Seminarpaper

Mario Wagner, 0730223

Graz, am January 8, 2017

Contents

List of Figures

List of Tables

1 Introduction

In the industry, model-driven development techniques are getting more and more important, as you can use them for simulation, documentation, model-checking, performance evaluation and many other things. The main problem with these techniques is the part where the model is constructed, because it is often costly and time consuming. Machine Learning techniques try to ease this by automatically constructing a high level model of an unknown system. These techniques are classified into two different categories, supervised learning and unsupervised learning approaches.

Supervised learning is the task of inferring knowledge from a fully labeled training data. In contrast, unsupervised learning is the task of inferring knowledge from an unlabeled dataset. In real-world scenarios, the usual case is to deal with gigantic datasets that are only partially labeled. These real-world datasets are difficult to deal with, because performing supervised learning is not possible due to lack of training data, and doing unsupervised learning is not cost-effective or beneficiary considering accuracy and precision. The common ground of supervised and unsupervised learning is the task of inferring knowledge from a small amount of labeled data and a huge amount of unlabeled training data; i.e., *semi-supervised learning*. In other words, semi-supervised learning is making use of that small amount of data (i.e., supervised learning) to do unsupervised learning considerably more accurate.

Active learning is a special case of semi-supervised learning in which the labeled data is incrementally provided by a continuous interaction between the *learner* and an *oracle*. The oracle can be a user, a system, or a teacher that provides a certain level of insight to supervise the learner by letting it to actively query for labels of unknown data. When the oracle is a system, actively querying it only means the learner is iteratively providing the system with input alphabet and asking for the output alphabet.

A popular active learning algorithm was proposed by Angluin [1] in 1987 and is called L^*. In this context, the *learner* efficiently learns an unknown regular set from an *oracle* by repeatedly asking questions and saving the findings in an observation table. There are two types of questions the learner can ask the oracle. First, she can ask whether a given string is a member of the unknown set (i.e., membership query) to which the oracle replies with yes or no. Second, a conjecture can be made and the oracle answers if the set is correct (i.e., equivalence query) or incorrect. In the later case, a counterexample is provided. The oracle can also be substituted by a random sampling *teacher*. The outcome of the algorithm is a minimal deterministic finite automaton (DFA).

Although L^* is a quite useful automata learning approach, there are also issues with it; first of which is while using L^* the learner either learns the correct system or nothing. Next issue with L^* is actually stated earlier, that is, when it is impossible for some systems to perform some queries for various reasons (e.g. not being cost-effective, safety critical behaviour, expensive overheads, randomly changing behaviour, and etc.). Moreover, there are some difficulties when trying to learn deterministic model of an unknown system; one of which is its randomly changing behaviours. For these reasons, sometimes it is better to construct the stochastic or nondeterministic model of the system under learn by sufficiently observing the system's behaviour. Similarly, passive learning (i.e.,

unsupervised learning) is recently used to learn the somewhat correct model of the system under learn out of its execution traces that represent the actual behaviour of the system.

In this paper, we look into *AALERGIA* [2], which is an extension of the well-known ALERGIA algorithm [3]. This approach learns a stochastic model by using only previously observed patterns in execution traces of the system (passive learning) for it cannot or may not be queried as in Angluin's algorithm. In contrast to statistical model checking, *AALERGIA* extracts an explicit probabilistic model, which one can use to verify a large class of properties without the need to re-sample the system. It is crucial to note that learning these kinds of models can represent different views of the underlying system, depending on the input and output actions. Therefore, if we tailor the observations to the properties of interest, our verification will be simpler and more efficient.

The paper is subdivided into eight parts. The next section is dedicated to some preliminaries, explaining Markov models, the PRISM tool along with the used models and the AALERGIA algorithm, which we are going to study in detail. Section 3 covers related work concerning probabilistic machine learning, model checking and the PAutomaC challenge. Following that, we briefly describe the implementation of the AALERGIA algorithm in Python in section 4. Section 5 explains the IEEE 802.15.4 PRISM model we use as case study. Next, we describe how the traces are generated with the PRISM model and the following evaluation in section 6 and section 7 respectively. Finally, in section 8, we conclude our findings.

2 Preliminaries

2.1 Markov Models

Assuming Markov property, which states future state of the system is only dependant on the current state and not on the events that occurred before, Markov model is a stochastic model capable of depicting randomly changing systems. There are different types of Markov models, some of which we are going to study in this section.

2.1.1 Deterministic Labeled Markov Chain

Markov chain is the simplest variation of Markov models used to model autonomous systems whose states are fully observable. A labeled Markov chain is a tuple $M = (Q, AP, \pi, \tau, L)$ where Q is a finite non-empty set of states, AP a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and π is the initial probability distribution as follows

$$\pi : Q \to [0, 1] \text{ s.t. } \sum_{q \in Q} \pi(q) = 1,$$

and τ is the transition probability function as follows:

$$\tau : Q \times Q \to [0, 1] \text{ s.t. } \forall q \in Q \cdot \sum_{q' \in Q} \tau(q, q') = 1.$$

A labeled Markov chain is deterministic (DLMC) if (1) there exist an initial state $q_0 \in Q$ such that $\pi(q_0) = 1$, and (2) for all $\sigma \in \Sigma$ and from all states $q \in Q$ there exists at most one destination state labeled with σ that is $\tau(q, q') > 0$ and $L(q') = \sigma$.

2.1.2 Discrete-Time Markov Chain

A Discrete-Time Markov Chain (DTMC) is a tuple $M = (Q, q_0, AP, \tau, L)$ where Q is a finite non-empty set of states, $q_0 \in Q$ is the initial state, AP a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and τ is the transition probability function as follows:

$$\tau : Q \times Q \to [0, 1] \text{ s.t. } \forall q \in Q \cdot \sum_{q' \in Q} \tau(q, q') = 1.$$

DTMC is actually representing a stochastic processes in discrete time $n \in \mathbb{N}$ as a sequence of random variables $Q = q_n$. We refer to q_n as the state of process at time n, with q_0 denoting the initial state. If the random variable q_i takes values in discrete space such as integers $\mathbb{Z}$, then the we are dealing with a discrete-valued stochastic process.

2.1.3 Markov Decision Process

Markov Decision Process (MDP) is another variation of Markov models, used to model non-autonomous (controlled) systems whose states are fully observable. An MDP is an extended DTMC such that it allows both probabilistic and non-deterministic behaviour. MDP is a tuple $M = (Q, q_0, AP, \tau, L)$ where Q is a finite non-empty set of states, q_0 is the initial state, AP is a finite non-empty set of atomic propositions, $\Sigma = 2^{AP}$ is a finite alphabet, $L : Q \to \Sigma$ is a state labeling function, and $\tau : Q \times AP \times Q \to [0, 1]$ is the partial probabilistic transition function defined for all q and σ where $\sigma \in L(q)$.

2.2 PRISM Model Checker

When dealing with deterministic models, it is easy to associate states with strings. Despite the fact that it is not a one-to-one association, it is an effective lexicographical representation. It is not only possible to identify a particular state by having a string belonging to a set of all strings emanating from it but it also is possible to denote an arbitrary string by a particular state. A path over the states $q_0, \ldots, q_n$ of a state labeled deterministic Markovian model M gives rise to a trace $s = \sigma_1, \ldots, s_n$ whose occurrence probability is defined by $\mathcal{P}_{\mathrm{M}}(s)$ as:

$$\mathcal{P}_{\mathrm{M}}(s) = \prod_{i=0}^{n-1} \tau(q_i, q_{i+1}),$$

Meanwhile, for non-deterministic Markovian models such as MDP it is also possible to perform Monte-Carlo simulation to estimate the occurrence probability of a trace.

PRISM model checker harnesses above described approaches to verify the correctness of stochastic properties of randomly changing systems over their stochastic models. These stochastic properties can be formally formulated by Probabilistic LTL which we are going to study in the next subsection.

2.2.1 Probabilistic Linear Temporal Logic

Modal and temporal properties of Markovian models are difficult to formulate because they represent randomly changing systems. Meanwhile, LTL is a modal temporal logic best suited to formally express the properties of reactive systems. In this section, we study a probability extended LTL that is suited to formulate the properties of Markovian models.

Linear temporal logic over set of atomic propositions AP is defined by the following syntax:

$$\varphi ::= \mathbf{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid \mathbf{G}\,\varphi \mid \mathbf{F}\,\varphi \mid \varphi_1\,\mathbf{U}\,\varphi_2 \qquad (a \in AP)$$

The syntax of Probabilistic LTL (PLTL) is defined as:

$$\phi ::= \mathcal{P}_{\bowtie \mathrm{r}}(\varphi)$$

where φ is an LTL property, $\bowtie$ is a comparison operator $\{\leq, <, =, \geq, >\}$ and $r \in [0, 1]$ is a confidence threshold. Semantic of a PLTL can be defined over a deterministic Markovian model M as:

$$M \models \mathcal{P}_{\bowtie r}(\varphi) \text{ iff } \mathcal{P}_{\mathrm{M}}(\varphi) \bowtie r,$$

where $\mathcal{P}_{\mathrm{M}}(\varphi)$ denotes $\mathcal{P}_{\mathrm{M}}(\{s \mid s \models \varphi\})$ meaning a string belonging to M models φ.

2.2.2 Dice Model

Since we are going to explain the *AALERGIA* implementation in detail, we are going to use two different running examples, which help us to explain the way the algorithm works. The first and simpler example is a probabilistic algorithm, the dice model[4]. We are mainly using it for the explanation of the frequency prefix tree acceptor (FTPA) creation in the *AALERGIA* algorithm. Figure 1 shows the states and the probability of the state transitions. We start at an initial state S and we toss a coin at each step, giving us a 50 percent chance of transiting to each of the two following states. The algorithm ends at the leafs of the tree, which represent the values of the dice.

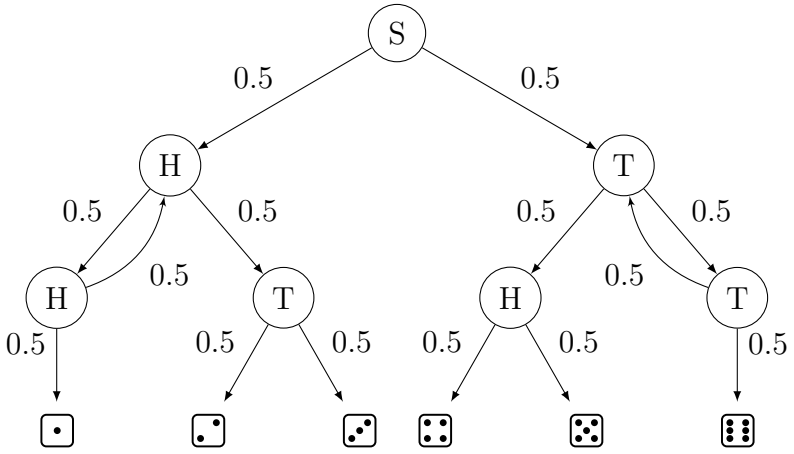

Figure 1: The outcome of a toss of a die generated by randomly throwing a fair coin.

The source code of the dice PRISM model is shown in figure 2. At the beginning, the code indicates that it is describing a discrete-time Markov chain (DTMC). The source code shows a single PRISM module called dice. The variable s, which represents the state of the system, can take the values from 0 to 7 and is initialised with the value 0 (line 6). The variable d, which represents the value of the dice, can range from 0 to 6 and is also initialised with the value 0, meaning that the dice has no value yet (line 8). Each line depicts a state and the probabilities of the transitions, e.g. line 11 shows there is a probability of 0.5 of transiting to state 3 or state 4 if we are currently in state 1. The model ends if a value is assigned to the dice, as there are no further transitions from that point on.

```
dtmc

module dice

  // local state
  s : [0..7] init 0;
  // value of the dice
  d : [0..6] init 0;

  [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
  [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
  [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
  [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
  [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
  [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
  [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
  [] s=7 -> (s'=7);

endmodule
```

Figure 2: The PRISM Dice Model

2.2.3 Herman's Self-Stabilisation Algorithm

The second running example we are going to use in this paper is a PRISM model of Herman's self-stabilising algorithm, as described in [5]. The algorithm guarantees that the system recovers from faults in a certain amount of time. Although it is certain that the system self-stabilises eventually, it is very hard to analyse the maximum execution time, which depicts the worst case scenario in this case.

Herman's algorithm works with a network of an odd number of identical processes in a ring, ordered anticlockwise. The processes work synchronously and possess a token, which is passed around the ring. The result of a random coin toss decides whether the process keeps the token or passes it on to its neighbour. If a process holds two tokens, they are eliminated. The result is a stable system, where each process has exactly one token and the tokens are passed around forever in the ring. In order to analyse the correctness and the performance of the proposed model, the authors use the tool PRISM Model Checker and invoke probabilistic model checking methods on the model.

The proposed methods worked well and proved the correctness of the algorithm. Their findings on the maximum execution time showed, that having 3 tokens, spaced evenly around the ring, always produced the worst-case behaviour. It is stated that probabilistic model checking is extremely useful for checking models of this kind, because simple tools, languages and techniques are provided to model the system. Also, an exhaustive analysis can be done, as opposed to other simulation techniques such as the Monte Carlo simulation. A disadvantage is the restriction of probabilistic model checking to finite state models. Furthermore, the size of the models under analysis are limited to the amount of time and space available.

The source code of the model is shown in figure 3. In this case, seven processes are used in the ring. The step notation in line 8 forces all processes to synchronise. The state of each process is depicted by a single two-valued variable ($x1$,$x2$,$x3$). Initially, $x1 = x2 = x3 = 1$. The implementation of this model is straightforward and the output of it (the traces of the system) are used as input for *AALERGIA*.

```
1  dtmc
2
3  // module for process 1
4  module process1
5
6    x1 : [0..1] init 1;
7
8    [step] (x1=x3) -> 0.5 : (x1'=0) + 0.5 : (x1'=1);
9    [step] !(x1=x3) -> (x1'=x3);
10
11 endmodule
12
13 // add further processes through renaming
14 module process2 = process1[x1=x2, x3=x1 ] endmodule
15 module process3 = process1[x1=x3, x3=x2 ] endmodule
```

Figure 3: The Prism Herman 3 Model

2.3 AALERGIA

ALERGIA[3] was introduced by Rafael C. Carrasco and Jose Oncina in 1994. They proposed an algorithm which is capable of identifying any stochastic deterministic regular language by using a prefix tree acceptor and merging the states in $O(|S|^3)$, where $|S|$ is the size of the sample. This is especially useful for learning automaton in realistic situations. They based their approach on previous work from Oncina[6]. ALERGIA has been extended in many ways, one of many is called *AALERGIA*[2].

The reason for creating AALERGIA was to show that the methods for learning probabilistic automata can be adapted for learning Markov Chain system models for verification. The algorithm also leads to stronger consistency for learning in the limit than previous implementations and the authors analyse how the convergence of the learned model relates to the convergence of probability estimates for system properties. This enables us to use the learned model for probabilistic linear temporal logic (PLTL) model checking.

However, there is still room for improvement, as shown by the winning team[7] in a competition called PAutomaC[1]. They introduced a new criterion for merging states based on marginal probability by greedily merging states if it increases the probability of the automaton.

[1] hai.cs.umbc.edu/icgi2012/challenge/Pautomac/index.php

In this paper, we take *AALERGIA* and the provided MATLAB implementation[2], implement it in Python and experiment with it. In order to get a better idea of the way the implementation works, we will describe it in detail and a running example for better replicability is used throughout this section. The implementation of the algorithm consists of several parts, which are depicted in the figure 4.

Figure 4: The Flow Diagram AALERGIA

2.3.1 Training Data

The data used to learn the Deterministic Labeled Markov Chain (DLMC) is supplied in the form of comma separated value files, which contain the training set and the alphabet to be used. The alphabet consists of a finite 1×N-cell array, where each column contains one letter in the alphabet. The training set consist of a finite N×1-cell array, where each row contains sequences of symbols generated from the model, separated by commas.

Dataset Generator It is possible to generate datasets by using the command line version of PRISM. With this, it is feasible to generate random paths through the model by specifying the model file, an output file and the path to be generated (for instance, the amount of steps to be processed). The output file can then be used as input file for the *AALERGIA* implementation.

The running example we are mainly going to use is a self-stabilizing ring network with 3 processes. Table 1 shows the beginning of the training set, generated by the PRISM modelchecker, which contains 50 Sequences in total. In our case, each symbol in the training set represents the states in the ring at a given moment. For example, the symbol 001 shows that process 3 is in the state 1 and process 1 and 2 are in the state 0.

	1
1	000,001,
2	000,011,100,010,101,010,001,100,010,101,
3	000,101,
4	000,111,010,001,110,011,101,010,001,110,011,
5	000,111,011,101,110,001,
6	000,101,010,101,110,011,100,011,101,110,

Table 1: The Training set

[2]http://mi.cs.aau.dk/code/aalergia

Table 2 shows the alphabet, which consists of 8 symbols from 000 to 111.

	1	2	3	4	5	6	7	8
1	000	001	010	011	100	101	110	111

Table 2: The Alphabet

2.3.2 Frequency Prefix Tree Acceptor

After importing the trainings set and the alphabet from the files, the Frequency Prefix Tree Acceptor (FPTA) is created. A deterministic frequency finite automaton (DFFA) is a tuple $A = \langle \Sigma, Q, I_{\text{fr}}, F_{\text{fr}}, \delta_{\text{fr}}, \delta \rangle$ where Σ = the finite alphabet, Q = the finite set of states, I_{fr} = the initial state frequencies, F_{fr} = the final state frequencies, δ_{fr} = the frequency transition function and δ = the transition function.

Figure 5 shows how a simplified FPTA for the in section 2.2.2 described dice model looks like. The sample S is a multiset of size 20. S = {(SH,1), (SHHH,1), (SHHH⚀,1), (SHH⚀,2), (SHH⚀⚀,1), (SHT,2), (SHT⚁,1), (SHT⚁⚁,1), (SHT⚂,1), (ST,1), (STH,2), (STH⚃,1), (STH⚄,1), (STT,2), (STT⚅,2)}. The letter H stands for heads, the letter T stands for tails and ⚀ to ⚅ stands for the different dice values. The numbers in the states indicate the count of the occurrences of the event, whereas the numbers on the transitions illustrate the count of all events below it.

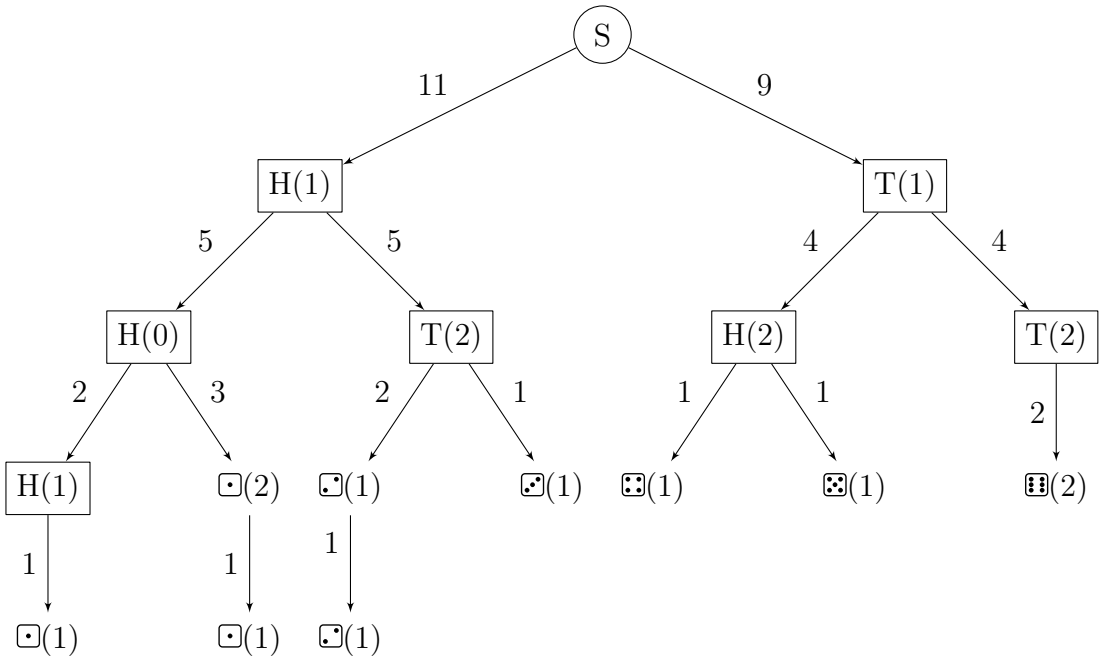

Figure 5: The FPTA for the DICE model

Algorithm 1 shows how the creation of the FPTA is implemented in the *AALERGIA* package. The code starts by sorting the training set and by removing double occurrences

(line 3). After that, a for-loop iterates through the sorted strings and splits them at each comma (line 6 - 15). This creates all the prefixes from the training set and adds them to the cell array named *prefix*, as shown in Table 3.

	1
1	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,001,
2	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,010,
3	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,100,
4	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,011,
5	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,100,
6	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,001,
7	000,000,010,001,100,011,101,110,001,110,001,110,011,101,010,
8	000,000,010,001,100,011,101,110,001,110,001,110,011,101,
9	000,000,010,001,100,011,101,110,001,110,001,110,011,
10	000,000,010,001,100,011,101,110,001,110,001,110,
11	000,000,010,001,100,011,101,110,001,110,001,

Table 3: The Prefixes

In our example, the dimension of *prefix* is 494x1 and contains all unique prefixes of the training set for further evaluation. The cell array *prefix* is sorted by width-first-sort (line 16). The shortest strings are at the beginning of the array and the largest at the end of the array, as shown in Table 4.

	1
1	000,
2	000,000,
3	000,001,
4	000,010,
5	000,011,
6	000,100,
7	000,101,
8	000,110,
9	000,111,
10	000,000,010,
11	000,000,100,
12	000,000,111,
13	000,001,100,
14	000,001,110,

Table 4: The Sorted Prefix Array

The function *find_predecessor* searches through the *prefixes* array and saves all the predecessors in the corresponding array (line 16). In the end, a for-loop over all the

Algorithm 1 Create the FPTA

Input: the $\sum$ and a set of strings S (training data)
Output: a DFFA A and a sorted set of strings U
$U \leftarrow \text{sort}(S)$
prefixes $\leftarrow U$
$F_{fr} \leftarrow \text{string\_count}(U)$
for `u` $\in$ `U` **do**
    index $\leftarrow$ find positions of "," in u
    while *index* > 0 **do**
        substr $\leftarrow$ *substring(1:index)*
        if *substr* $\notin$ *prefix* **then**
            prefixes(end+1) $\leftarrow$ *substr*
        end if
        index $\leftarrow$ *index-1*
    end while
end for
prefixes $\leftarrow$ width_first_sort(*prefixes*)
predecessor $\leftarrow$ find_predecessor(*prefixes*)
for `p` $\in$ `prefixes` **do**
    sym $\leftarrow$ *get_symbol(p)*
    pre $\leftarrow$ *predecessor(p)*
    freq_trans_matrix^1 *(pre, sym)* $\leftarrow$ *predecessor(p)*
    frequency_transition $\leftarrow$ *freq_trans_matrix*^{2} *(pre, sym)*
    frequency_transition $\leftarrow$ *frequency_transition* + *frequency(p)*
    while *pre* **do**
        update_freq_trans_matrix(*p, pre*)
        pre $\leftarrow$ *predecessor(p)*
    end while
end for
init_states $\leftarrow$ *freq_trans_matrix*^{1}*(1, all)*
return A

prefixes is executed in order to find all transitions and their frequencies respectively. The values are saved into the *freq_trans_matrix* (line 18 - 28). This Matrix contains 2 elements: the first element is the transition matrix, which contains all the transition between the states. The second element is a matrix, which contains all the frequencies between the states.

The returned object is a *DFFA* class, representing the DFFA (line 30). It has the following members:

- a finite set of states
- the labels of the states
- the alphabet
- the initial state
- the initial state frequency
- the final state frequency
- the frequency transition matrix
- RED states: a finite set of states, which have already been determined to be in the final DLMC
- BLUE states: a finite set of states, which need to be tested for compatiblity

Table 5 shows how the first element of the frequency transition matrix (the transition matrix) is structured. The columns represent the 8 symbols, the rows the different states. For example, row number 2, column 1 shows the number 3. This indicates, that state 2 and symbol 1 lead to state 3. The second element of the frequency transition matrix has the same dimensions as the first element and follows the same logic, but instead of transitions it contains the frequencies.

	1	2	3	4	5	6	7	8
1	2	0	0	0	0	0	0	0
2	3	4	5	6	7	8	9	10
3	0	0	11	0	12	0	0	13
4	0	0	0	0	14	0	15	0
5	0	16	0	0	0	17	0	0

Table 5: The Transition Matrix

2.3.3 Normalization

The algorithm 2 shows how a DFFA is converted into a deterministic probabilistic finite automaton (DPFA). The computation is pretty straightforward. As input, the algorithm takes a well defined DFFA, that means that the number of strings entering and leaving a given state is identical. The probabilistic transition matrix (line 5) is identical to the frequency transition matrix, but its second element contains the probabilities of the transitions instead of the frequencies of the transitions.

The algorithm searches through the frequency transition matrix, sums up the frequencies of the transitions and adds them to the frequency of the state itself in order to get the total number of frequencies (line 7). Following that, the frequency of each transition and of the state is divided by the total number of total frequencies, which results in the corresponding probabilities for the transitions. (line 9).

Algorithm 2 Create a DPFA from a DFFA

1: **Input:** a well defined DFFA
2: **Output:** corresponding DPFA
3: **for** $q \in Q$ **do**
4: $freq\_state \leftarrow dffa.finalStateFrequency(q)$
5: $ptm^1 \leftarrow dffa.frequencyTransitionMatrix^1$
6: $freq\_trans \leftarrow dffa.frequencyTransitionMatrix^2(q,\ all)$
7: $freq\_total \leftarrow freq\_trans + freq\_state$
8: **if** $freq\_total > 0$ **then**
9: $ptm^2(node,\ index) \leftarrow \frac{freq\_trans}{freq\_total}$
10: $fsp(q) \leftarrow \frac{freq\_state}{freq\_total}$
11: **end if**
12: **end for**
13: **return** $DPFA(dffa,\ fsp,\ ptm)$

2.3.4 Golden Section Search

First, we need to search for a left and right border of ε in order to get a region for our golden section search. This is done by the following algorithm 3. In order to save space, the algorithm only explains how to find the left border of the region. The right border is found the same way, with some minor modifications. The algorithm iteratively calculates BIC-scores for the values of ε. For each run of the while-loop, the ε is multiplied by 0.5 until a left border is found. If the new ε produces a Bayesian Information Criterion(BIC)-score larger than the previous ε (line 8), it essentially means that we found a right border, but the left border is further to the left and we need to search again. If the new score is the same as the old score (line 13), we search again. If the value of the score doesn't change for 5 consecutive tries, we break the loop and take the last value of ε as our left border. Last but not least, if the new score is smaller than the old score (line 17), we found the left border and can continue searching for the right border (indicated by the dots in the pseudo-code).

Algorithm 3 Find ε-region

Input: the DFFA
Output: the region for ε, scores, epsilon_values
$\varepsilon \leftarrow 1$
$score \leftarrow$ calculate_BIC_Score(ε, *dffa*)
$new\_\varepsilon \leftarrow \varepsilon * 0.5$
while $new\_\varepsilon > 0$ **do**
    $new\_score \leftarrow$ calculate_BIC_score(new_ε, *dffa*)
    if $new\_score > score$ **then**
        $region\_right \leftarrow \varepsilon$
        $\varepsilon \leftarrow new\_\varepsilon$
        $new\_\varepsilon \leftarrow new\_\varepsilon * 0.5$
        $score \leftarrow new\_score$
    else if $new\_score = score$ **then** ▷ do this max. 5 times
        $\varepsilon \leftarrow new\_\varepsilon$
        $new\_\varepsilon \leftarrow new\_\varepsilon * 0.5$
        $score \leftarrow new\_score$
    else
        $region\_left \leftarrow new\_\varepsilon$
        break
    end if
end while
...
...
return *region_right, region_left*

BIC Score This is a measure that can be used to evaluate the learned model. It creates a score that is calculated on the likelihood minus a penalty term for the model complexity. The BIC score of a given DLMC A given data $S[n]$ is defined in formula 1. Here, $|A|$ depicts the size of A and $N = \sum l_i$ is the total size of the data.

$$BIC(A|S[n]) := log(P_A(S[n])) - 1/2|A|log(N) \tag{1}$$

Table 6 shows the values of the left and right border of ε. The algorithm ran only once for the left border (ε startes with 1 and is halfed every run) and 5 times for the right border, since the scores always stayed the same. Table 7 shows the BIC scores and table 8 shows the corresponding ε values. For example, we calculated the BIC score for $\varepsilon = 0.5$ and the result was $-1.5769e + 03$.

	1	2
1	0.5	64

Table 6: The Search Region

	1	2	3	4	5	6	7	8
1	-840.608	-1.5769e+03	-840.608	-840.608	-840.608	-840.608	-840.608	-840.608

Table 7: The BIC scores

	1	2	3	4	5	6	7	8
1	1	0.5	2	4	8	16	32	64

Table 8: The Epsilon values

After having found the region, the golden section search can begin. Algorithm 4 shows how it is implemented. Basically, the algorithm uses the golden ratio on our 2 borders a1 and a2 and tries to find a maximum (the highest BIC score) between these 2 values. If it exists, the section that contains the maximum is selected as new region and the search is started again, until the recursion reaches depth 3 or the condition at line 11 is fulfilled. At line 7, 8 we calculate the 2 values a1,a2 for our golden ratio, given the region r1,r2. After that, the corresponding BIC scores f1,f2 are calculated(line 9, 10). If the distance between the 2 values a1,a2 is small enough, we stop the recursion and the average score between the 2 values is taken as a good ε (line 12). If f1 is smaller than f2, the maximum has to be between a1 and r2 and a new golden section search is started. If it is the other way around and f1 is larger than f2, the maximum is between r1 and a2 and we start the search with these values. If it happens that the score of f1 and f2 is exactly the same, we need to make sure, that the recursion is not endless (line 24). We check if f1 is larger than the largest score we already calculated. Should this be the case, the maximum is between a1 and a2 (line 27). Lastly, if we do not have any luck and we have not found a maximum so far, we simply start 2 new golden searches for the left (r1, a1) and the right (a2, r2) region respectively (line 34, 35). In the end, the algorithm returns a suggestion for a good ε as shown in Table 9, which we use as input parameter (α) for our merging algorithm.

	1
1	64

Table 9: The Epsilon

Algorithm 4 Golden_section_search

```
1: Input: region, dffa, scores
2: Output: good_eps, scores, epsilon
3: good_eps ← 0
4: r1 ← region(1,1)
5: r2 ← region(1,2)
6: while true do
7:     a1 ← r1 + 0.382(r2-r1)
8:     a2 ← r2 + 0.618(r2-r1)
9:     f1 ← calculate_BIC_score(a1,dffa)
10:    f2 ← calculate_BIC_score(a2, dffa)
11:    if |a1-a2| < 0,00001 then
12:        good_eps ← (a1 + a2) / 2
13:        return
14:    end if
15:    if f1 < f2 then
16:        region ← region(a1, r2)
17:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
18:        return
19:    else if f1 > f2 then
20:        region ← region(r1, a2)
21:        good_eps, scores, epsilon ← Golden_section_search(region, dffa, scores)
22:        return
23:    else
24:        if recursion_depth = 3 then
25:            return
26:        end if
27:        if f1 > max(scores) then
28:            new_region ← a1, a2)
29:            good_eps, scores, epsilon ← Golden_section_search(new_region, dffa,
    scores)
30:            return
31:        else
32:            regionL ← region(r1, a1)
33:            regionR ← region(a2, r2)
34:            good_eps, scores, epsilon ← Golden_section_search(regionL, dffa, scores)
35:            good_eps, scores, epsilon ← Golden_section_search(regionR, dffa, scores)
36:            return
37:        end if
38:    end if
39: end while
```

2.3.5 Merging

After having calculated a suitable α value, the *AALERGIA* algorithm can finally do its work. RED is a set of states, that has beeny revised (and is included in the final automaton), whereas BLUE is the set of states that we need to look at. The algorithm starts by assigning the initial state to the RED set(line 3), while the successors of the initial state are assigned to the BLUE set(line 4). After that, we loop through all the elements of the BLUE set and check if we can merge them. This happens, if the AALERGIA_compatible criterion (line 14) is satisfied. If the merge happens, the size of our final tree is reduced, if not, the state gets promoted to the RED set (line 21) and is excluded from the BLUE set (line 24). In the end, we get a merged DFFA, which is reduced in size but still provides a good BIC score. Further details on the AALERGIA_compatible criterion can be found at [2]

Algorithm 5 AALERGIA

```
1:  Input: a DFFA, a DPFA, the alpha
2:  Output: the merged DFFA
3:  RED ← dffa.initial_state
4:  BLUE ← freq_trans_matrix^1(dffa.initial_state, all)
5:  promote ← 1
6:  while BLUE is not empty do
7:      BLUE ← sort(BLUE)
8:      q_b ← BLUE(1)
9:      BLUE.remove(q_b)
10:     same_label_ind ←find_same_labels(RED, q_b)
11:     for ind ∈ same_label_ind do
12:         q_r ← RED(ind)
13:         threshold ← calculate_compatible_params(dffa, q_r, q_b, alpha)
14:         if AALERGIA_compatible(dffa, dpfa, q_r, q_b, alpha, threshold) then
15:             dffa_merged ← AALERGIA_merge(RED, BLUE, q_r, q_b)
16:             promote ← 0
17:             break
18:         end if
19:     end for
20:     if promote then
21:         RED ← q_b and RED
22:     end if
23:     successors ← dffa_merged.freq_trans_matrix^1(RED, all)
24:     BLUE ← all successors not in RED
25: end while
```

3 Related Work

This section summarizes previous and existing approaches in science which are relevant to this paper. Furthermore, the previously mentioned PAutomaC competition, which was conducted online and focussed on learning of probabilistic automatons, is described and the implementations of the three best teams are presented.

3.1 Probabilistic machine learning

In order to learn probabilistic models, which are perfectly suited for learning real systems in a physical environment, Sen et al. [8] adapted the algorithm of [3] and presented a novel approach for learning continuous time Markov chain models. However, one limitation of their work is that the algorithm doesn't scale up for large underlying blackbox models. Mao et al. [2] wrote the paper that presented *AALERGIA* and proved a strong theoretical and experimental consistency with their results. One year later, Mao et al. [9] wrote a paper about learning Markov decision processes for model checking, however, their results need further research, as the algorithm is not suitable for routine use yet.

3.2 Model checking

Model checking is used to make sure that certain properties hold in a given model of a system. It is used for exhaustive and automatic verification of a property in a finite state system. PRISM model checker has been used for model checking for many network protocols successfully: Duflot et al. [10] used probabilistic model checking to compute the performance of different device discovery scenarios in the Bluetooth wireless communication protocol. They managed to perform an exhaustive low level analysis and examined the run times (best and worst) of the protocol, but were limited by the size of the probabilistic models that were needed. In order to cope with this problem, a combination of simulation or abstraction and symmetry reduction methods are proposed. Kwiatkowska et al. [11] performed probabilistic model checking techniques on the CSMA/CD communication protocol and the IEEE1394 FireWire root contention protocol. They give an overview of probabilistic temporal-logic properties of wireless networks [12]. Matthias Fruth [13] modelled the IEEE 802.15.4 standard in PRISM model checker and verified correctness properties, like the successful transmission or the amount of collisions, on it.

3.3 PAutomaC online challenge

The PAutomaC online challenge [14] was conducted in 2012 and its goal was to find the best performing model/algorithm for learning probabilistic automatons. The competition provided different distribution sets, generated from various models with different sizes and ranges. The participants had access to artificially generated training and test sets, which they could use to learn the corresponding string distributions. The learned data structure was not evaluated, so the use of any learning method was allowed. Furthermore, the competition included two real world data sets. As stated on their website,

the competition had 38 participants, but only five managed to score points. The clear winner of the competition is team Shibata Yoshinaka [7].

Team Kepler [15] used variable length N-grams to learn the automatons. They created a tree structure, which they used to store the contexts with variable length. After having inserted all the sequences into the tree, they pruned it with the help of the Kullback-Leibler divergence between the leaf and the parent. They managed to produce better results than the 3-gram algorithm, which was provided by the competition as a baseline.

Another interesting approach was submitted by team Hulden [16]. The team implemented a tool called 'treba', which uses a variant of the Baum-Welch algorithm to train a probabilistic finite automaton. They put strong focus on the numerical stability and the parallelization of the algorithm. Their contribution managed to gain a few points in the competition and performed best on dense instances with few states [14].

The winning team Shibata Yoshinaka [7] obtained the best perplexity values on most instances. Perplexity measures how well a probability distribution predicts a sample. The lower it is, the better it is at predicting the sample. The more data team Yoshinaka had, the better their perplexity values became, which indicates, that they make good use of more data. They implemented different algorithms but concluded, that their version of *Collabsed Gibbs Sampling (CGS)*, which is a Bayesian method for sampling, works best on the data. For PDFAs they implemented a state-merging algorithm, which is based on the marginal probability. Among the presented algorithms, CGS performed best, but had quite high computational cost. It performs best when there are many strings (100k) or the target contains few (< 21) states.

4 Implementation of the AALERGIA algorithm in Python

The functionality and the structure of the *AALERGIA* algorithm was covered in section 2 in detail. We used the existing implementation in Matlab and implemented it in Python. Python is open source, focusses on code readability and is widely used, therefore, has a great community behind it. In order to support the array calculations in the implementation, we included NumPy in the project, a Python package which is used for scientific computing. Even though NumPy is similar to Matlab, there are still a few differences to note, such as the difference in their basic data types (NumPy differs between array type operations and matrix type operations), the difference in indexing (Matlab uses one based indexing, Python uses zero based indexing) and the general purpose of the language (Matlabs purpose is linear algebra, Python is a general-purpose programming language). Since NumPy differs between an array class and a matrix class, we decided to only use array classes in this project. Even though there are differences, many operations behave similar and therefore NumPy is the perfect choice for our implementation.

Since we already covered the functionality of the algorithm, there is no need to plunge into the implementation further. Figure 6 shows the *ALLERGIA* algorithm implemented in Python.

```
def AALERGIA(dffa, alpha,dpfa_orig):
    dffa_merged = copy.deepcopy(dffa)
    initial_state = dffa.initial_state
    dffa_merged.RED = np.append(dffa_merged.RED, initial_state)

    initial_blue_states = dffa.frequency_transition_matrix[0][
    initial_state][:]
    initial_blue_states = initial_blue_states[np.nonzero(
    initial_blue_states)]
    dffa_merged.BLUE = np.append(dffa_merged.BLUE, initial_blue_states)

    while len(dffa_merged.BLUE) > 0:
        dffa_merged.BLUE =  np.msort(dffa_merged.BLUE)
        q_b = dffa_merged.BLUE[0]
        dffa_merged.BLUE = dffa_merged.BLUE[1:len(dffa_merged.BLUE)]
        promote = 1
        labels = dffa.state_labels[dffa_merged.RED]
        state_labels_q_b = dffa_merged.state_labels[q_b]
        label_index = findall(labels, state_labels_q_b)

        for i in range(0, len(label_index)):
            q_r = dffa_merged.RED[label_index[i]]
            thresh = calculate_compatible_parameter(dffa, q_r, q_b, alpha)

            if (AAlergia_compatible(dffa, dpfa_orig, q_r, q_b, 1, 1, alpha
    , thresh)):
                dffa_merged = AAlergia_merge(dffa_merged, q_r, q_b)
                promote = 0
                break

        if promote == 1:
            dffa_merged.RED = np.append(dffa_merged.RED, q_b)

        #build new blue set
        qr_succ = dffa_merged.frequency_transition_matrix[0][dffa_merged.
    RED][:]
        qr_succ = qr_succ[np.nonzero(qr_succ)]

        #gets the new blue data (returns data that is not in intersection)
        difference = np.setxor1d(qr_succ, np.append(dffa_merged.RED,
    dffa_merged.BLUE))
        ia = np.empty(0, dtype = int)

        for i in range(0, len(qr_succ)):
            for j in range(0, len(difference)):
                if (difference[j] == qr_succ[i]):
                    ia = np.append(ia, qr_succ[i])

        dffa_merged.BLUE = np.append(dffa_merged.BLUE, ia)
    return dffa_merged
```

Figure 6: The ALLERGIA algorithm implemented in Python

5 Implementation of the IEEE 802.15.4 PRISM model

In order to test and prove the correctness of our implementation of *AALERGIA* in Python, we decided to learn a simple model of the IEEE 802.15.4 unslotted protocol. The concept of the model was taken from the paper of Wu et al. [17], where they introduced a probabilistic timed automaton (PTA) model, on which they performed model checking. However, since PRISM model checker only provides limited support for PTAs, we cannot generate traces out of them. Since we want to get traces for our algorithm as input, it is crucial to find a way around that issue. Fortunately, it is possible to represent the PTA as a finite state Markov decision process (MDP), using the digital clocks approach presented in the paper of Kwiatkowska et al. [18]. The authors have shown that this semantics preserves probabilistic reachability, which allows us to use model checking on the converted model.

A wireless network control system (WNCS) with two sending modules and one channel module in parallel composition is considered. The two sending modules start their sending process at the same time. They access the channel according to the unslotted CSMA/CA protocol. The protocol uses a mechanism called exponential backoff, where a sender listens to the channel before attempting to send. If the channel is busy, it waits a random amount of time, before retrying. Since we are modelling a MDP, we need to work with discrete time. The smallest unit of time in our model is called backoff period T_b. If the sender wants to send a packet, it needs to initialise the variables NB and BE. The variable NB denotes the number of times a backoff was required due to a busy channel and is bounded by an upper bound called NB_{max}. BE describes the backoff-exponent and is related to the random amount of time the sender waits before it attempts to listen to the channel again. BE is initially set to BE_{min} and the upper bound is BE_{max}.

Figure 7 [17] shows, how the sender (a) and the channel (b) are modelled.

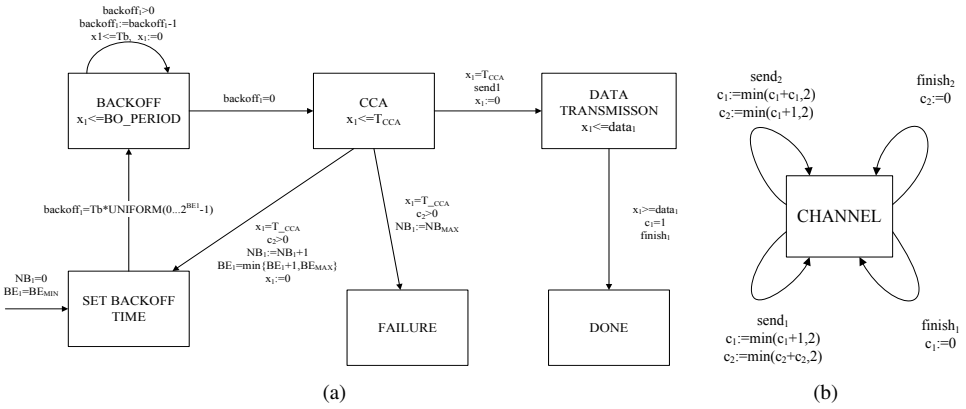

Figure 7: The concept of the sender and the channel (Source: Wu et al., 2014)

The sender starts by setting the backoff time, which is discrete and chosen randomly

between $0\,T_B$ and $(2^{BE_{min}} - 1)\,T_B$. When the time is set, it enters a state where it waits for the expiration of the previously chosen time. After that, the sender enters the state *CCA*, where it listens to the channel and checks if the channel is busy or not. This takes T_{CCA} time, which equals T_B in our case. From there on, three transitions can be taken. If the channel is busy and the number of backoff times *NB* is equal to NB_{max}, the module ends in the state *FAILURE* and stays there. However, if the channel is busy but *NB* is smaller than NB_{max}, the sending module starts with its first state again. Additionally, *BE* is increased by one, which increases the chance for a longer backoff time. *NB* is also increased by one, noting that one more backoff attempt is started. Last but not least, if the channel is idle, the sender can start sending the packet. Once the process is finished, it remains in the state *DONE*.

The channel module is simple as well. It implements two states, *c1* and *c2*, that indicate the channel condition. If $c1 = c2 = 0$, the channel is idle. If $c1 = 1$ or $c2 = 1$, either *c1* or *c2* are sending. If both of the sending modules are attempting to send at the same time, $c1 = c2 = 2$ and the process failed due to a collision.

The source code of the PRISM model for the channel is shown in Figure 8.

```
1
2  module channel
3    c1 : [0..2] init 0;
4    c2 : [0..2] init 0;
5
6    [send1] c1=0 & c2=0 -> (c1'=1);
7    [send2] c2=0 & c1=0 -> (c2'=1);
8
9    // check if something is sent -> collision!
10   [send1] c1=0 & c2>0 -> (c1'=2) & (c2'=2);
11   [send2] c2=0 & c1>0 -> (c1'=2) & (c2'=2);
12
13   // message was sent and channel is clear again
14   [finish1] c1>0 -> (c1'=0);
15   [finish2] c2>0 -> (c2'=0);
16 endmodule
```

Figure 8: The PRISM Channel Model for IEEE 802.15.4 unslotted

The source code of the PRISM model for the sender is shown in Figure 9. The code depicts, in the PRISM coding language, the various guards that are used to decide which states can be reached in the next step. The sender has a clock x, which is increasing during each time step and has a maximum value of *SAMPLING_TIME*. We set the maximum value to $80\,T_B$. In this time, both senders need to finish their transmission, otherwise the deadline is exceeded. The guards where *s=2* model the non determinism of the packet length, varying from 20-30 T_B.

```
module sender1

  NB : [0..NB_MAX] init 0;
  BE : [0..BE_MAX] init BE_MIN;
  BOFF_DELAY : [0..BOFF_DELAY_MAX];
  DL : [0..DL_MAX] init DL_MAX;

  s : [0..5] init 0; // location

  x : [0..SAMPLING_TIME] init 0; // clock

  [] s = 0 & BE=1 -> (1/2):(s'=1) & ...;
  [] s = 0 & BE=2 -> (1/4):(s'=1) & ...;
  [] s = 0 & BE=3 -> (1/8):(s'=1) & ...;
  [] s = 0 & BE=4 -> (1/16):(s'=1) & ...;
  [] s = 0 & BE=5 -> (1/32):(s'=1) & ...;
  [] s = 0 & BE=6 -> (1/64):(s'=1) & ...;

  [time] s=1 & x < TB -> (x'=min(x+1,SAMPLING_TIME));
  [] s=1 & x = TB & BOFF_DELAY > 0 -> (s'=1) & (x'=0) & (BOFF_DELAY'=
   BOFF_DELAY-1);
  [] s=1 & x = TB & BOFF_DELAY = 0 -> (s'=2) & (x'=0);

  [time] s=2 & x < TCCA -> (x'=min(x+1,SAMPLING_TIME));

  [] s=2 & x = TCCA & ch_busy & NB < NB_MAX  -> (x'=0) & (NB' = min(NB+1,
   NB_MAX)) & (BE' = min(BE+1, BE_MAX)) & (s'=0);
  [] s=2 & x = TCCA & ch_busy & NB = NB_MAX  -> (s'=5);

  // Move to next state, set nondeterministic data_length
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-10);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-9);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-8);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-7);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-6);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-5);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-4);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-3);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-2);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-1);
  [send1] s=2 & x = TCCA & ch_free -> (x'=0) & (s'=3) & (DL'=DL_MAX-0);

  [time] s=3 & x < DATA_LENGTH -> (x'=min(x+1,SAMPLING_TIME));
  [finish1] s=3 & x = DATA_LENGTH & ch_busy -> (s'=4);

  [time] s = 4 -> (s'=4); // FINISHED, stay here
  [time] s = 5 -> (s'=5); // FAILED, stay here
endmodule
```

Figure 9: The PRISM Sender Model for IEEE 802.15.4 unslotted

6 Generating traces of the model and learning the model with AALERGIA

In order to generate traces out of the PRISM model we mentioned in the previous section, some reflections and further work needs to be done. It is especially important to think about how the states can be represented in the traces and how the alphabet looks like. The level of detail can be abstracted or more refined, depending on the application of the model. For the final model, the discrete time Markov chain created by *AALERGIA*, only the synchronized time steps in the MDP are essential, because they are equal to steps in the DTMC. This allows us to filter all the steps in the traces that are not related to changes in time.

For this reason, we implemented a small Python tool that calls the PRISM command line tool and generates a certain amount of traces from the model, since PRISM can only run one simulation at a time. In this tool, the output of the model is filtered and written to two CSV-files, one being the alphabet, the other one being the training data (the traces of the system).

When we look at our model, only the states of the two senders and the state of the channel is of importance. This is represented by a symbol that depicts the state of the first sender, the state of the second sender, the channel condition 1 and the channel condition 2 in a row. Every symbol stands for a time step and all the symbols in a trace are seperated by commas. A random example of these traces is shown in table 10. For instance, a symbol like *1301* shows, that sender 1 is in state 1, sender 2 is in state 3, channel condition 1 is in state 0 and channel condition 2 is in state 1. If we look at our model, this means, that sender 2 is sending and sender 1 is waiting. Following this representation, the final state we are looking for is *4400*, as it signals that sender 1 is in state 4 and sender 2 is in state 4 as well, meaning that they finished sending their packet successfully.

	1
1	0000,1100,2100,3110,3110,3110,3210,3110,3110,3110,3110,3110,3110,3210,3110,3110,..
2	0000,1100,1100,1100,1100,1100,1200,1301,1301,2301,1301,2301,1301,2301,1301,2301,...
3	0000,1100,1100,1200,1301,1301,2301,1301,2301,1301,1301,1301,1301,1301,1301,2301,...

Table 10: The Traces of the IEEE 802.15.4 protocol

In order to create the DTMC, *AALERGIA* needs an alphabet as well. The full alphabet contains all the symbols that occur in the traces and is contained in a seperate file. An example is given in table 11.

	1
1	0000
2	1100
3	2100
3	3110
3	3210
3	4301
3	1200
3	2301

Table 11: A Sample of the alphabet of the IEEE 802.15.4 protocol

7 Evaluation

After learning the DTMC model with *AALERGIA*, the results need to be compared with model checking. The initial question Wu et al. [17] tried to answer with probabilistic model checking was, whether the network is stable or not. For this reason, they defined the minimum probability they are interested in as follows:

- *PR = The minimum probability of both stations successfully transmitting their packets within one sampling period*

This can be checked by satisfying the stability specification:

- $\mathcal{P}_{\geq 1-p}$ [true $\mathcal{U}$ (done1 $\wedge$ done2 $\wedge$ z $\leq$h)]

In the following case, we set h (the sampling period) to $80\,T_B$. z depicts the clock, *done1* and *done2* means that sender 1 and sender 2 have finished sending the packet successfully. The specification above essentially expresses PR in PLTL. The packet length is set non-deterministically between 20-30 T_B.

Figures 10-12 show the different test setups and their corresponding results. In each graph, a different parameter of the IEEE 802.15.4 PRISM model is changed. After that, the model is learned and the results are compared. PTA stands for the model Wu et al. created and published in their paper, MDP stands for the model that we build from their paper and DTMC stands for the learned model with *AALERGIA*.

Figure 10 shows how changing BE_{min} affects the stability of the network, while $BE_{max} = 5$ and $NB_{max} = 4$. As Wu et al. stated before, the maximum PR is achieved when $BE_{min} = 3$. The reason for that is simple, if BE_{min} is small, the backoff time will be short and it is very likely that the channel is still busy when trying to send again. Therefore, it is more likely that $NB = NB_{max}$ and the sending process fails. On the other hand, if BE_{min} is too large, it is more likely that the process cannot finish sending in the given timeframe.

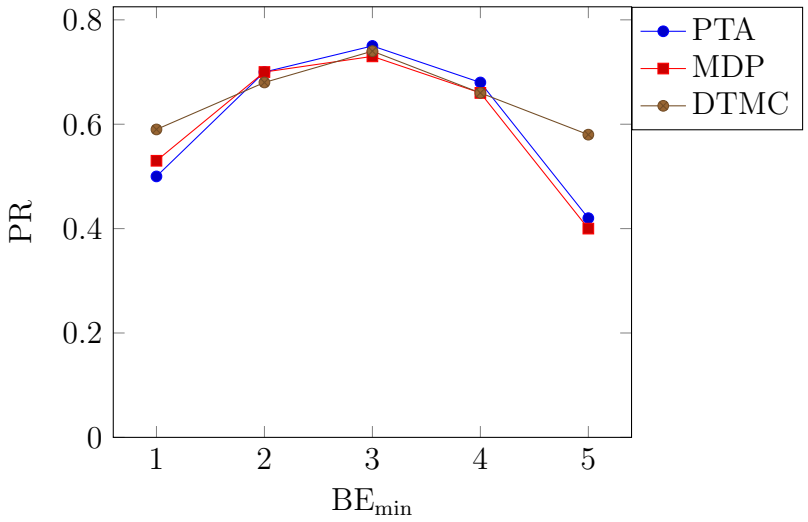

Figure 10: The results while changing BE_{min}

Figure 11 indicates how PR changes, if BE_{max} is modified and $BE_{min} = 3$ and $NB_{max} = 4$. The explanation is similar to figure 10.

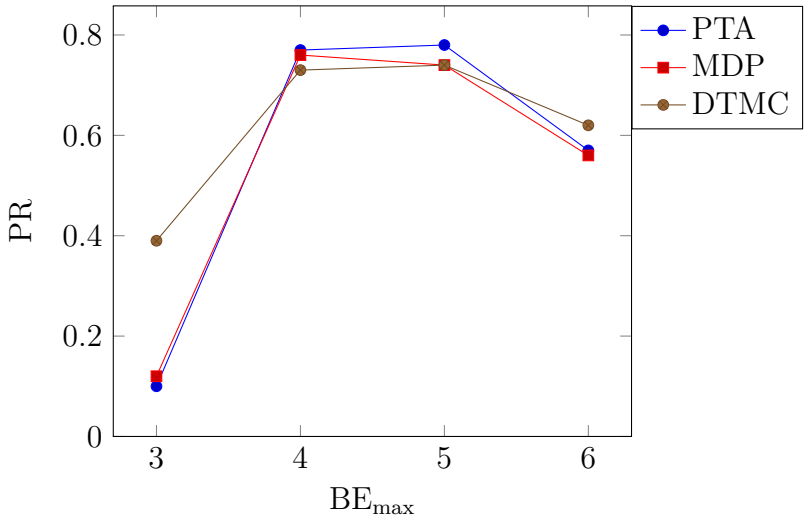

Figure 11: The results while changing BE_{max}

In the last plot (figure 12), we altered NB_{max}, with $BE_{min} = 3$ and $BE_{max} = 5$. When NB_{max} is small, the maximum number of retries are reached very fast and the sending process fails. However, if NB_{max} gets larger, PR levels off quickly, because it exceeds the timelimit.

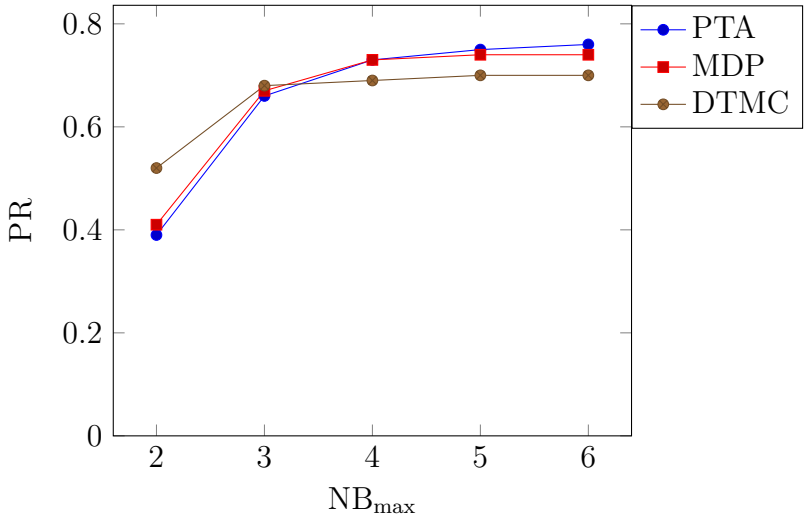

Figure 12: The results while changing NB_{max}

The DTMCs we created with *AALERGIA* were learned from 300 traces of the original MDP models. In order to improve the accuracy, further research in the topic of sampling complexity is needed. It took us between 3-6 minutes to get one result, however, including the trace generation we needed 10 minutes on an Intel Core i7 Quadcore CPU @ 2.00GHz per result. The learned models consist of many more states than the original models (400-500 compared to 20), because we suffer from state-space explosion.

8 Conclusions

The approach generally showed good results, but in some cases, the learned model differs quite a lot (e.g. Figure 11, $BE_{max} = 3$) from the actual model. As of now, we cannot say for sure why this is happening. In order to investigate this further, more research in future work in the topic of sampling complexity is needed. This is going to give us valuable insight into AALERGIA and its limits. We also need to further evaluate the state-space explosion that we get in the final model. One way to deal with that is to sample the DTMC and get a smaller model as a result. We could also tackle this issue with a higher abstraction level, which hides details, but might give us the same results with a lot less states for the final model. Another possible improvement would also be to enhance the result with active learning. If we take advantage of the data acquisition, we can get more useful traces and thus the learned automaton will be more accurate. We can also prevent the fact that, if some behaviour in the model is never executed, it is impossible to learn the right model. Active learning takes care of that problem, as it allows us to trigger any possible behaviour of the system.

References

[1] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[2] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, "Learning probabilistic automata for model checking," in *2011 Eighth International Conference on Quantitative Evaluation of Systems (QEST)*, pp. 111–120.

[3] R. C. Carrasco and J. Oncina, "Learning stochastic regular grammars by means of a state merging method," in *Grammatical inference and applications* (R. C. Carrasco, ed.), vol. 862 of *Lecture notes in computer science Lecture notes in artificial intelligence*, pp. 139–152, Berlin: Springer, 1994.

[4] D. Knuth and A. Yao, *Algorithms and Complexity: New Directions and Recent Results*, ch. The complexity of nonuniform random number generation. Academic Press, 1976.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic verification of herman's self-stabilisation algorithm," *Formal Aspects of Computing*, vol. 24, no. 4-6, pp. 661–670, 2012.

[6] J. Oncina and P. Garcia, "Identifying regular languages in polynomial time," in *Advances in structural and syntactic pattern recognition, Volume 5 of series in machine perception and artificial intelligence*, pp. 99–108, World Scientific, 1992.

[7] C. Shibata and R. Yoshinaka, "The 11th icgi marginalizing out transition probabilities for several subclasses of pfas," *Machine Learning*, pp. 259–263, 2012.

[8] K. Sen, M. Viswanathan, and G. Agha, "Learning continuous time markov chains from sample executions," in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 146–155, Sept 2004.

[9] H. Mao, Y. Chen, M. Jaeger, T. D. Nielsen, K. G. Larsen, and B. Nielsen, "Learning markov decision processes for model checking," *Electronic Proceedings in Theoretical Computer Science*, vol. 103, pp. 49–63, 2012.

[10] M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker, "A formal analysis of Bluetooth device discovery," *Int. Journal on Software Tools for Technology Transfer*, vol. 8, no. 6, pp. 621–632, 2006.

[11] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang, "Symbolic model checking for probabilistic timed automata," *Information and Computation*, vol. 205, no. 7, pp. 1027–1077, 2007.

[12] M. Kwiatkowska, G. Norman, and D. Parker, "Prism: Probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.

[13] M. Fruth, "Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol," in *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'06)*, 2006.

[14] S. Verwer, R. Eyraud, and C. de la Higuera, "Pautomac: A probabilistic automata and hidden markov models learning competition," *Machine Learning*, vol. 96, no. 1-2, pp. 129–154, 2014.

[15] F. Kepler, S. Mergen, and C. Billa., "Simple variable length n-grams for probabilistic automata learning," *Machine Learning*, pp. 249–253, 2012.

[16] M. Hulden, "Treba: Efficient numerically stable em for pfa," *Machine Learning*, pp. 249–253, 2012.

[17] B. Wu, H. Lin, and M. Lemmon, "Formal methods for stability analysis of networked control systems with ieee 802.15.4 protocol," in *53rd IEEE Conference on Decision and Control*, pp. 5266–5271, Dec 2014.

[18] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, "Performance analysis of probabilistic timed automata using digital clocks," *Formal Methods in System Design*, vol. 29, pp. 33–78, 2006.