Identifying an automaton model for timed data

Sicco Verwer[1] 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

A model for discrete event systems (DES) can be learned from observations. We propose a simple type of timed automaton to model DES where the timing of the events is important. Learning such an automaton is proven to be NP-complete by a reduction from the problem of learning deterministic finite state automata (DFA) without time. Based on this reduction, we show how the currently best learning algorithm for DFAs (state merging) can be adapted to deal with time information.

1. Introduction

Many systems we encounter in nature or in our society, we do not fully understand. Nonetheless, we like to have a model of such systems. If there is no explicit model available, we might try to construct it from observations of the behavior of the system. Such a model can then, for example, be used to give more insight in how the system works (maybe even leading to improvements of the system itself). Moreover, a model can be used to simulate such a system, to predict future behavior, or to monitor the system on-line to be able to act when the system gets into an undesired state.

In this paper we focus on systems of which the execution is determined by a finite set of discrete events. Such a system is known as a discrete event system (DES) (Cassandras & Lafortune, 1999). Sometimes no model is known for these systems. Therefore, we are interested in learning a model for these kinds of system from observations.

[1]This research has been supported and funded by the Dutch Ministry of Economical Affairs under the SENTER program. Project IS041022 Real-Time Optimalisatie Motor Management.

A common way to model discrete event systems is by using deterministic finite state automata (DFA). The problem we study in this paper can be stated very succinctly as follows: *Given a data sample of observations, how to identify the correct DFA for a specific DES.*

When observing a real-world system, however, there often is information in addition to the system events, namely, their times of occurrence. If this time information is important, a DFA is too limited. For example, 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.

Consequently, we would like a model that can also deal with time delays. We call this type of automaton a delay automaton (DA). Our goal is to find an algorithm that can use sampled data (i.e., events including time information) to identify a DA describing the underlying system. As far as we know, currently no other learning algorithm exists to identify a timed automaton from a timed sample.

The paper is structured as follows: first we define a DFA and this model of a DA, and then we briefly discuss an established method for identifying an automaton, called state-merging. Next, in Section 3, we study the complexity of the learning problem of identifying a DA. We prove that learning the time constraints (guards) in our model on itself already is an NP-complete learning problem. This proof relies on a reduction from the (ordinary) DFA learning problem. We then observe that this proof in fact neatly describes a relation between learning a DA and learning an ordinary DFA. From this observation we propose an adaptation of the state merging algorithm for DFAs to deal with DAs. This algorithm is described in Section 4. For this we first discuss some background on identifying DFAs. We end with stating some conclusions and future work.

2. Background on Automata and Identification

Many different systems we encounter in practice can be modeled using a finite set of discrete states that are associated with a set of discrete events. These systems are known as *discrete event system*s (DES) (Cassandras & Lafortune, 1999). A DES has the following properties:

- It is in one state at each moment in time.
- An event can occur instantaneously, which causes a transition from one state to another state (possibly the same).
- It is completely *event-driven*, which means that its state evolution depends entirely on the occurrence of discrete events.

The study of DESs is mainly concerned with the sequence (ordering) of events that can occur in a given system. These sequences are called *strings*. The set of all possible strings is known as the *language* of a DES. There are several ways to represent a DES language. The most common DES model is the *deterministic finite automaton* (DFA).

2.1. Deterministic finite state automata

A DFA is a directed graph consisting of a set of states (nodes) and transitions (directed arcs), see for example the introduction by Sipser (1997). Transitions point from source to target states, and are labeled with *events* (denoted by symbols). When an event occurs, it activates the transition labeled with that event. This changes the current state of the DFA to the target state of the transition (the next state). There is an important distinction between a DFA and a *non-deterministic finite automaton* (NFA). In the non-deterministic case, two (or more) transitions can be activated by the same label.

An example of a DFA is shown in Figure 1. This DFA models an automatic bike light. The states of the DFA are given names for convenience. Execution of the DFA starts in the state off. This is indicated by an arc pointing to this state from nowhere. The execution can end when the DFA is back in its starting position, indicated by the double circle. The bike light is turned on when it is both dark and someone is riding the bike. Formally, a DFA is defined as follows:

Definition 2.1 *A* deterministic finite state automaton *$\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where*

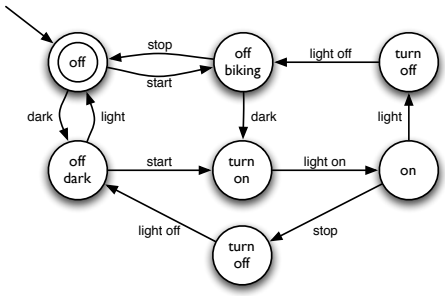

Figure 1. This DFA models the execution of an automatic bike light. The events consist of: start and stop (cycling), (it getting) dark and light (outside), and (turning the) light on and light off.

- *Q is a finite set of states,*
- *Σ is a finite set of symbols to label the transitions, known as the alphabet,*
- *δ is a partial mapping from $\Sigma \times Q$ into Q that represents all transitions,*
- *$q_0 \in Q$ is the start state, and*
- *$F \subseteq Q$ is the set of final states.*

The mapping δ defines the transitions of the automaton, and is therefore known as the *transition function*. In an NFA the transition function is a function from a source state and a symbol to a *set* of possible target states: from $\Sigma \times Q$ into 2^Q. The automaton uses the transition function to accept strings. How this is done by the automaton is defined in the definition of a *computation* of a DFA.

Definition 2.2 *A (finite)* computation *of a DFA $(Q, \Sigma, \delta, q_0, F)$ over a string $s_0 s_1 \ldots s_{f-1}$ is a sequence $\mathcal{C} = q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} q_2 \ldots \xrightarrow{s_{f-1}} q_f$ of states and transitions, such that for all $0 \leq i \leq f-1$, it holds that $\delta(s_i, q_i) = q_{i+1}$, $q_i \in Q$, and $s_i \in \Sigma$. A finite computation is called* accepting *when $q_f \in F$.*

The language $L(\mathcal{A})$ of a DFA $\mathcal{A}$ is the set of strings s for which the computation of $\mathcal{A}$ over s is accepting. A language is a subset of the set of all strings, denoted by Σ^*.

We are interested in learning a DES, modeled by a DFA, from the observations of a real-world system. These observations are elements of the language of the system (positive examples). The DFA learning framework also requires a set of strings that is disjunct from the language of the DFA (negative examples). These examples can, for example, be obtained

by monitoring other systems. Learning a DFA from both positive and negative example strings is known as *DFA identification*.

The task of DFA identification is to find the smallest DFA $\mathcal{A}$ (in terms of the number of states) such that $L(\mathcal{A})$ is consistent with the given input data. This data, known as an *input sample*, consists of two sets of strings: *positive strings*, which are elements of $L(\mathcal{A})$, and *negative strings*, which are elements of $\Sigma^* - L(\mathcal{A})$.

A language is *consistent* with an input sample S when it contains all positive examples, and none of the negative examples. From an input sample a consistent DFA can be identified using an algorithm called state merging.

2.2. State merging

In DFA learning practice a technique known as *state merging* is regarded as being the most succesful (Bugalho & Oliveira, 2005). There are two kinds of state merging algorithms: exact and polynomial. The original state merging (Oncina & Garcia, 1992) algorithm is a polynomial time algorithm that correctly infers a DFA when the input set contains a so called *characteristic set* (de la Higuera, 1997). The set is a subset of the input sample such that the polynomial state merging algorithm has to return the correct solution. When no such characteristic set is available an exact algorithm can be used to find the correct DFA. Exact algorithms use techniques such as backtracking in order to find the minimal DFA. In this paper we focus on the exact algorithm, but the algorithm in Section 4 can also be applied as a polynomial time algorithm. For example as a form of *evidence driven state merging* (Lang et al., 1998).

State merging starts with the construction of a *prefix tree acceptor* (PTA) from the input data. Such a PTA is the automaton in the form of a tree where each string is represented by a path from start state to a leaf. Such a leaf (state) is labeled final in the PTA when the input string reaching that state is positive. In order to detect inconsistencies, the states in which negative examples end are labeled negative (or unfinal). The labels of the remaining states are (as of yet) undefined. When strings have the same prefix, they share the first part of their path in the tree, hence the name prefix tree acceptor. Figure 2 shows an example of an input sample and the corresponding PTA.

The process of combining identical prefixes of the input sample in a PTA reduces the complexity of the input. All that state merging algorithms try to do is to combine identical *suffixes*. This is done by merging

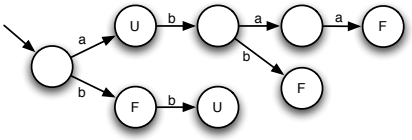

Figure 2. This prefix tree acceptor is constructed from the input sample $S_+ = \{\mathsf{b}, \mathsf{abb}, \mathsf{abaa}\}, S_- = \{\mathsf{a}, \mathsf{bb}\}$. Final states are labeled with an F, unfinal states with a U.

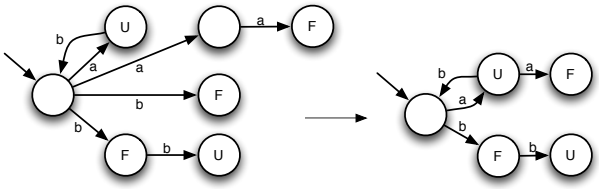

Figure 3. The left automaton is the result of a *merge* of the initial state and the state reachable by ab of the PTA from Figure 2. The right automaton shows the result of the *determinization* of the left automaton.

states from the PTA.

A *merge* of two states q_1 and q_2 is an operation that combines the states q_1 and q_2 into one new state q'. All transitions of the DFA which have either q_1 or q_2 as their target state, then get q' as their target state. The same holds for the source states. Note that the result of this operation can be an NFA, because the original states may have an outgoing transition with the same label. Therefore, when learning a DFA, a second function is used to make the automaton deterministic. This is the determinization function.

Given an NFA, the *determinization function* continuously merges the target states of a nondeterministic choice in the NFA. In other words, if the transition function δ is such that $\delta(q, s) = \{q_1, q_2, \ldots, q_n\}$, for some $q \in Q$, $n \geq 2$ and $s \in \Sigma$, then a merge of the states q_1 and q_2 is performed. This determinization is repeated until the result is a DFA. Figure 3 shows the result of a merge and determinization applied to the PTA of Figure 2. In this case, the resulting DFA is consistent, i.e., it still accepts all positive input strings and rejects all negative input strings. What happens when the resulting DFA is inconsistent is defined by the main procedure of the *state merging* algorithm.

Using i) the PTA, ii) the merge operation, and iii) the determinization function, a *state merging* algorithm can be constructed that learns a minimal DFA from an input sample. The algorithm starts by constructing a PTA from the input sample, and then it merges

states of the PTA (using the determinization function whenever necessary) until no smaller consistent DFA can be found. When no two states can be merged such that the resulting automaton is consistent, the process backtracks on its last merge(s).

Unfortunately DFAs do not make use of the time information that is sometimes associated with the events in the strings of the input sample. For this, variants of a DFA exist, called timed automata.

2.3. Delay Automata

An automaton that accepts (or generates) strings with a timestamp associated with each event is called a *timed automaton* (Alur, 1999). These strings consisting of event-timestamp 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.

In timed automata, timing conditions are added using a finite number of clocks and a clock guard for each transition. In this section, we introduce a simple type of timed automaton, which we use in this paper. This type of timed automaton has only one clock that represents the time delay between two events. The clock guards for the transitions are then constraints on this time delay. Therefore, we represent a *delay guard* (constraint) by an interval in $\mathbb{R}^+$. We say that such a delay guard G is *satisfied* by a time delay $d \in \mathbb{R}^+$ if $d \in G$. We call the simple type of timed automaton using this delay guards a *delay automaton* (DA). A DA is defined as follows:

Definition 2.3 *A* delay automaton *(DA) is a tuple* $\mathcal{A} = \langle Q, \Sigma, T, q_0, F \rangle$*, where*

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

A transition $t \in T$ *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* $\mathbb{R}^+$.

Furthermore, we would like a delay automaton to be deterministic, just as a DFA. Therefore, no two transitions with the same label and the same source state should have overlapping delay guards.

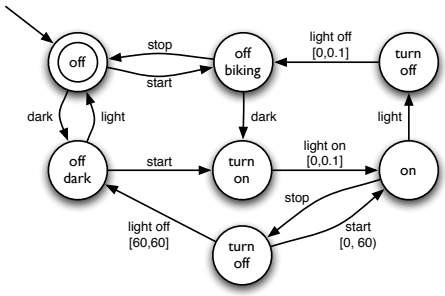

Figure 4. This DA models a 'smart' bike light that leaves the light on when the bike stops for less than 60 seconds.

In a DA 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 a DA 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 DA in Figure 4 models a 'smart' bike light that does not turn the light off when the bike stops for a traffic light. Such a stop should not last longer than one minute. This is modeled by a light off event only occurring at time 60. A start event before 60 seconds will make the automaton return to the on state.

Some transitions in this example have a time interval associated with them. These are the delay guards. The definition of a computation (Definition 2.2) needs to be adapted to deal with these guards. Moreover, also the new transition rule discussed above is included in the following definition of a *computation* of a DA.

Definition 2.4 *A* computation *of a DA* $\langle Q, \Sigma, T, q_0, F \rangle$ *over a timed string* $(s_1, t_1) \dots (s_n, t_n)$ *is a finite sequence of states and transitions* $q_0 \xrightarrow{(s_1,d_1)} q_1 \dots q_{n-1} \xrightarrow{(s_n,d_n)} q_n$*, such that for all* $1 \leq i \leq n$*,* $\langle q_{i-1}, q_i, s_i, \phi_i \rangle \in T$*, where* ϕ_i *is satisfied by delay* $d_i = t_i - t_{i-1}$*, i.e.* $d_i \in \phi_i$*. A computation of a DA over a timed string of length n such that* $q_n \in F$ *is called an* accepting *computation.*

The example shows that the timed automaton introduced in this section can model systems that have observations with their observed times (timed strings) as input, and have different results depending on this time information. Our goal is to identify such a delay automaton given such a set of timed strings. In

the next section we study this problem of learning a delay automaton in more detail.

3. The Complexity of Learning Delay Guards

The main reason we had to formalize a DA is to be able to learn timed relations of reactive systems. A DA is one of the simplest ways to include timed relations into the framework of automata. Thus an important question to ask is how we would be able to learn a DA from data. The data our learning algorithm would get as input consists of both positive and negative examples of timed strings. This is the same as the input sample for learning DFA, but now each symbol is paired with its time of occurrence.

For designing a learning algorithm, it is interesting to know the complexity of learning a DA, and of the subproblem of inferring the optimal delay guards. For example, if the delay guard learning problem is in P, we might be able to use a delay guard learning algorithm as a subroutine of an existing DFA learning algorithm. In this section, however, we prove that the delay guard learning problem is NP-complete by a reduction from the problem of learning a DFA. After that, we discuss the complexity of the complete problem of learning a DA.

Let us start by giving the definitions of the (ordinary) DFA learning problem and the delay guard learning problem.

Definition 3.1 *Given an alphabet Σ, an integer k, and an input sample S, the* DFA learning problem *is the problem of finding a DFA of k states, with alphabet Σ, that is consistent with S.*

Definition 3.2 *Given an NFA $\mathcal{A}$, and a timed input sample S, the* delay guard learning problem *is the problem of adding delay guards to the transitions of $\mathcal{A}$, such that the resulting DA is consistent with S.*

The DFA learning problem has been proven to be NP-complete (Gold, 1978). We now show that:

Theorem 1 *The delay guard learning problem is NP-complete.*

In the following proof of this theorem we show that any instance of the NP-complete problem of DFA learning can be translated to an instance of the delay guard learning problem in polynomial time. From this and the fact that the delay guard learning problem can be verified in polynomial time, it follows that the delay guard learning problem is also NP-complete. In this reduction the main idea is that the delay guards can be used to identify the transitions to use in the DFA learning problem. Note that this choice is the only choice which needs to be made in the problem of learning a DFA. The choice of which states are final is then easily solved by computing the ending state of each string in the input sample.

Proof Suppose we are given a DFA learning problem instance consisting of an alphabet Σ, an integer k, and an input sample S. From this we have to find a DFA with k states that is consistent with S. We transform this instance to an instance of the delay guard problem as follows:

First we construct a (universal) NFA $\mathcal{A}$ with a set Q of k states, and for each state q and for each symbol $a \in \Sigma$ a transition (q, a, q') to each state $q' \in Q$.

Next, we transform each example string $s \in S$ into a timed string in the following way:

$$s_1 s_2 \dots s_n \rightarrow (s_1, 1.0)(s_2, 2.0) \dots (s_n, n)$$

Thus there always is a delay of exactly 1.0 between successive symbols in an example string.

Now suppose that we have found a set of delay guards for the NFA resulting in a deterministic delay automaton that is consistent with S. Without loss of generality, we may assume that in processing sample strings from S every state $q \in Q$ is used. Since the resulting DA is deterministic, for every state q and input symbol a, exactly one of the guards will be satisfied and this guard (interval) is uniquely determined by having the property that 1.0 is contained in the interval. This is illustrated in Figure 5.

In order to find the DFA consistent with S we can select the right transitions from the resulting DA as follows. For each state q and each input symbol a, we select (from the set of all transitions) the unique transition (q, a, q', ϕ) for which $1.0 \in \phi$. From this we can easily find the DFA consistent with S solving the DFA-learning problem: for every such a transition (q, a, q', ϕ) selected, let (q, a, q') be a transition in the DFA.

Because DFA learning is NP-complete, we can now conclude that delay guard learning is NP-hard. Furthermore, a solution to the delay guard learning problem can be verified by checking that each positive timed input string ends up in a final state, and that each negative input string does not. This can be done in polynomial time (i.e. the sum of the lengths of all input strings). This makes the problem of learning

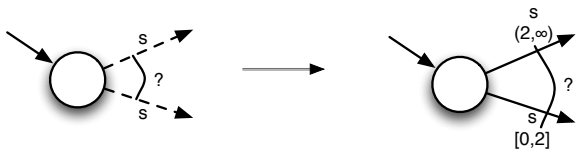

Figure 5. In the reduction the choice of which transition to add is mapped to the choice of which delay guard to give to which transition. Since the delay time is always 1.0, the timed strings that at some point reach the state in the figure with s as their next event, will all activate the transition pointing downwards.

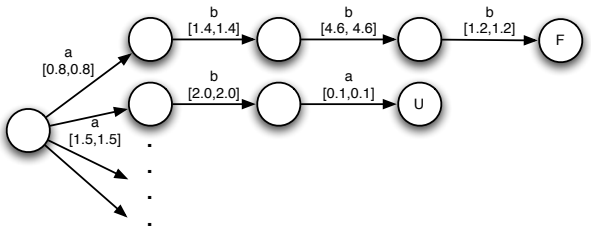

Figure 6. Because timed strings rarely have exactly the same prefixes, a timed PTA usually looks more like a list of strings.

delay guards NP-complete. □

The correspondence between delay guards and transitions can also be used to prove that the problem of learning a consistent DA of k states from a timed input sample is NP-hard. Also for this problem, membership of NP holds, because verifying that a DA is consistent with an input sample set can be done in polynomial time. This proves the following corollary.

Corollary 2 *The problem of learning a consistent DA of k states from a timed input sample is NP-complete.*

This result is not restricted to DAs. It applies also to, for example, event clock automata (Alur et al., 1999). In event clock automata there are two clock values for every event: one for the future and one for the past. In fact, the reduction can be reconstructed for any timed automata in which we can force the clocks to have the same value at each input symbol, i.e. implementing a delay of 1.0.

4. A DA Learning Algorithm

The reduction we presented in the previous section describes the correspondence between the delay guards of a DA and the transitions of a DFA. Our idea is to use this correspondence to design a DA learning algorithm based on the DFA learning algorithm, state merging, described in Section 2.2. In this section we elaborate on this idea and, eventually, present a state merging algorithm for learning delay automata.

In the state merging algorithm the transitions are derived by merging the states of a prefix tree acceptor (PTA)(see Section 2.2). From the reduction we can conclude that learning a delay guard is similar to deriving a transition. Therefore, it should be possible to use the same state merging technique for learning the delay guards. This requires a timed version of a PTA, which is constructed in the same way as the standard PTA, but with the following modification.

In a timed PTA, each transition is given as delay guard the degenerate interval (of length zero) of exactly the delay value of the timed sample string used to create the transition. For example, suppose that the pair (s_i, t_i) of a timed example string is used to create a transition t. In this case, the value of the delay guard of t is set to be the 'interval' $[t_i - t_{i-1}, t_i - t_{i-1}]$ (where t_{i-1} is the time value of the previous symbol-time value pair).

The fact that we set these intervals to include only one real number implies that there is almost no overlap in the prefixes of the timed strings. Consequently, the timed PTA in fact is a list of timed strings instead of a tree. This, however, is no big issue for the algorithm, apart from that the starting size of the PTA is a lot bigger than the PTA of the DFA learning algorithm. Figure 6 shows an example of such a timed PTA.

Starting with a timed PTA, states are merged in the same way as described in Section 2.2. The only difference from a DFA state merging algorithm is the determinization step. In this step the delay guards of several transitions with the same symbol can be merged into one, merging the transitions and the target states. The way in which this is done is determined by the following rules:

- Whenever there are two transitions with the same symbol as label and with the same target state, these transitions are merged into one. Given the delay guards of the original transitions $[l_1, h_1]$ and $[l_2, h_2]$, the delay guard of the merged transition becomes $[min(l_1, l_2), max(h_1, h_2)]$.
- Whenever there are two transitions with the same symbol as label and with overlapping delay guards, these transitions and their target states are merged in the standard way, and the guards

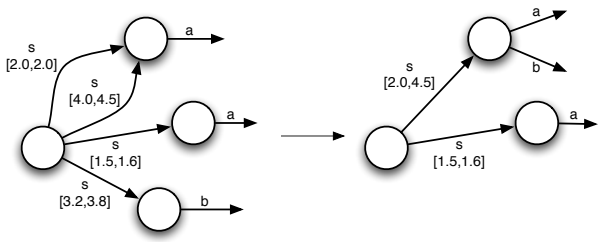

Figure 7. The determinization function merges i) transitions with the same label and the same target state, and ii) transitions with the same label and an overlapping interval.

are combined into one as above.

These rules are depicted in Figure 7. Note that by using these rules we actually learn a restricted type of DA. This type consists of DAs where two transitions t_1 and t_2 cannot have the same target state s if they originate from the same source state, and have the same symbol as label. These transitions, however, can be modeled by using an additional state s'. This state s' is a copy of s (including incoming and outgoing transitions). By letting t_1 and t_2 have s and s' as their target states respectively, the behavior of the DA is identical to the DA which was disallowed. Since we require additional states to model more complex delay guards, a consequence of this restriction is that minimizing the amount of states of a DA also minimizes the complexity of its delay guards. Since in learning it is customary to minimize the amount of states, this restricted DA is a natural type of DA for learning purposes.

With the two simple determinization rules our learning algorithm for DAs is nearly complete. Our algorithm is capable of merging states, starting from a timed PTA, such that the result is a deterministic DA. This resulting DA, however, is only defined (in every state, and for each symbol) for a subset of $\mathbb{R}^+$. To use this model for new inputs, however, we would like to generalize this to the whole of $\mathbb{R}^+$. A simple routine, which we call *finalize*, realizes this generalization by extending the intervals of all delay guards until they meet. This can be done in several ways, for example:

Let $D_{q,s}$ be the set of delay guards of the transitions from state q with symbol s as its label. The finalize routine applies the following rules to each set $D_{q,s}$:

- The delay guard $[l, h]$ with the highest upper bound h of all delay guards in $D_{q,s}$ is given the value $[l, \infty)$.
- The delay guard $[l, h]$ with the lowest lower bound l of all delay guards in $D_{q,s}$ is given the value $[0, h]$.
- For each pair of delay guards $[l_1, h_1]$ and $[l_2, h_2]$, with $h_1 < l_2$, such that there is no delay guard in $D_{q,s}$ in the interval $[h_1, l_2]$, are given the values $[l_1, (h_1 + l_2)/2]$ and $((h_1 + l_2)/2, h_2]$.

When the finalize routine is finished, the result is a complete DA consistent with the timed input sample. Note that the third rule (which just takes the average between the upper and lower bound of the two intervals) is a very simple way to generalize. While it does not matter for the consistency of the DA, an implementation of this algorithm could use a more sophisticated method, using for example the size and density of both guards. Algorithm 1 shows the main routine of the algorithm in pseudo code.

Algorithm 1 State merging delay automata

Require: A timed input sample S. A required size of the target DA k.
Ensure: $\mathcal{A}$ is a DA of size k, that is consistent with the input sample S, when no such DA exists false is returned.

```
Construct PTA A from S.
Call the find_da function.
Function find_da():
for All pairs of states {q1, q2} in A do
  A = determinize(merge(q1, q2))
  if A constains no inconsistent state then
    if size(A) equals k then
      return A
    end if
    if find_dfa() ≠ false then
      return find_dfa()
    end if
  end if
  undo determinize(A).
  undo merge(q1, q2).
end for
return false
```

5. Discussion and Current Work

Our research goal is to find an algorithm to learn a model for systems that can be described by discrete events and the time at which they occur. More specifically, we would like to be able to identify a finite state automaton with time information from the observations of the events and their starting time. To this end, we defined a simple type of timed automaton, which we called a delay automaton (DA).

In this paper we showed that learning the guards alone is already NP-complete. We also showed that learning both the automaton and the time labels is NP-complete. This is a bit unexpected, since one would think that adding time information to a learning problem makes it more expressive, and hence more difficult. We may, however, conclude that learning (certain types of) timed automata is not significantly harder than learning DFAs. From this we derived an alternative to the straightforward approach of first mapping the timed input sample to an untimed input sample, and then to learn the DFA from the untimed data. In future work we would like to find out whether the solutions found by this approach and our alternative algorithm are identical.

We derived the idea for our learning algorithm from the reduction proof: we discovered how to adapt the state merging algorithm for DFAs to our DAs. To the best of our knowledge, this is the first proposal for an algorithm that can identify a timed finite state automaton from a timed input sample. Closely related work deals with the problem of learning an event recording automaton (an event clock automaton without clocks for future events) from a timed teacher for membership and equivalence queries (Grinchtein et al., 2004). A problem of this approach is that it requires a polynomial amount of queries in the size of the zone graph, and the size of this graph can be doubly exponential in the size of the minimal automaton.

Current work is to implement and test this DA learning algorithm on observations with time information. Furthermore, we would like to generalize both this algorithm and the complexity results to probabilistic timed automata. Probabilistic automata are equivalent to commonly used hidden Markov models (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. An adaptation of the state merging algorithm to the problem of learning probabilistic DFAs from just positive examples is given in (Carrasco & Oncina, 1994). In the probabilistic setting there is also some work on learning timed systems. In (Sen et al., 2004), for example, the state merging algorithm is adapted in order to learn continuous-time Markov chains in the limit with probability one. We would like to adapt our state merging algorithm in a similar way to the problem of learning a probabilistic delay automaton.

References

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

Alur, R., Fix, L., & Henzinger, T. A. (1999). Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, *211*, 253–273.

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

Carrasco, R., & Oncina, J. (1994). Learning stochastic regular grammars by means of a state merging method. *Proceedings of the 2nd International Colloquium on Grammatical Inference* (pp. 139–150).

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.

de la Higuera, C. (1997). Characteristic sets for polynomial grammatical inference. *Machine Learning*, *27*.

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

Gold, E. M. (1978). Complexity of automaton identification from given data. *Information and Control*, *37*, 302–320.

Grinchtein, O., Jonsson, B., & Leucker, M. (2004). Learning of event-recording automata. *Lecture Notes In Computer Science*, *3253*, 379–395.

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. *ICGI* (pp. 1–12).

Oncina, J., & Garcia, P. (1992). Inferring regular languages in polynomial update time. In *Pattern recognition and image analysis*, vol. 1 of *Series in Machine Perception and Artificial Intelligence*, 49–61. World Scientific.

Sen, K., Viswanathan, M., & Agha, G. (2004). Learning continuous time Markov chains from sample executions. *Proceedings of The Quantitative Evaluation of Systems* (pp. 146–155).

Sipser, M. (1997). *Introduction to the theory of computation*. PWS Publishing.