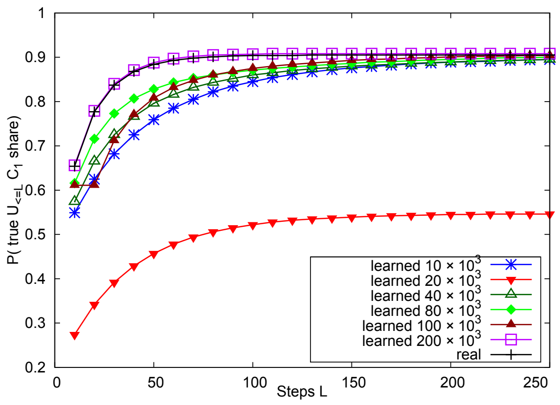

Figure 6: Model checking results for user templates.

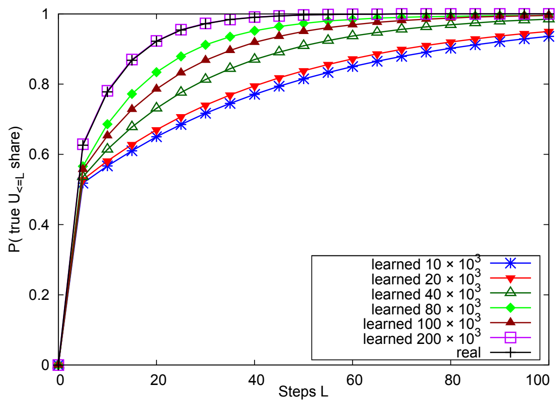

Figure 7: Model checking results for probabilistic scheduler.

As the number of components increases, the size of the product grows exponentially, and it becomes intractable for PRISM to model check properties of users in large networks. However, we can still perform formal model checking analyses of learned user templates with a view towards users in large networks by simulating a larger network environment with simple IODPFAs that are designed to provide inputs for a user template. Adapting standard concepts from the verification literature, we call IODPFAs of this kind *schedulers*. In a first test, we construct a very simple scheduler that generates the outputs $N = 0, 0 < N < 3, N \geq 3$ randomly, with uniform probability. The scheduler itself does not require any inputs. We can now model check PLTL properties of the user template in the two-component network consisting of a single user and a scheduler component. Results for the same PLTL queries as before are shown in Figure 7.

Often it will be difficult to design a full probabilistic specification of a scheduler representing a complex environment. In that case we can also construct schedulers in which only possible transitions are specified, but not their probabilities, i.e., the scheduler is given by a nondeterministic automaton model, not a probabilistic one (note that nondeterministic here is not the negation of "deterministic" in the sense of Definition 1). Based on such a nondeterministic scheduler, one can compute minimal and maximal probabilities for PLTL properties that can be obtained by taking transitions according to specific policies.

We have constructed a nondeterministic scheduler that produces output sequences of the form $\Sigma^{out}_{sched} = "N = 0"("3 > N > 0" \mid "N \geq 3")^{\omega}$. Thus, it is only specified what

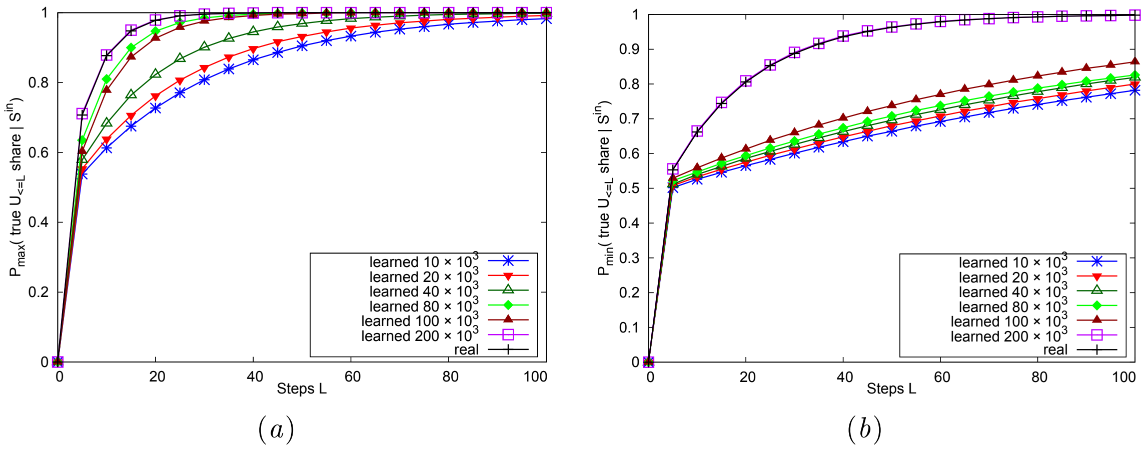

Figure 8: Model checking results for the nondeterministic scheduler

outputs the scheduler can produce, but not their relative probabilities. Figure 8 shows the minimum and maximum probabilities for the PLTL queries $P(true \mathsf{U}\, share|S^{in})$ obtainable by optimal choices of $S^{in} \in \Sigma^{out}_{sched}$. Again we observe that learned templates provide good approximations for queries asked relative to partially specified environment models given by nondeterministic schedulers.

4.3. Rainfall Occurrence over Brazil

We use daily rainfall data collected at 10 stations from the state of Ceará in Brazil over the years 1975-2002, provided by FUNCEME. The dataset contains 24 sequences of 90 day observations (February, March and April) for each of 10 weather stations located in northeast Brazil. The years 1976, 1978, 1984 and 1986 were omitted because of missing data for certain months at one or more stations, yielding 24 complete 90-day (2160 days).

If the nearest station is wet/dry, then the current station will probable also be wet/dry. We assume that the current station j will be affected by its nearest neighbor station i, denoted as $i \to j$. Then relation of these 10 stations are: $4 \to 1$, $7 \to 2$, $6 \to 3$, $8 \to 4$, $10 \to 5$, $3 \to 6$, $2 \to 7$, $4 \to 8$, $10 \to 9$, $9 \to 10$. Each station will receive input from its nearest neighbor, and stations can be modeled as IODPFAs. In order to make each sequence start from the same symbol, an artificial symbol 'start' is added at the beginning of every sequences, and each sequence has 181 input and output symbols. There are 3 outputs for each station, i.e., $\Sigma^{out} = \{$start, rain, not-rain$\}$, and two inputs $\Sigma^{in} = \{$rain, not-rain$\}$.

Following Taylor and Hinton (2009), under leave-6-out we train 4 models, one for each set obtained by leaving out 1/4 non-overlapping consecutive sequences. Each model is then evaluated on the corresponding left-out 1/4 sequences. The ϵ we used to learn the model from all sequences is the average value of these four different values. For each station, we learn 3 types of models. A) Learn a single template T_A for all 10 stations. All 240 training sequences are then used to learn the structure and the parameters. B) Keep the structure of T_A, but learn individual transition distributions for each station. Parameter, thus, are fitted for each station from 24 training sequences. C) The model for each station will be learned separately from their own data (24 sequences) by the learning algorithm.

Learning results for experiment A, B and C of 4 models are shown in table 3: LL_i is the log likelihood per-station, per-year and per-day for model i; $\overline{LL}$ is the average log likelihood of $LL_1, \ldots, LL_4$; $|Q|$ and $|Tran|$ are the number of states and transitions in the model learned from all data.

Table 3: Learning results for the rainfall data.

	LL_1	LL_2	LL_3	LL_4	$\overline{LL}$	$\|Q\|$	$\|Tran\|$
A	-0.623	-0.627	-0.641	-0.621	-0.628	4	14
B	-0.6204	-0.626	-0.632	-0.612	-0.623	4	14
C	-0.6209	-0.624	-0.630	-0.612	-0.622	3	10

The log-likelihood score obtained in our model is a little lower than that obtained in HMMs (Taylor and Hinton, 2009). One reason is that HMMs can model the non-stationarity of daily rainfall probability over the measuring period by means of a hidden state variable that represents different time intervals. The IODPFA network model does not allow for such a latent time variable. On the other hand, the I/O relations in our model enables us to model the dependencies of rainfall among spatially neighboring stations, which is not taken into account in the HMM model. While, thus, not being as accurate in terms of overall likelihood score, we may investigate with out models local dependencies, as illustrated in the following.

We build networks consisting of connected components according to the nearest-neighbor relationship, and there are 4 connected networks:$(1, 4, 8), (2, 7), (3, 6)$, and $(5, 9, 10)$. Stations that are not in the same network have no influence on each other. We want to test how well the observation at one station at time t predicts the observation at another station at time $t + 1, t + 2, ...$ by checking two kinds of PLTL properties on the network of stations 1, 4, and 8: $P(\varphi_1) = P(\bigcirc^L\ rain(j) \mid \bigcirc\ rain(i))$: the probability that station j has rain at step L given station i has rain in the second step, and $P(\varphi_2) = P(\bigcirc^L\ rain(j))$: the probability that station j has rain at step L ($\bigcirc^L = \overbrace{\bigcirc\bigcirc\ldots\bigcirc}^{L}, i, j \in \{1, 4, 8\}$). Differences between φ_1 and φ_2, i.e., $D(\varphi_1, \varphi_2) = P(\varphi_1) - P(\varphi_2)$, for experiments A and B are shown in table 4.

Table 4: Differences between φ_1 and φ_2 for experiments A and B

	Steps	$1 \to 4$	$1 \to 8$	$4 \to 1$	$4 \to 8$	$8 \to 1$	$8 \to 4$
$D_A(\varphi_1, \varphi_2)$	L=2	0	0	0.1332	0.1332	0	0.1332
	L=3	0	0	0.0489	0.0497	0.0225	0.0497
	L=4	0	0	0.0402	0.0333	0.0122	0.0238
	L=5	0	0	0.0143	0.0173	0.0067	0.0079
	L=6	0	0	0.0056	0.0055	0.0026	0.0035
$D_B(\varphi_1, \varphi_2)$	L=2	0	0	0.1316	0.1049	0	0.1212
	L=3	0	0	0.0332	0.0233	0.0184	0.0338
	L=4	0	0	0.0252	0.0144	0.0077	0.0176
	L=5	0	0	0.0059	0.0109	0.0040	0.0034
	L=6	0	0	0.0022	0.0013	0.0011	0.0020

The information of rain in station 1 will not effect station 4 and 8 since station 1 will not provide inputs for other stations. From table 4, we get the same result, i.e., the difference of $P(\varphi_1)$ and $P(\varphi_2)$ for both networks built in experiments A and B are 0. The rain in station 8 can not affect station 1 in the second since they are not directly connected with each other, but the information can be propagated to station 1 through station 4 in the third step. Then for $L = 2$, the difference is 0, and for $L > 2$, the differences for $8 \to 1$ are greater than 0. There are direct connections between 4 and 8, 1 and 4, i.e., $4 \to 1$, $8 \to 4$, and $4 \to 8$, and the rain from the neighbor has impact on the connected stations, and differences are larger than 0. As the time increases, this impact of the nearest neighbor will decrease, i.e., the difference decreases as L increases.

5. Related Work

The full system model given by the synchronous product of component I/O automata can be understood as a special type of Hidden Markov Model: the states $q \in \times_{i=1}^{k} Q_i$ of the synchronous product are hidden states of a HMM generating outputs in $\times_{i=1}^{k} \Sigma_i^{out}$. Apart from the special structure of state and output space, the model is special in that the observed output is a deterministic function of the current state, so that the system behavior is characterized by the state transition probabilities only. However, it should be noted that any HMM can be transformed into an equivalent one with deterministic observations by a suitable extension of the hidden state space.

Several extensions of the basic HMM model have been proposed in which the unstructured hidden state space is replaced by a structured representation consisting of several factors (Ghahramani and Jordan, 1997; Taylor and Hinton, 2009). The individual state factors here have a completely different interpretation from our component automata, however, and the factorial structure of the state space is not reflected in an analogous structure of the output space. Apart from generic model-selection techniques (for example by cross-validation or by model scores such as BIC or MDL), no specialized learning methods for the hidden state space structure are available for these models. This is in stark contrast to our learning method for I/O automata with its strong theoretical identifiability guarantees. These guarantees, however, depend crucially on the restrictive assumption that the automata are deterministic, which in the general HMM perspective means that the next state is uniquely determined by the current state and the next observed symbol.

An individual IODPFA is related to an Input/Output HMM (IOHMM) (Bengio and Frasconi, 1994). Originally introduced as models for continuous inputs and outputs, discrete versions of IOHMMs have also been used (Hochreiter and Mozer, 2001). Again, IODPFAs are distinguished by the special structure and learning algorithm for their internal state space.

Within the field of SRL, Logical Hidden Markov Models (LOHMMs) (Kersting et al., 2006) have been proposed as a dedicated framework for modeling sequences of symbols that have the same relational structure *attribute(object)* and *relation(object1,object2)* as the observations in our framework. However, the underlying generative model for such structured symbol sequences is very different from our IODPFA networks, and it supports different types of inference tasks. A number of more expressive SRL frameworks exist that could be used to encode IODPFA network models, but, once again, learning in these more general frameworks does not afford similar convergence guarantees as one obtains for learning IODPFAs, and they do not support a type of model analysis corresponding to PLTL model checking.

Various forms of Multi-Agent Markov Decision Processes are another type of models that share some of the features of our networks of IODPFAs (Guestrin et al., 2001; Proper and Tadepalli, 2009). They, too, are modular models of components that combine nondeterministic inputs (or actions) with probabilistic state transitions. Main objective in this context, however, is to find policies for maximizing a reward function, whereas our models are intended for the analysis of stochastic systems without a reward structure.

6. Conclusion

We have introduced IOPDAF networks as a new modeling framework that extends automata learning methods to complex, structured domains, and the kind of network data often associated with statistical relational learning techniques. Main purpose of the present paper was to demonstrate the general feasibility and potential benefits of the approach, and especially, to illustrate the possibilities of model analysis using model checking techniques. The results presented in this paper, obviously, are quite preliminary, and future work will be directed at testing these methods on more substantial application scenarios.

Acknowledgments

We are grateful to Sergey Kirshner for his help with the Ceará dataset.

References

G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of POPL*, pages 4–16, 2002.

D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

C. Baier and J. P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and Y. Wang. Developing uppaal over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.

Y. Bengio and P. Frasconi. An input output hmm architecture. In *Proceedings of NIPS 7*, pages 427–434, 1994.

R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings of Second International Colloquium on Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. 1994.

Yingke Chen, H. Mao, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen. Learning markov models for stationary system behaviors. In *Proceedings of the 4th NASA formal method symposium (NFM)*, pages 216–230, 2012.

C. de la Higuera. *Grammatical Inference — Learning Automata and Grammers*. Cambridge University Press, 2010.

Z. Ghahramani and M. Jordan. Factorial hidden markov models. *Machine Learning*, 29: 245–273, 1997.

C. Guestrin, D. Koller, and R. Parr. Multiagent planning with factored mdps. In *Proceedings of NIPS 14*, pages 1523–1530, 2001.

T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

S. Hochreiter and M. Mozer. A discrete probabilistic memory model for discovering dependencies in time. In *Proceedings of International Conference on Artificial Neural Networks (ICANN'01)*, pages 661–668, 2001.

A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1), 1990.

K. Kersting, L. De Raedt, and T. Raiko. Logical hidden markov models. *J. of Artificial Intelligence Research*, 25:425–456, 2006.

M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. Symbolic model checking for probabilistic timed automata. In *Proceedings of Joint International Conferences on Formal Modelling and Analysis of Timed Systems, and Formal Techniques in Real-Time and Fault-Tolerant Systems (FORMATS/FTRTFT)*, volume 3253 of *Lecture Notes in Computer Science*, pages 293–308, 2004.

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591, 2011.

H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen. Learning probabilistic automata for model checking. In *Proceedings of the eighth international conference onQuantitative Evaluation of Systems (QEST)*, pages 111 –120, 2011.

H. Mao, Y. Checn, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen. Learning markov decision processes for model checking. In *Proceedings of the first workshop on Quantities in Formal Methods (QFM),to appear*, 2012.

Scott Proper and Prasad Tadepalli. Multiagent transfer learning via assignment-based decomposition. In *In Proceedings of the International Conference on Machine Learning and Application (ICMLA'09)*, pages 345–350, 2009.

K. Sen, M. Viswanathan, and G. Agha. Learning continuous time markov chains from sample executions. In *Proceedings of the first international conference on Quantitative Evaluation of Systems (QEST)*, pages 146–155, 2004.

G. W. Taylor and G. E. Hinton. Products of hidden markov models: It takes n>1 to tango. In *Proceedings of UAI'09*, 2009.