Learning Mealy Machines with One Timer

Frits Vaandrager[1](✉), Roderick Bloem[2](✉), and Masoud Ebrahimi[2](✉)

[1] Radboud University, Nijmegen, Netherlands
f.vaandrager@cs.rul.nl

[2] Graz University of Technology, Graz, Austria
{roderick.bloem,masoud.ebrahimi}@iaik.tugraz.at

Abstract. We present Mealy machines with a single timer (MM1Ts), a class of models that is both sufficiently expressive to describe the real-time behavior of many realistic applications, and can be learned efficiently. We show how learning algorithms for MM1Ts can be obtained via a reduction to the problem of learning Mealy machines. We describe an implementation of an MM1T learner on top of LearnLib, and compare its performance with recent algorithms proposed by Aichernig et al. and An et al. on several realistic benchmarks.

1 Introduction

Model learning, also known as active automata learning, is a black-box technique for constructing state machine models of software and hardware components from information obtained through testing (i.e., providing inputs and observing the resulting outputs). Model learning has been successfully used in numerous applications, for instance for spotting bugs in implementations of major network protocols. e.g.. in [5–8,20]. We refer to [13,24] for surveys and further references.

Timing plays a crucial role in many applications. A TCP server, for instance, may retransmit packets if they are not acknowledged within a specified time. Also, a timeout will occur if a TCP server does not receive an acknowledgment after a number of retransmissions, or if it remains in certain states too long. Timing behavior cannot be captured using existing learning tools, which typically only support learning of deterministic finite automata (DFAs) and related models. In the case of TCP, previous work only succeeded to learn models of real implementations by having the network adaptor ignore all retransmissions, and by completing learning queries before the occurrence of certain timeouts [8]. All timing issues had to be artificially suppressed.

This work was supported by the Austrian Research Promotion Agency (FFG) through project TRUSTED (867558), Graz University of Technology's LEAD project "Dependable Internet of Things in Adverse Environments" and by Radboud University's NWO TOP project 612.001.852 "Grey-box learning of Interfaces for Refactoring Legacy Software (GIRLS)".

The challenge to extend model learning algorithms to a setting of timed systems has been addressed by several authors. Most proposals aim to develop learning algorithms for the popular framework of timed automata [2], which extends DFAs with clock variables. Transitions of timed automata may contain both guards that test the values of clocks, and resets that update the clocks. Since guards and resets are not directly observable in a black-box setting, this poses major challenges during learning. Grinchtein et al. [9,10] developed learning algorithms for deterministic event-recording automata (DERAs), which have a clock for each action in the alphabet, and where each transition resets the clock corresponding to its input action. This restriction makes resets observable, but the complexity of the resulting algorithms still appears to be prohibitively high, due to the difficulties of inferring guards. The restrictions of DERAs also make it hard to capture the timing behavior of common network protocols. For instance, a pattern that often occurs is that within t time units after an event a there should be an event b. (For instance, in TCP a SYN should be followed by a SYN-ACK within a specified time interval.) In a DERA, upon occurrence of two consecutive a's, the automaton no longer remembers when the first a has occurred, and can thus not ensure the occurrence of a timeout at the required moment in time. Recently, Henry et al. [11] proposed a learning algorithm for a slightly larger class of reset-free DERAs, where some transitions may reset no clocks. Even though this algorithm appears to be more efficient than those of [9,10], it still suffers from a combinatorial blow up because, for each transition, it has to guess whether this transition resets a clock. An et al. [3] developed a learning algorithm for deterministic one-clock timed automata (DOTAs), using a brute force approach to reset guessing, also leading to a combinatorial blow up. Entirely different, heuristic algorithms are proposed recently by Aichernig et al. [1,23], using genetic programming. They succeeded to learn timed automata models with one clock for several industrial benchmarks.

Given the difficulties to infer the guards and resets of timed automata, the question arises whether timed automata provide the right modeling framework to support learning algorithms. As an alternative, we propose to consider the use of *timers* instead of *clocks*. The difference is that the value of a timer decreases when time advances, whereas the value of a clock increases. In a setting with clocks, guards and invariants are required to constrain the timing of events, but a timer simply triggers a timeout whenever its value becomes 0. The absence of guards and invariants makes model learning much easier in a setting with timers. A learner still has to figure out which transitions set a timer, but this also becomes easier and does not create a combinatorial blow-up. If a transition sets a timer then slight changes in the timing of this transition will trigger corresponding changes in the timing of the resulting timeout, allowing a learner to figure out the exact cause of each timeout event. DFAs with timers are strictly less expressive than timed automata if we assume that timeout events can be observed. For many realistic applications, however, this reduced expressivity causes no problems. Kurose and Ross [15], for instance, use finite state machine models with timers to explain transport layer protocols. Caldwell et al. [4]

propose a learning algorithm for a simple class of automata with timers, which they call time delay Mealy machines. These machines have only a single timer, which is reset on every transition. As a result, time delay Mealy machines are not sufficiently expressive to capture the timing behavior of realistic network protocols.

In this paper, we present Mealy machines with a single timer (MM1Ts), a class of models that is both sufficiently expressive to describe the real-time behavior of many realistic applications, and can be learned efficiently. In an MM1T, the timer can be set to integer values on transitions, and may be stopped or time out in later transitions. Each timeout triggers an observable output, allowing a learner to observe the occurrence of timeouts. We show how learning algorithms for MM1Ts can be obtained via a reduction to the problem of learning Mealy machines. We describe an implementation of an MM1T learner on top of LearnLib, a state-of-the-art tool for learning Mealy machines [17], and compare its performance with the tools of Aichernig et al. [1] and An et al. [3] on several benchmarks: TCP connection setup, Android's Authentication and Key Management (AKM) service, and some industrial benchmarks taken from [1]. Our implementation outperforms the tool of [1] with several orders of magnitude in terms of the total number of input symbols required to learn a model. The tool of [3] is only able to learn the benchmarks with a "helpful" teacher that provides information about resets; without help, it is unable to learn the benchmarks.

2 Mealy Machines with a Single Timer

In this section, we introduce the notion of Mealy machines with a single timer (MM1T). We write $f : X \rightharpoonup Y$ to denote that f is a partial function from X to Y. We write $f(x) \downarrow$ to mean that the result is defined for x, that is, $\exists y : f(x) = y$, and $f(x) \uparrow$ if the result is undefined. We often identify a partial function f with the set of pairs $\{(x, y) \in X \times Y \mid f(x) = y\}$.

MM1Ts are just regular (deterministic) Mealy machines, augmented with a timer that can be switched on and off, a timeout input, and a function that specifies how transitions affect the timer. We view timeout's as input events, a choice that makes sense if we view the hardware clock (or whatever the device is that triggers timeout interrupts) as part of the environment of the machine.

Definition 1. *A* Mealy machine with a single timer (MM1T) *is defined as a tuple* $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda, \tau)$, *where*

- I *is a finite set of* inputs*, containing a special element* timeout*,*
- O *is a finite set of* outputs*,*
- $Q = Q_{\mathit{off}} \cup Q_{on}$ *is a finite set of* states*, partitioned into subsets where the timer is on and off, respectively;* $q_0 \in Q_{\mathit{off}}$ *is the* initial state*,*
- $\delta : Q \times I \rightharpoonup Q$ *is a* transition function*, satisfying*

$$\delta(q, i) \uparrow \quad \Leftrightarrow \quad i = \mathsf{timeout} \wedge q \in Q_{\mathit{off}} \tag{1}$$

(inputs are always defined, except for timeout *in states where timer is off),*

– $\lambda : Q \times I \rightharpoonup O$ *is an* output function*, satisfying*

$$\lambda(q,i)\downarrow \;\Leftrightarrow\; \delta(q,i)\downarrow \tag{2}$$

(each transition has both an input and an output),

– $\tau : Q \times I \rightharpoonup \mathbb{N}^{>0}$ *is a* reset function*, satisfying*

$$\tau(q,i)\downarrow \;\Rightarrow\; \delta(q,i) \in Q_{on} \tag{3}$$

$$q \in Q_{\mathit{off}} \wedge \delta(q,i) \in Q_{on} \Rightarrow \tau(q,i)\downarrow \tag{4}$$

$$\delta(q,\mathsf{timeout}) \in Q_{on} \;\Rightarrow\; \tau(q,\mathsf{timeout})\downarrow \tag{5}$$

(when a transition (re)sets the timer, the timer is on in the target state; when it moves from a state where the timer is off to a state where the timer on, it sets the timer; if the timer stays on after a timeout, it is reset).

Let $\delta(q,i) = q'$ *and* $\lambda(q,i) = o$*. We write* $q \xrightarrow{i/o,n} q'$ *if* $\tau(q,i) = n \in \mathbb{N}^{>0}$*, and* $q \xrightarrow{i/o,\perp} q'$ *or just* $q \xrightarrow{i/o} q'$ *if* $\tau(q,i)\uparrow$*.*

Example 1. The MM1T shown in Fig. 1 is a simplified model of the sender from the alternating-bit protocol, adapted from [15, Figure 3.15]. We write *set-timer*(n) on the i-transition from state q to indicate that $\tau(q,i) = n$. The MM1T has four states, with $Q_{on} = \{q_1, q_3\}$ and $Q_{\mathit{off}} = \{q_0, q_2\}$. In the model, input *in* corresponds to a request from the upper layer to transmit data. Initially, upon receipt of such a request, the sender builds a packet from the data and a sequence number 0, sends this over the network (output *send0*), and starts the timer with timeout value 3. When the sender receives an acknowledgement with the correct sequence number 0 (input *ack0*) it stops the timer and jumps to state q_2 without generating visible output (*void*). Acknowledgement with the incorrect sequence number (input *ack1*) are ignored. Likewise, inputs *in* in state q_1 and acknowledgements in state q_0 are ignored (for readability, these transitions are not shown in the diagram). If no *ack0* input arrives within 3 timeunits, a timeout occurs and the same packet is retransmitted. The behavior in states q_2 and state q_3 is symmetric to that in states q_0 and q_1, respectively, except that the roles of sequence numbers 0 and 1 is swapped.

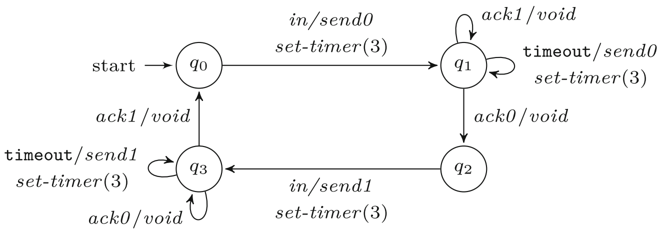

Fig. 1. MM1T model of alternating-bit protocol sender.

Semantics. We give two semantics for MM1Ts, an untimed and a timed one. In the untimed semantics, we just record the labels of sequences of transitions. Formally, an *untimed word* over inputs I and outputs O is a sequence

$$w = (i_0, o_0, n_0), (i_1, o_1, n_1) \cdots (i_k, o_k, n_k),$$

where each $i_j \in I$, each $o_j \in O$, and each $n_j \in \mathbb{N}^{>0} \cup \{\bot\}$ is a timer value. An *untimed run* of MM1T $\mathcal{M}$ over w is a sequence

$$\alpha = q_0 \xrightarrow{i_0/o_0, n_0} q_1 \xrightarrow{i_1/o_1, n_1} q_2 \cdots \xrightarrow{i_k/o_k, n_k} q_{k+1}$$

of transitions of $\mathcal{M}$ such that all q_j are states of $\mathcal{M}$ and q_0 is the initial state. Note that, since MM1Ts are deterministic, for each untimed word w there is at most one untimed run over w. We say that w is an untimed word of $\mathcal{M}$ iff $\mathcal{M}$ has an untimed run over w. MM1Ts $\mathcal{M}$ and $\mathcal{N}$ with the same set of inputs are *untimed equivalent*, $\mathcal{M} \approx_{untimed} \mathcal{N}$, iff they have the same untimed words.

The timed semantics, which is slightly more involved, describes the real-time behavior of a MM1T. It associates an infinite state transition system to a MM1T that describes all possible configurations and transitions between them. A *configuration* of a MM1T is a pair (q, t), where $q \in Q$ is a state and $t \in \mathbb{R}^{\geq 0} \cup \{\infty\}$ specifies the value of the timer. We require $t = \infty$ iff $q \in Q_{\mathit{off}}$. We refer to (q_0, ∞) as the *initial configuration*. Using four rules we define a transition relation that describes how one configuration may evolve into another. For all $q \in Q$, $r \in Q_{\mathit{off}}$, $s, s' \in Q_{on}$, $i \in I$, $o \in O$, $t \in \mathbb{R}^{\geq 0} \cup \{\infty\}$, $d \in \mathbb{R}^{\geq 0}$ and $n \in \mathbb{N}^{>0}$,

$$\frac{d \leq t}{(q,t) \xrightarrow{d} (q, t-d)} \quad (6) \qquad \frac{q \xrightarrow{i/o, n} s, \quad i = \mathsf{timeout} \Rightarrow t = 0}{(q,t) \xrightarrow{i/o} (s, n)} \quad (7)$$

$$\frac{q \xrightarrow{i/o} r, \quad i = \mathsf{timeout} \Rightarrow t = 0}{(q,t) \xrightarrow{i/o} (r, \infty)} \quad (8) \qquad \frac{s \xrightarrow{i/o} s', \quad i \neq \mathsf{timeout}}{(s,t) \xrightarrow{i/o} (s', t)} \quad (9)$$

Rule (6) states that the value of the timer decreases proportionally when time advances, until it becomes 0. Here we use the convention that $\infty - d = \infty$, for any $d \in \mathbb{R}^{>0}$. So when the timer is off, time may advance indefinitely. Rule (7) describes events where the timer is (re)set; a timeout may occur only when the timer has expired in the source state. Rule (8) describes events where the timer is off in the target state; again, a timeout may occur only when the timer has expired in the source state. Finally, rule (9) describes events where the timer remains on and is not reset.

A *timed word* over inputs I and outputs O is a sequence

$$w = (t_0, i_0, o_0), (t_1, i_1, o_1) \cdots (t_k, i_k, o_k),$$

where each $i_j \in I$, each $o_j \in O$, and each $t_j \in \mathbb{R}^{\geq 0}$. A timed word w describes a behavior that an experimenter may observe when interacting with an MM1T:

after an initial delay of t_0 time units, input i_0 is applies which triggers output o_0, after a subsequent delay of t_1 time units, input i_1 is applied, etc. For such a timed word w, a *timed run* of MM1T $\mathcal{M}$ over w is a sequence

$$\alpha = C_0 \xrightarrow{t_0} C_0' \xrightarrow{i_0/o_0} C_1 \xrightarrow{t_1} C_1' \xrightarrow{i_1/o_1} C_2 \cdots \xrightarrow{t_k} C_k' \xrightarrow{i_k/o_k} C_{k+1}$$

of transitions of $\mathcal{M}$ such that all C_j, C_j' are configurations of $\mathcal{M}$ and C_0 is the initial configuration. Since MM1Ts are deterministic, for each timed word w there exists at most one run over w. We say w is a timed word of $\mathcal{M}$ if there exists a run of $\mathcal{M}$ over w. MM1Ts $\mathcal{M}$ and $\mathcal{N}$ with the same set of inputs are *timed equivalent*, $\mathcal{M} \approx_{timed} \mathcal{N}$, iff they have the same sets of timed words.

Although the definitions are quite different, it turns out that timed and untimed equivalence coincide.

Theorem 1. $\mathcal{M} \approx_{timed} \mathcal{N} \Leftrightarrow \mathcal{M} \approx_{untimed} \mathcal{N}$

3 Learning MM1Ts

It will be useful to explore this connection between the timed and untimed semantics in some more detail, because this will allow us to reuse existing active learning algorithms for untimed systems [18,21] for learning MM1Ts.

3.1 From MM1Ts to Mealy Machines and Back

MM1Ts generalize the classical notion of a Mealy machine: essentially, a Mealy machine is just an MM1T in which the timer is off in all states. Conversely, each MM1T can be viewed as a Mealy machine of a special form.

Definition 2. *A* Mealy machine *is a tuple $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda)$, where I is a finite set of inputs, O a set of outputs, Q a finite set of states, $q_0 \in Q$ the initial state, $\delta : Q \times I \to Q$ a transition function, and $\lambda : Q \times I \to O$ an output function. We generalize the transition function to sequences of inputs as usual. Function $mq_{\mathcal{M}} : I^+ \to O$ assigns to each sequence of inputs the final output: $mq_{\mathcal{M}}(\sigma i) = \lambda(\delta(q_0, \sigma), i)$. Mealy machines $\mathcal{M}$ and $\mathcal{N}$ with the same set of inputs I are* equivalent*, denoted by $\mathcal{M} \approx \mathcal{N}$, if for all $\sigma \in I^+$, $mq_{\mathcal{M}}(\sigma) = mq_{\mathcal{N}}(\sigma)$.*

We associate a Mealy machine $\mathsf{Mealy}(\mathcal{M})$ to each MM1T $\mathcal{M}$ as follows. We keep the same states, inputs and transitions, but add $\mathsf{timeout}$ self-loops for each state in Q_{off} to make the Mealy machine input enabled. We introduce a fresh output nil and associate this special output to each new $\mathsf{timeout}$ self-loop. The outputs of the other transitions of $\mathsf{Mealy}(\mathcal{M})$ are pairs consisting of the output from $\mathcal{M}$ and the timer update.

Definition 3. *Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda, \tau)$ be a MM1T. Then* $\mathsf{Mealy}(\mathcal{M})$ *is the Mealy machine* $(I, (O \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}, Q, q_0, \delta', \lambda')$, *where*

$$\delta'(q,i) = \begin{cases} \delta(q,i) & \textit{if } \lambda(q,i)\downarrow \\ q & \textit{otherwise} \end{cases}$$

$$\lambda'(q,i) = \begin{cases} (\lambda(q,i), \tau(q,i)) & \textit{if } \tau(q,i)\downarrow \\ (\lambda(q,i), \bot) & \textit{if } \lambda(q,i)\downarrow \textit{ and } \tau(q,i)\uparrow \\ \mathsf{nil} & \textit{otherwise} \end{cases}$$

Conversely, suppose that $\mathcal{N} = (I, (O \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}, Q, q_0, \delta', \lambda')$ *is a Mealy machine. Then we may reverse the above construction and define a tuple* $\mathsf{MM1T}(\mathcal{N}) = (I, O, Q, q_0, \delta, \lambda, \tau)$ *in the obvious way.*

The following result, which follows from the definitions and Theorem 1, asserts that Mealy and MM1T act like adjoint operators.

Theorem 2. *Let $\mathcal{M}$ be a MM1T and let $\mathcal{N}$ be a Mealy machine such that* $\mathsf{Mealy}(\mathcal{M}) \approx \mathcal{N}$. *Then* $\mathsf{MM1T}(\mathcal{N})$ *is a MM1T and* $\mathcal{M} \approx_{timed} \mathsf{MM1T}(\mathcal{N})$.

Theorem 2 suggests that we can obtain a learner for MM1Ts from a learner for Mealy machines. To achieve this, we place an *adaptor* between a Mealy machine learner and a System Under Learning (SUL) that behaves like MM1T $\mathcal{M}$. From the perspective of the Mealy machine learner, the adaptor behaves like a teacher for $\mathsf{Mealy}(\mathcal{M})$ that answers membership and equivalence queries. In order to answer these queries, the adaptor interacts with the SUL and observes timed words of $\mathcal{M}$. When the learner has succeeded to learn a Mealy machine $\mathcal{N}$ that is equivalent to $\mathsf{Mealy}(\mathcal{M})$, we know by Theorem 2 that $\mathcal{M} \approx_{timed} \mathsf{MM1T}(\mathcal{N})$, and so we have learned a MM1T that is equivalent to $\mathcal{M}$. Effectively, the combination of the adaptor and the Mealy machine learner acts as an MM1T learner.

We implemented an adaptor that interacts with LearnLib [18] so we can benefit from all optimizations already integrated into this well maintained automata learning library. Our adaptor is available online[1]. Below we describe how to implement a membership oracle for learning MM1Ts. An equivalence oracle can be implemented in a similar manner, and is not discussed here for reasons of space.

3.2 Membership Queries

In order to answer membership queries, the adaptor maintains an observation tree defined as follows.

Definition 4. *Let $\mathcal{M}$ be a MM1T. An* observation tree *for $\mathcal{M}$ is a triple $\mathcal{T} = (S, mq, timer)$, where $S \subset I^*$ is a non empty, finite, prefix closed set of input sequences, referred to as* nodes, *$mq : S \backslash \{\epsilon\} \to O \times (\mathbb{N}^{>0} \cup \{\bot\})$ is a node labeling function, and $timer : S \to \{on, off\}$ is a function that specifies whether the timer is on or off in a node. We require that $timer(\epsilon) = off$ and $\sigma \cdot \mathsf{timeout} \in S \Rightarrow timer(\sigma) = on$.*

[1] https://extgit.iaik.tugraz.at/scos/scos.sources/LearningMMTs.

Initially, the adaptor starts with a trivial observation tree with a single node ϵ and $timer(\epsilon) = off$. The observation tree is then extended one node at a time. For this, the adaptor maintains a maximum timer value Δ. Initially, Δ can be assigned some arbitrary value in $\mathbb{N}$. Suppose that $\sigma = i_1 \cdots i_{k-1}$ is a leaf node of observation tree $\mathcal{T}$, $i_k \in I$ and $i_k = \mathsf{timeout} \Rightarrow timer(\sigma) = on$. In order to add node $\sigma \cdot i_k$ to $\mathcal{T}$, the adaptor resets the SUL and then eagerly applies $\sigma \cdot i_k$. That is, for each $j \in [1, k]$, the adaptor processes input i_j as follows:

- if $i_j \neq \mathsf{timeout}$, the adaptor feeds the input to the SUL without any delay,
- otherwise, the oracle waits for the timeout event.

The immediate response o after feeding i_k accounts for the output value that will be recorded in $mq(\sigma \cdot i_k)$. Next the adaptor waits for Δ time units. If a timeout occurs after $n < \Delta$ time units then the value of $timer(\sigma \cdot i_k)$ is set to on, otherwise it is set to off. If $timer(\sigma \cdot i_k) = off$ then we set $mq(\sigma \cdot i_k) = (o, \bot)$. Otherwise, the adaptor performs another experiment to decide whether the clock was set on the last transition or before:

- it resets the SUL and eagerly applies σ,
- it waits for 1/2 time unit and then applies input i_k,
- it then waits until a timeout event occurs at time $n' \leq n$.
- If $n' = n$ it sets $mq(\sigma \cdot i_k) = (o, n)$, otherwise it sets $mq(\sigma \cdot i_k) = (o, \bot)$.

Once the observation tree $\mathcal{T}$ is big enough, the adaptor can answer a membership query σ by computing the sequence σ' obtained by omitting spurious $\mathsf{timeouts}$ from σ, that is, $\mathsf{timeouts}$ from nodes of $\mathcal{T}$ where the timer is off. If σ ends with a spurious $\mathsf{timeout}$ then the response of the adaptor is nil, otherwise it is $mq(\sigma')$.

Query Complexity. Note that in order to add a new node to the observation tree, we need one or two experiments (membership queries) on the MM1T (SUL), depending whether the timer is on in the target node. Thus, starting from the trivial observation tree, we will need at most $2n$ membership queries on the MM1T to implement a single membership query with n input symbols by LearnLib, with a total number of inputs in $O(n^2)$. This way of learning MM1Ts has a higher query complexity than learning Mealy machines, but the growth of the total number of input symbols required is still polynomial. If the number of states where the timer is on is low, the query complexity is comparable.

Learning the Maximum Timer Value. If the maximum timer value Δ is greater than or equal to the SUL's maximum timer value, no timeout event will be missed during learning a hypothesis. Otherwise, the equivalence oracle will at some point return a counterexample containing a timeout event that is not present in the observation tree. Based on this counterexample, we then update Δ and start learning from scratch.

4 From MM1T to DOTA Learning

In order to compare our approach to those of [1,3], we translate MM1Ts to Deterministic One-Clock Timed Automata (DOTAS). In the interest of brevity, we will not formalize this transition, but rather illustrate it with an example. We will construct a DOTA of the alternating-bit protocol from Fig. 1; the result can be found in Fig. 2.

Edges of DOTAs are labeled with an action, a clock guard that is an interval on allowed clock values, and a Boolean that states whether to reset the clock to zero. The set of actions consists of all input labels of the MM1T (except `timeout`) prefixed with '?' and all output labels (except *void*) prefixed with '!'. In general, DOTAs have accepting and non-accepting states, we will construct DOTAs with only accepting states.

We split each transition of the MM1T into an input and an output transition. For instance, we encode the transition $q_0 \rightarrow q_1$ into transitions $q_0' \rightarrow l_0$ and $l_0 \rightarrow q_1'$. In this case, the input transition can be taken at any time and it resets the clock, causing the output transition to be taken immediately. For edges with a *void* output, we omit the output transition. Thus, the self loop on q_1 labeled *ack1/void* is represented by a self-loop on q_1' in the DOTA.

An MM1T transition that sets the timer is replaced by a DOTA transition that reset the clock and appropriate clock guards on subsequent states. For instance, the transition $q_0 \rightarrow q_1$ sets the timer to 3; thus, a `timeout` event will occur in q_1 at time 3, causing a *send*. In the DOTA, this is reflected by a clock reset on the transition $l_0 \rightarrow q_1'$ and a clock guard with value $[3, \infty)$ on the self loop on q_1' labeled *!send*.

The rest of the translation follows along the same lines.

5 Case Studies

5.1 Learning Setup

We instantiated L_M^* and TTT using the MM1T membership oracle. For counterexample processing we used *Rivest and Shapire's* method [19]. We close tables using *close shortest* strategy. Finally, we use a *random word* equivalence oracle with 1000 tests and word length of minimum 4 and of maximum 11. For further details on above terminologies, we refer readers to LearnLib's documentation.

5.2 Android Authentication and Key Management

To show that our algorithm can learn realistic Mealy machines with timers, we used our algorithm to learn the Authentication and Key Management of the WiFi implementation of a Huawei Mate10-lite running Android 8.0.0 (Kernel 4.4.23+) with a security patch dated July 5, 2019. The IEEE 802.11 standard gives an abstract automaton of an Authentication and Key Management (AKM) service in [14, p. 1643]. The automaton has a state that encapsulates a 4-way handshake mechanism granting access to the controlled port. Since learning the 4-way handshake mechanisms is already addressed in [22], we focus on learning the AKM service. We used the following management frames: Auth(Open), AssoReq, Deauth(leaving), Disas(leaving), ProbeReq, and timeout [14, p. 45–49].

Our learning experiments resulted in the MM1T shown in Fig. 3. The SUL deviates from the specified standards in the following ways.

Disassociation: The reference prescribes that a disassociation (Disas) terminates an established association but maintains authentication. In the learned model (state q_2), a disassociation instead drops both the established association and the authentication. To correct this, the access point should transit to q_1 when disassociating in q_2 (red transitions from q_2 must go to q_1).

Association Timeout: Along the red transition from q_1 to q_2, SUL does not include *BSS Max Idle Period* element in *AssoResp* frames. Yet, it implements an association timeout event, which violates the specification. To confirm this, we manually inspected the Android 8.0.0 (r39) source code, which excludes the element mentioned above except for access points of Wireless Mesh Networks.

5.3 Performance Comparison

We apply our learning method for MM1Ts to a set of real-world benchmarks. This demonstrates the expressiveness of MM1Ts, and shows the practicality of our implementation.

Benchmarks: Our benchmark set consists of the AKM (Sect. 5.2), the TCP Connection State Diagram ([16, p. 23]), a car alarm system (CAS) [1], and a particle counter (PC) [1]. For the TCP benchmark, we used the one timeout on the transmission control block indicated in the diagram in the RFC. See Table 1 for statistics on the size of the benchmarks.

Algorithms: Table 2 shows benchmark results for MM1T and DOTA learning algorithms. GTALearn represents the learning algorithm by Aichernig et al. [1]. OTALearn* represents the learning algorithm by An et al. [3] using a "smart" teacher that provides the clock reset information. Finally, OTALearn uses a normal teacher and timed out on all benchmarks.

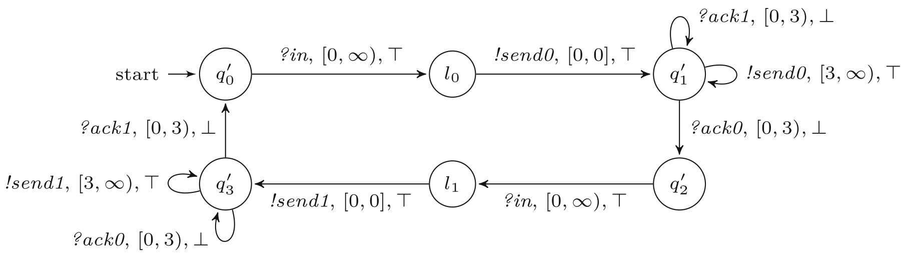
start
q'_0
$?in$, $[0,\infty)$, $\top$
l_0
$!send0$, $[0,0]$, $\top$
q'_1
$?ack1$, $[0,3)$, $\bot$
$!send0$, $[3,\infty)$, $\top$
$?ack0$, $[0,3)$, $\bot$
q'_2
$?in$, $[0,\infty)$, $\top$
l_1
$!send1$, $[0,0]$, $\top$
q'_3
$!send1$, $[3,\infty)$, $\top$
$?ack0$, $[0,3)$, $\bot$
$?ack1$, $[0,3)$, $\bot$

Fig. 2. DOTA model of alternating-bit protocol sender.

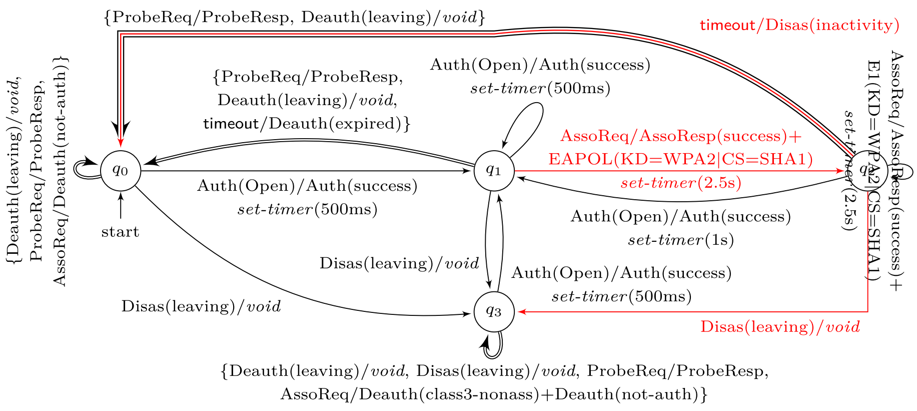
{ProbeReq/ProbeResp, Deauth(leaving)/void}
timeout/Disas(inactivity)
{Deauth(leaving)/void, ProbeReq/ProbeResp, AssoReq/Deauth(not-auth)}
{ProbeReq/ProbeResp, Deauth(leaving)/void, timeout/Deauth(expired)}
Auth(Open)/Auth(success) set-timer(500ms)
AssoReq/AssoResp(success)+ E1(KD=WPA2|CS=SHA1) set-timer(2.5s)
q0
q1
q2
q3
start
AssoReq/AssoResp(success)+ EAPOL(KD=WPA2|CS=SHA1) set-timer(2.5s)
Auth(Open)/Auth(success) set-timer(500ms)
Auth(Open)/Auth(success) set-timer(1s)
Disas(leaving)/void
Auth(Open)/Auth(success) set-timer(500ms)
Disas(leaving)/void
Disas(leaving)/void
{Deauth(leaving)/void, Disas(leaving)/void, ProbeReq/ProbeResp, AssoReq/Deauth(class3-nonass)+Deauth(not-auth)}

Fig. 3. MM1T of a Huawei Mate10-lite that captures granting uncontrolled port. Double and triple edges represent a set of transitions. We rounded timer values to the nearest 500 ms and marked specification violations with the color red.

Table 1. Benchmarks in terms of state-space (|**S**|) and input size (|**Σ**|).

Model	AKM		TCP		CAS		PC	
	|**S**|	|**Σ**|	|**S**|	|**Σ**|	|**S**|	|**Σ**|	|**S**|	|**Σ**|
MM1T	4	5	11	8	8	4	8	8
DOTA	15	12	20	13	14	10	26	14

Performance Metrics: Since OTALearn* implements its own equivalence checker and GTALearn is not an Angluin style algorithm, we report the total number of resets (#**R**), and inputs (#**I**) performed rather than the number of membership and equivalence queries. We believe this gives a fair comparison of the algorithms under the assumption that most time is spent executing the SUL.

AKM has a more sophisticated timed behavior than the other benchmarks, which explains the higher number of resets for MM1T learners. Meanwhile, if considering number of inputs, OTALearn* straggles by an order of magnitude. GTALearn shows a competitive performance in the number of inputs performed that indicates the potentials of this novel approach.

TCP has only one timeout transition; thus, the learning algorithms do not need to reset the SUL as often. (With the exception of OTALearn.) L^*_M learns the MM1T for TCP in one round, while TTT requires 8 rounds, which justifies the better performance of L^*_M. MM1T learners outperform those for DOTAs by nearly an order of magnitude when considering the number of inputs performed.

CAS and PC show a slightly more sophisticated timed behavior than TCP. For both, the MM1T algorithms also significantly outperform the algorithms for DOTA. Similarly, if considering the inputs performed, OTALearn* straggles by three orders of magnitude.

Table 2. Benchmark results in terms of total resets and total performed inputs.

Algorithm	AKM		TCP		CAS		PC	
	#R	#I	#R	#I	#R	#I	#R	#I
MM1T-L^*_M	5587	35002	413	2401	613	4822	408	2271
MM1T-TTT	5714	35948	640	4773	623	4978	369	2443
GTALEARN	2626	36411	1186	33779	1609	30870	3368	33824
OTALEARN*	2103	356762	2924	86880	1448	3791091	10003	3540458
OTALEARN	timeout		timeout		timeout		timeout	

6 Conclusion and Future Work

Timers are commonly used in software to enforce real-time behavior, and so it is natural to use them in formal models. We presented a framework of Mealy machines with a single timer and showed how a learning algorithm can be obtained via reduction to the problem of learning Mealy machines. Our approach assumes that timers are set when input events occur, and timeouts trigger instantaneous outputs. While these assumptions do not always hold, there are many real-time systems for which the delays between timer events and observable inputs and outputs are negligible, and the assumptions are justified. We evaluated our approach on a number of realistic applications, and showed that it outperforms the approaches of Aichernig [1] et al. and An et al. [3].

An obvious direction for future research is to extend our work to Mealy machines with multiple timers. We expect that a learning algorithm can be developed, but a simple reduction to Mealy machine learning is no longer possible. It would be interesting to apply the genetic programming approach of [1] in a setting of Mealy machines with timers. Since it no longer needs to learn transition guards, one may expect that a genetic algorithm will converge faster. Of course, as noted by [3], we may resort to grey-box techniques for model learning [12] to obtain efficient learning algorithms for real-time software. However, this forces us to deal with numerous programming language specific details. Black-box techniques can be applied without knowledge of the underlying hardware/software, which makes it important to push these techniques to their limits.

Acknowledgement. We would like to thank Andrea Pferscher and Miaomiao Zhang for help with running the benchmarks on their tools [1,3].

References

1. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 1–19. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_1

2. Alur, R., Dill, D.: A theory of timed automata. Theoret. Comput. Sci. **126**, 183–235 (1994)
3. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. TACAS 2020. LNCS, vol. 12078, pp. 444–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_25
4. Caldwell, B., Cardell-Oliver, R., French, T.: Learning time delay mealy machines from programmable logic controllers. IEEE Trans. Autom. Sci. Eng. **13**(2), 1155–1164 (2015)
5. Fiterău-Broştean, P., Howar, F.: Learning-based testing the sliding window behavior of TCP implementations. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) FMICS/AVoCS -2017. LNCS, vol. 10471, pp. 185–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_12
6. Fiterău-Broştean, P., Jonsson, B., Merget, R., de Ruiter, J., Sagonas, K., Somorovsky, J.: Analysis of DTLS implementations using protocol state fuzzing. In: USENIX Security Symposium. USENIX Association (2020)
7. Fiterău-Broştean, P., Lenaerts, T., et al.: Model learning and model checking of SSH implementations. In: SPIN Symposium. ACM (2017)
8. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 454–471. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_25
9. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theoret. Comput. Sci. **411**(47), 4029–4054 (2010)
10. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_29
11. Henry, L., Jéron, T., Markey, N.: Active learning of timed automata with unobservable resets. In: Bertrand, N., Jansen, N. (eds.) FORMATS 2020. LNCS, vol. 12288, pp. 144–160. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_9
12. Howar, F., Jonsson, B., Vaandrager, F.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G. (eds.) Computing and Software Science. LNCS, vol. 10000, pp. 563–588. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_26
13. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 123–148. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_5
14. IEEE: Std 802.11-2016 (Revision of IEEE Std 802.11-2012): Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications (2016)
15. Kurose, J.F., Ross, K.W.: Computer Networking: A Top-Down Approach, 6th edn. Pearson, London (2013)
16. Postel, J.E.: Transmission Control Protocol. RFC 793, September 1981
17. Raffelt, H., Steffen, B., Berg, T.: LearnLib: a library for automata learning and experimentation. In: FMICS 2005. ACM Press (2005)
18. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT **11**(5), 393–407 (2009)
19. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences (extended abstract). In: ACM Symposium on Theory of Computing. ACM (1989)

20. Ruiter, J.d., Poll, E.: Protocol state fuzzing of TLS implementations. In: USENIX Security Symposium. USENIX (2015)
21. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_8
22. McMahon Stone, C., Chothia, T., de Ruiter, J.: Extending automated protocol state learning for the 802.11 4-way handshake. In: Lopez, J., Zhou, J., Soriano, M. (eds.) ESORICS 2018. LNCS, vol. 11098, pp. 325–345. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99073-6_16
23. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn – learning timed automata from tests. In: André, É., Stoelinga, M. (eds.) FORMATS 2019. LNCS, vol. 11750, pp. 216–235. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29662-9_13
24. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)