An algorithm for learning real-time automata

Sicco Verwer S.E.VERWER@TUDELFT.NL
Mathijs de Weerdt M.M.DEWEERDT@TUDELFT.NL
Cees Witteveen C.WITTEVEEN@TUDELFT.NL
Delft University of Technology, P.O. Box 5031, 2600 GA, Delft, the Netherlands

Abstract

We describe an algorithm for learning simple timed automata, known as real-time automata. The transitions of real-time automata can have a temporal constraint on the time of occurrence of the current symbol relative to the previous symbol. The learning algorithm is similar to the red-blue fringe state-merging algorithm for the problem of learning deterministic finite automata. In addition to *state merges*, our algorithm can perform *state splits* by making use of the time values in the input data. We tested our learning algorithm on randomly generated problems. The results are promising and show that learning a real-time automaton directly from timed data outperforms a method that uses sampling in order to deal with the timed data.

1. Introduction

When no model of a system is known, one can try to construct a model automatically from observations of the system. For example, we would like to model the behavior of truck drivers using a discrete event system (DES) (Cassandras & Lafortune, 1999), but there is not enough expert knowledge available to construct this model directly. We are therefore interested in the automatic generation (learning) of this DES from sensor-data.

A common DES model is a *deterministic finite automaton* (DFA). An advantage of this model is that it is an intuitive framework, i.e., the model can be interpreted by domain experts. When observing a real-world system, however, there often is more information than just the sequence of discrete events: the timing of these events. When time information is important, the DFA model is too limited. Using a DFA, it is impossible to distinguish between events that occur quickly after each other, and events that occur after each other with a significant delay between them. For example, in our project the time between speedups and slowdowns is significant. A sequence of fast changes from slowing down to speeding up and vice versa indicates driving in a city, while a sequence of slow changes indicates driving on a freeway.

A variant of a DFA that includes the notion of time is called a *timed automaton* (TA) (Alur, 1999). In this model, each symbol of a word occurs at a certain point in time. The state transitions of a TA contain constraints on the time values of these occurrences relative to previous occurrences. Thus the execution of a TA depends not only on the type of symbol occurring, but also on the time that has elapsed since some previous symbol occurrence. We are interested in the problem of identifying such a time dependent system from a data sample.

The problem of learning (also known as identification or inference) a DFA from a data set is a well-studied problem in learning theory (see e.g. (Bugalho & Oliveira, 2005; Lang et al., 1998)). There are, however, almost no studies of the inference of TAs from data. Closely related work deals with the problem of learning event recording automata (a restricted but powerful class of TAs) from a timed teacher using query learning (Grinchtein et al., 2006).

Because of the high complexity of learning general TAs, we focus on a another class of TAs known as real-time automata (RTA) (Dima, 2001). An RTA only contains time constraints relative to the previous symbol. In this paper, we study the problem of identifying an RTA from a data sample containing both positive and negative examples. We have constructed an algorithm for this problem. As far as we know, currently no other learning algorithm exists that can

Proceedings of the 18th Benelearn
P. Adriaans, M. van Someren, S. Katrenko (eds.)

identify a timed automaton from a timed sample.

Note that this problem is a lot harder than the problem of learning a DFA from such a sample: in addition to identifying the correct DFA structure, the algorithm needs to identify the correct time constraints. This additional problem causes a large increase in the search space of the algorithm: each transition can be replaced by time-dependent transitions in 2^N ways, where N is the number of possible time values. In previous work we have proven the subproblem of just identifying the time constraints given a correct DFA structure to be NP-complete (Verwer et al., 2006).

Our algorithm is based on ideas similar to those used in state-merging algorithms, which currently are the best performing algorithms for the problem of learning a DFA.[1] In state merging, two states are merged if they display similar behavior. In our approach two states are still merged if they are similar, and in addition a state can be split into two if the two resulting states are dissimilar. These state splits are the result of the addition of a time constraint to the RTA. We define a timed heuristic in order to determine when to perform which split or merge. Our algorithm shows promising results on data from random RTAs.

This paper is structured as follows. We start with a brief introduction to the state merging algorithm for the identification of DFAs in Section 2. We then formally define RTAs in Section 3. In Section 4 we describe our algorithm for the identification of RTAs from data. Then, in Section 5, we compare the performance of our RTA learning algorithm to the straightforward approach of first translating (sampling) the timed input data to untimed data, and then using a standard DFA learning algorithm.

2. State Merging

We assume the reader to be familiar with the basics of the theory of languages and automata. The algorithm we use for the identification of real-time automata is similar to the *red-blue fringe state merging algorithm* for DFAs (Lang et al., 1998). We briefly explain the main elements of this algorithm.

Given a target DFA $\mathcal{A}$, an *input sample* S is a pair of finite sets of positive examples $S_+ \subseteq L(\mathcal{A})$ and negative examples $S_- \subseteq L(\mathcal{A})^C$. The idea of a state merging algorithm is to first construct a tree automaton from this input, and then merge the states of this tree. The tree is called an *augmented prefix tree acceptor* (APTA). An APTA is an automaton representation of the input examples: each input example is represented by a path from the root node to a node in the tree. The node in which a positive or negative example ends is marked positive (accepting) or negative (rejecting), respectively. Figure 1 shows an example of an APTA.

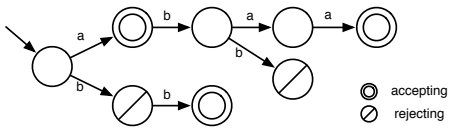

Figure 1. An augmented prefix tree acceptor for the input sample: $(S_+ = \{a, abaa, bb\}, S_- = \{abb, b\})$.

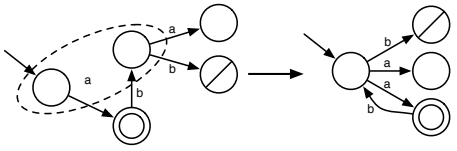

Figure 2. A merge of two states from the APTA of Figure 1. On the left the original part of the automaton is shown, the nodes that are to be merged are surrounded by a dashed ellipse. On the right the result of the merge is shown. This resulting automaton still has to be determinized.

A MERGE (see Figure 2) of two states combines the states into one: all input transitions of both nodes point to this new node and this new node contains the output transitions of both nodes. Such a merge is only allowed if the states are *consistent*, i.e. when no positive node is merged with a negative node. When a non-deterministic choice is introduced, i.e. two output transitions with the same label, the target nodes of these transitions are merged as well. This is called the DETERMINIZATION process, and is continued until there are no non-deterministic choices left. The consistency requirement needs to hold for all states involved in this determinization process. The algorithm continues the state merging process until no more consistent merges are possible.

The red-blue framework follows the algorithm mentioned above, but in addition maintains a core of red nodes with a fringe of blue nodes (see Figure 3). The red nodes are the already identified nodes of the target DFA, and the blue nodes are the current options for merging. A red-blue algorithm starts with the root of the APTA colored red, and its children colored blue. At each iteration the algorithm can either merge a blue node with a red node, or change the color of

[1] See for example the Gowachin language learning competition hall of fame: http://www.irisa.fr/Gowachin/cgi-bin/hallfame.

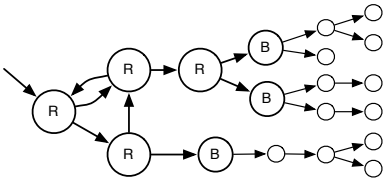

Figure 3. The red-blue framework. The red nodes (labeled *R*) are the identified parts of the automaton. The blue nodes (labeled *B*) are the current candidates for merging. The uncolored nodes (not labeled) are pieces of the APTA.

a blue node into red if no such merge can be found. We call this changing of color a COLOR operation. After this and the subsequent determinization step, all the uncolored children of red nodes are colored blue. Note that a red-blue fringe algorithm never makes changes to red nodes. At each iteration the core of red nodes is assumed to be correctly identified by previous iterations. This is an important property for the heuristic we use in our RTA identification algorithm (see Section 4.1).

The result of a state-merging algorithm can be any DFA that is consistent with the input sample. The main goal of a DFA identification algorithm is to find the smallest such DFA. Currently, the most successful method to find this is evidence driven state merging (EDSM) (Lang et al., 1998). In EDSM each possible merge is given a score based on the amount of evidence in the merges that are performed by the merge and determinization processes. A merge gets an evidence score equal to the amount of positive states merged with positive states plus the amount of negative states merged with negative states. At each iteration of the EDSM algorithm, the merge with the highest evidence score is performed. In this paper we use a similar evidence score.

3. Real-Time Automata

An automaton that accepts (or generates) strings that have a time stamp associated with each event is called a *timed automaton* (Alur, 1999). These strings consisting of event-time stamp pairs are called *timed strings*. Since the symbols in a string represent an ordered sequence of events, we require that the time labels are non-decreasing. We model the time values using natural numbers.[2]

In timed automata, timing conditions are added using a finite number of clocks and a clock guard for each transition. In this section, we describe the class of timed automata that we use in this paper, known as a *real-time automata* (RTAs) (Dima, 2001). An RTA has only one clock that represents the time delay between two consecutive events. The guards for the transitions are then constraints on this time delay. Therefore, we represent a *delay guard* by an interval in $\mathbb{N}$. We say that such a delay guard G is *satisfied* by a time value $t \in \mathbb{N}$ if $t \in G$. An RTA is defined as follows:

[2]This is expressive enough because in practice we always deal with a finite precision of time.

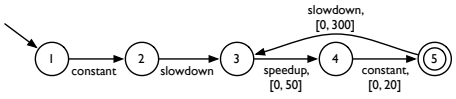

Figure 4. The 'harmonica' driving behavior modeled as an RTA. The numbers used in the delay guards are amounts of tenths of a second.

Definition 3.1 *A* real-time automaton *(RTA) is a tuple* $\mathcal{A} = \langle Q, \Sigma, D, q_0, F \rangle$*, where*

- Q *is a finite set of states,*
- Σ *is a finite set of symbols,*
- D *is a finite set of transitions,*
- q_0 *is the start state, and*
- $F \subseteq Q$ *is a subset of final states.*

A transition $d \in D$ *in this automaton is a tuple* $\langle q, q', s, \phi \rangle$*, where* $q, q' \in Q$ *are the source and target states,* $s \in \Sigma$ *is a symbol, and* ϕ *is a delay guard defined by an interval in* $\mathbb{N}$*.*

In this paper we only regard deterministic (or unambiguous) RTAs. An RTA is called deterministic if no two transitions with the same label and the same source state have overlapping delay guards.

In an RTA it is not only possible to activate a transition to another state, but it is also allowed to remain in the same state for some time (delay). Such a time delay is possible in every state and increases the current delay. A transition to another state is possible only if its delay guard is satisfied by the current delay. A transition $\langle q, q', s, \phi \rangle$ of an RTA is thus interpreted as follows: whenever the automaton is in state q, reading s, and the delay guard ϕ is satisfied by the current delay, then the machine will move to state q'.

The RTA in Figure 4 models a specific driving behavior known as 'harmonica driving'. This often occurs

when a truck is driving at a somewhat higher speed than the vehicle directly in front of it. The driver slows down a bit, waits until there is enough distance between him and the vehicle in front, and then speeds up again, closing in on the vehicle. This whole process often repeats itself a couple of times before the driver finally adjusts the speed of the truck to match the vehicle in front of him. The result of this whole process is unnecessary fuel consumption, which we are trying to reduce. Therefore, we are interested in learning these kinds of patterns, detecting them in real-time, and giving feedback to the truck driver.

Most transitions in this example have a time interval associated with them. These are the delay guards. The definition of a computation as used in DFAs needs to be adapted to deal with these guards. The following definition of a *computation* of an RTA contains the new transition rule discussed above.

Definition 3.2 *A* computation *of an RTA* $\langle Q, \Sigma, D, q_0, F\rangle$ *over a timed string* $(s_1, t_1)\ldots(s_n, t_n)$ *is a finite sequence of states and transitions* $q_0 \xrightarrow{(s_1,t_1)} q_1 \ldots q_{n-1} \xrightarrow{(s_n,t_n)} q_n$, *such that for all* $1 \leq i \leq n$, $\langle q_{i-1}, q_i, s_i, \phi_i\rangle \in D$, *where* ϕ_i *is satisfied by the delay value* $t_i - t_{i-1}$. *A computation of an RTA over a timed string of length n such that* $q_n \in F$ *is called an* accepting *computation.*

The language of an RTA $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of timed strings s such that the computation of $\mathcal{A}$ over s is an accepting computation. In the real world there exist many systems that can be modeled using a time dependent language. We try to identify such a language from examples by using an RTA model and a modified state merging algorithm.

4. Learning Real-Time Automata

Our algorithm for the identification of RTA $\mathcal{A}$ from a timed sample S uses a framework similar to the red-blue framework. In fact, if the algorithm is given an untimed input sample (with all time values 0) then its execution will be identical to an EDSM algorithm using the red-blue framework.

Like a state merging algorithm, our algorithm starts with an augmented prefix tree acceptor (APTA), see Figure 1. Unlike a state merging algorithm, however, each transition of the APTA has a delay guard, see Figure 5. The initial values of the lower and upper bounds all of these guards are set to the minimum and maximum delay values respectively. These values can easily be obtained by taking the minimum and maximum of all delay values occurring in the input sample. Note that this allows for the possibility of inconsistent states in the APTA. We can get rid of these inconsistencies using a *split* operation:

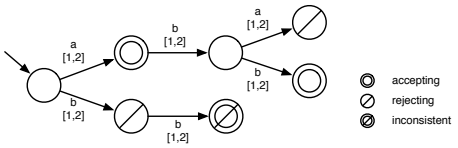

Figure 5. A real-time APTA for the timed input sample: $(S_+ = \{(a,1);(a,1)(b,3)(b,4);(b,2)(b,3)\}$, $S_- = \{(a,1)(b,3)(a,4);\ (b,2);\ (b,1)(b,2)\})$. The minimum delay value of this sample is 1, the maximum delay value is 2.

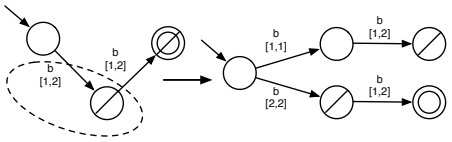

Figure 6. A split of a part of the real-time APTA from Figure 5. On the left the original RTA is shown. The guard and target node that are to be split are surrounded by a dashed ellipse. On the right the result of the split is shown. The split is called using time value $t = 1$.

Definition 4.1 *A* split $s(d, t)$ *of transition d, with clock guard* $g = [t_1, t_2]$, *at time t divides d into two new transitions* d' *and* d'', *with delay guards* $g' = [t_1, t]$ *and* $g'' = [t + 1, t_2]$ *respectively. These new transitions have the same label as d.*

The change a split operation makes in the APTA is determined by the future behavior of individual examples s from S. We call the suffix of s defining this behavior a *tail* of s. In each node of the APTA we maintain a set of these tails. A split divides this set into two new sets: one with all tails with initial delay values less than or equal to t and one with initial delay values greater than t.

After a split $s(d, t)$ we need to change the APTA starting from the node d pointed to, see Figure 6. This node (and all of its children) is replaced by two new nodes. These nodes are the targets of the new transitions resulting from the split. The children of the new nodes are determined by the method used for the original construction of the APTA, but now with the two tail sets as input sample.[3]

[3]This can be implemented efficiently by reusing large parts of the APTA structure starting from the node d pointed to.

The reason for using splits in order to deal with the time information from the input sample is that we want to use as much information as possible to determine the bounds of delay guards. The amount of inconsistencies in the TAPTA that are resolved by a split operation gives us a great deal of this information. The main problem we now have to solve is that we need to find a good measure of the amount of consistencies and inconsistencies in the APTA.

4.1. A Timed Evidence Value

We believe that a good heuristic should be based on the evidence available to the algorithm. We achieve this by calculating an evidence value (score) for the result of each possible operation. The operation that results in a partial solution that agrees most with the available evidence is then chosen to be performed.

In our RTA identification algorithm we could simply use a score almost identical to EDSM. The only difference being that it has to deal with inconsistent merges in addition to consistent merges. This score can be something like the number of added consistent merges minus the number of added conflicting merges. Such a score, however, does not make use of the time information that is available in the APTA in the form of tails.

For instance, suppose we merge two states in the APTA (in the determinization process), each with one tail. We know that both tails start at the same red node. Because of the red-blue framework, we know that we are not going to change their initial execution. Also, due to the fact that they are merged in the APTA, the untimed execution of both tails starting from the red node is identical. Let the tails starting from this red node onwards to be something like: $(a,1)(b,3)(c,5)$ and $(a,2)(b,3)(c,4)$. These tails lie close to each other in time and should get have a higher impact on the score than say: $(a,1)(b,3)(c,5)$ and $(a,5)(b,6)(c,7)$.

The intuition as to why we want these values to be different is easy: tails that lie far away from each other are more likely to be pulled apart by a future split operation than tails that lie close to each other. Based on this intuition, we define the timed distance between two tails s and s' as the probability that s and s' are not pulled apart if we were to choose a split point uniformly at random in each transition. Let t and t' be two delay values. Given the maximum and minimum delay values t_{max} and t_{min}, the probability a uniformly chosen split point divides t and t' is calculated as follows:

$$P(t,t') = \frac{|t-t'|}{t_{max}-t_{min}} \tag{1}$$

Let $t(s,i)$ be a mapping that returns the time delay of s at index i. The probability that two tails s and s' are pulled apart at or before index $i > 1$ if a split point is chosen at every index is calculated as follows:

$$\begin{aligned} P(s,s',i) &= P(s,s',i-1) + \\ &\quad (1-P(s,s',i-1))P(t(s,i),t(s',i)) \end{aligned} \tag{2}$$

This function returns 0 if i is no index of s (and hence $P(t(s,1),t(s',1)$ if $i = 1$). Let B denote the set of tails in one blue node. Our evidence value is determined by the amount of *overlap* within the tails of each blue node. Two tails have a high amount of overlap if their untimed strings (obtained by removing all timestamps) are identical and their probability of being pulled apart (at or before their final index) is low:

$$o(s,s') = \begin{cases} P(s,s',|s|) & \text{if } \mathrm{UT}(s) = \mathrm{UT}(s') \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

Here $\mathrm{UT}(s)$ is a function that returns the untimed version of s. Thus for each pair of tails (s,s') we can obtain a value between 0 and 1, which is close to 1 if s and s' almost certainly overlap, and close to 0 if they almost certainly do not. Let $l(s)$ be a mapping that returns the label (positive or negative) of the example string s is a suffix of. We define the consistency value of a tail s to be the highest overlap value when paired with a tail s' with the same label:

$$c(s) = \max\{1 - o(s,s') \mid s,s' \in B \wedge l(s) = l(s')\} \tag{4}$$

The inconsistency value of a tail is the highest of overlap values when paired with tail with a different label:

$$i(s) = \max\{1 - o(s,s') \mid s,s' \in B \wedge l(s) \neq l(s')\} \tag{5}$$

Thus, instead of the EDSM score which is either 0 or 1 for each tail, we calculate two values between 0 and 1 for each tail. Let $\mathcal{B}$ be the set of all sets of tails in blue nodes. We sum the consistency and inconsistency values over all tails in sets of tails in $\mathcal{B}$ and add them to the amount of consistent merges in red nodes, denoted by RED_MERGES to obtain our timed measure:

$$\text{score} = \text{RED\_MERGES} + \sum_{B \in \mathcal{B}} \sum_{s \in B} c(s) - i(s) \quad (6)$$

This measure is calculated for all merge, split and color operations and the highest scoring operation is performed. In the case of ties preference is given to the operation that minimizes the total size of the RTA, i.e. we use the preference order: merge, split, color. In the often occurring case where two splits have identical scores, we choose the one that maximizes the size of the smallest delay guard resulting from the split. The intuition here is that we want to maximize the amount of information in both delay guards. Since, if we then have to perform a second split to remove an inconsistency, this second split will be based on a fair amount of evidence.

4.2. The Algorithm

Our algorithm is a timed version of the EDSM algorithm using the red-blue framework. In addition to merging a blue node b, or coloring it red, it is capable of splitting any transition d to a blue node b at any time point t. Since we can obtain a minimal and maximal time value there is a finite amount of possible splits.

Many of these splits have an identical and/or similar result on the APTA. We can calculate these values efficiently as follows. We obtain the time values of all the tails in b, and store them in increasing order, resulting in a set $(t_1, t_2, \ldots, t_n)$. We calculate the score of every split (d, t) such that $t = (t_i + t_{i+1})/2$ for some $0 < i < n$. This can be computed efficiently because each next split we try only changes the path of one single tail (or a few if their first delay value is identical and thus cannot be split).

The merge and determinize operations of the DFA identification algorithm are modified slightly to deal with delay guards. Because of the clock guards, it is possible that two nodes that should be merged have transitions with different guards. Because we use the red-blue framework, this will only be the case when we merge a node b with a red node r. Also, b will always still have its initial delay guards. Thus we simply split b at exactly the same values as r before merging the two.

Due to our timed heuristic the operation of coloring a blue node red can also get a positive score. This is due to the way we count consistent merges in red nodes, and because we evaluate a new set of tails. That is why we calculate the score for every possible operation. We only disallow merges and colorings that create inconsistencies in a red node. Algorithm 1 shows the pseudo code of the main routine of our algorithm.

Algorithm 1 State merging and splitting for RTAs

Require: A timed input sample.
Ensure: The result is a small RTA that is consistent with the input sample.
  Construct the timed APTA from the input sample.
  Color the root node red and all of its children blue.
  while Some nodes are colored blue **do**
    Evaluate all possible consistent merges, splits, and colorings of blue nodes.
    if A MERGE(r, b) scores highest **then**
      Apply all splits of transitions in r to b.
      Perform MERGE(r, b) and call DETERMINIZE().
    end if
    if A SPLIT(d, t) scores highest **then**
      Perform SPLIT(d, t) (including the creation of new nodes and children)
    end if
    if A COLOR(b) scores highest **then**
      Perform COLOR(b).
    end if
    Color all uncolored children of red nodes blue.
  end while
  return The constructed RTA

5. Testing

In order to test our algorithm we compared the results with a straightforward approach for the same problem. This involves first sampling the data using some fixed frequency, and then using a DFA learning algorithm to learn the language of the sampled data. We tested the solutions found by both approaches on the size of the solution and the error made when asked to label new data. In the two sections below we will discuss our results.

5.1. Data

We created random data in order to test our algorithm. We first created a random RTA, with a fixed number of states, and a fixed number of split intervals. The split intervals were generated by applying the SPLIT routine to a randomly picked transition, using a time value chosen uniformly between the upper and lower bound of the guard of the transition. The minimum and maximum time values are 0 and 10000. Each state of the RTA has a chance of 0.5 to be a final state. We disallowed the case that all or none states were chosen to be final.

Next, we randomly generated different amounts of

timed strings: 50, 500, 1000, 2000, 5000, 10000, and 100000. For each symbol of a timed string we uniformly picked a value from the interval $(0, 10000)$, which we used as its time value. Each timed string has a chance of $\frac{1}{10}$ to stop in each state it visits (uniformly pick a random value v between 0 and 1, generate the next symbol only if v is greater then $\frac{1}{10}$). The label of each string was determined by the state it ended in.

We generated these data sets for RTAs with 2, 4, 8, 16, and 32 states. All RTAs had 3 different amounts of splits: half, equal, and two times the amount of states of the RTA. For each state-split combination we created 5 random RTAs. We then collected random samples from these RTAs and ran our algorithm on the samples of all sizes up to 100000. The 100000 sample was used to evaluate the performance of the algorithm. We use the percentage of correctly labeled new examples as an indicator for the performance.

The RTA learning algorithm is an alternative to the straightforward approach of first mapping the timed input sample to an untimed input sample, and then to learn a DFA from the untimed data. We sampled the data using fixed sampling lengths: 100 and 1000. Thus, for each symbol s, replace s with n untimed copies of s, where n equals the delay of s divided by the sampling length. We used normal rounding to get rid of fractions.

We ran a standard state merging algorithm on the sampled datasets. The algorithm we tested is the red-blue algorithm, which we downloaded from the Abbadingo web-site.[4] Figure 7 shows the result of the red-blue learning algorithm when applied to the sampled data compared to the results of the RTA learning algorithm applied to the unsampled original datasets.

5.2. Results

In the graphs in Figure 7 we can observe a couple of things. First of all our timed state merging/splitting algorithm seems to perform really well compared to the sampling method. Our algorithm is capable of achieving 80% of correctly classified new examples for RTAs up to 16 states and 8 splits (number 9 in the graphs). This shows that it is possible to apply an algorithm such as ours to real-world problems.

Furthermore, the bad performance of the sampling methods shows us the difficulty of the problems. For larger sizes the sampling methods perform only slightly better than tossing a fair coin in order to classify new examples.

An interesting phenomenon is that for the performance of the algorithm when given 10000 examples is worse than when given 2000 examples. We are not really sure why this happens. A possible explanation could be that with more input, there are more possible splits and more states in the APTA. Because of this, there are more ways in which the algorithm can make a mistake. We think this might be the cause of the degrading performance. This is supported by the fact that the sizes of the found solutions are also larger when there are more examples in the input sample.

Another interesting result is that the sampling length does not really seem to matter: the results for the two lengths are almost identical. One would have expected the more accurate method (length 100) to have a better performance. But in most cases it even seems to perform slightly worse.

The amount of time it took to calculate all these results is nearly two weeks. But the results of our algorithm only took a weekend to compute. Thus our algorithm does not only obtain a better performance, but it is also more efficient than the sampling method.

6. Conclusions and Future Work

We have described an identification algorithm for real-time automata. These automata can be used to model systems for which the time between consecutive events is important for the system behavior. To the best of our knowledge, ours is the first algorithm that can identify a real-time (or any timed) automaton from a timed input sample. Our results show that RTA idenitification should be possible in real-world applications. Also, learning time constraints directly seems to outperform methods that first sample the data, and then use the sampled data in an untimed learning algorithm.

Currently, we are constructing a more sophisticated search method for the RTA identification problem. Our algorithm right now uses a greedy approach for the identification problem. We believe a search method, such as breadth-first search or a local search, to be more appropriate. For example, when our algorithm makes a decision to split a delay guard at a specific time value, it is often slightly off. We want to be able to change or slightly adapt this decision when the future search process reveals it to be incorrect. We hope that the use of a search method solves the problem of the degrading performance when there are more examples in the input sample (see Section 5.2).

[4] http://abbadingo.cs.nuim.ie/

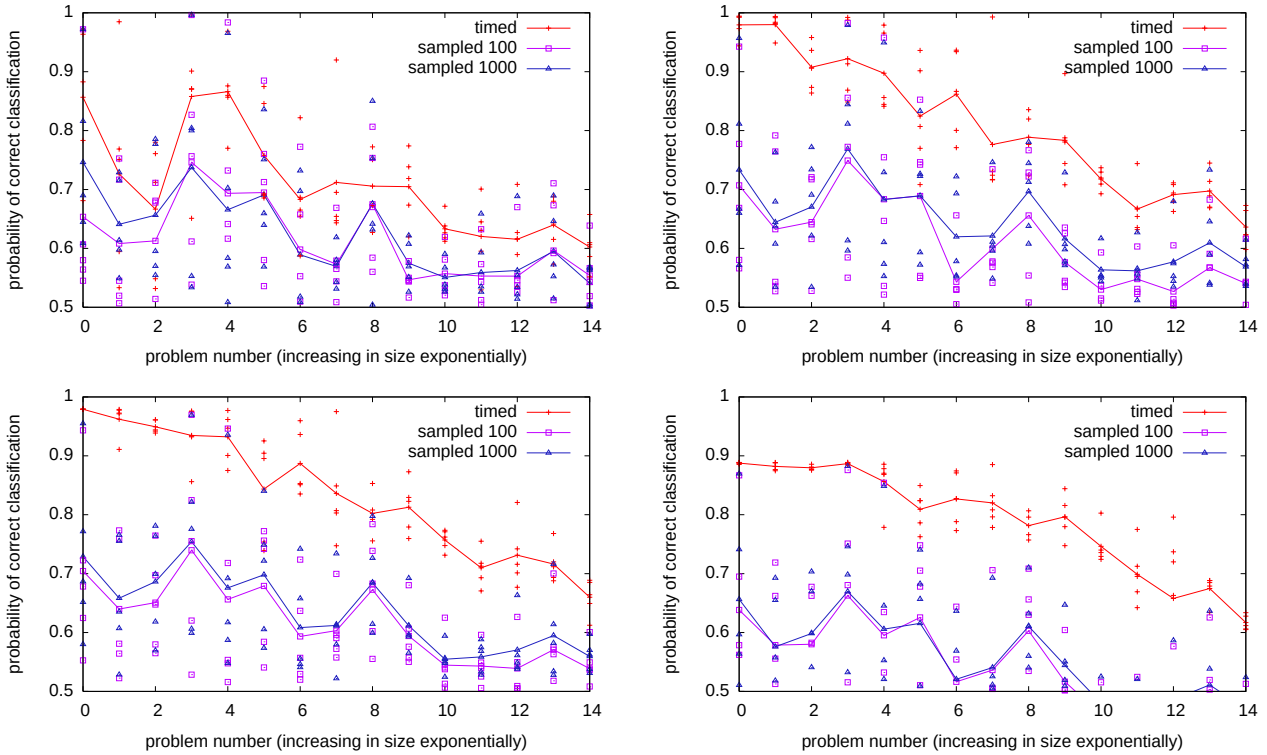

Figure 7. Results from our experiments. Each of the graphs shows the results obtained with different input sample sizes: 50 examples top-left, 500 examples top-right, 2000 examples bottom-left, and 10000 examples bottom-right. We do not show the results of the sets with 1000 and 5000 examples because they are very similar to the results with 2000. The probability that a new example is classified correctly (from a test set of 100000 examples) is shown for each of the tested problem instances (5 of each size). The problem instances range from 2 states, with 1 split, to 32 states with 64 splits. All of the found results are shown as points. The shown lines are the averages of these points.

In future work, we would like to generalize this algorithm to probabilistic timed automata. Probabilistic automata are equivalent to commonly used hidden Markov models (in the sense that they generate the same distributions) (Dupont et al., 2005). Since a probabilistic DFA defines a distribution over strings, it is possible to learn a probabilistic DFA solely from positive examples. This makes it easier to apply in a real-world setting.

References

Alur, R. (1999). Timed automata. *International Conference on Computer-Aided Verification* (pp. 8–22). Springer-Verlag.

Bugalho, M., & Oliveira, A. L. (2005). Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, *38*, 1457–1467.

Cassandras, C. G., & Lafortune, S. (1999). *Introduction to discrete event systems*, vol. 11 of *The Kluwer International Series on Discrete Event Dynamic Systems*. Springer Verlag.

Dima, C. (2001). Real-time automata. *Journal of Automata, Languages and Combinatorics*, *6*, 2–23.

Dupont, P., Denis, F., & Esposito, Y. (2005). Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*.

Grinchtein, O., Jonsson, B., & Petterson, P. (2006). Inference of event-recording automata using timed decision trees. *CONCUR* (pp. 435–449). Springer.

Lang, K. J., Pearlmutter, B. A., & Price, R. A. (1998). Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. *Proceedings of the ICGI*. Springer.

Verwer, S., de Weerdt, M., & Witteveen, C. (2006). Identifying an automaton model for timed data. *Proceedings of the BENELEARN* (pp. 57–64).