Information and Computation 295 (2023) 105013

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Learning Mealy machines with one timer ☆

Frits Vaandrager [a,*], Masoud Ebrahimi [b,*], Roderick Bloem [b]

[a] Radboud University, Nijmegen, Netherlands
[b] Graz University of Technology, Graz, Austria

ARTICLE INFO

Article history:
Received 6 July 2021
Received in revised form 8 February 2023
Accepted 12 February 2023
Available online 20 February 2023

ABSTRACT

We present Mealy machines with a single timer (MM1Ts), a class of sufficiently expressive models to describe the real-time behavior of many realistic applications that we can learn efficiently. We show how we can obtain learning algorithms for MM1Ts 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 [4–9]. We refer to [10,11] for surveys and further references.

Timing plays a crucial role in many applications. A TCP server, for instance, may retransmit packets if the recipient does not acknowledge them within a specified time. Also, a timeout will occur if a TCP server does not receive an acknowledgment after several retransmissions or if it remains in certain states for too long. Existing learning tools cannot capture timing behavior. These tools typically only support the learning of deterministic finite automata (DFAs) and related models. In the case of TCP, previous work only succeeded in learning models of real implementations by having the network adaptor ignore all retransmissions, and by completing learning queries before the occurrence of certain timeouts [5]. All timing issues had to be artificially suppressed.

Several authors already addressed the challenge of extending model learning algorithms to a setting of timed systems. Most proposals aim to develop learning algorithms for the popular framework of timed automata [12], which extends DFAs with clock variables. Transitions of timed automata may contain both guards that test clocks' values 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.

☆ 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)". This article is an extended version of [1]. We have included missing proofs that were left out due to the page limit, describe a way to implement untimed membership queries with fewer timed membership queries, improved the exposition of this key part of our approach (in section 3.4), and improved the experimental evaluation of our new algorithm and the comparison with the related approaches of Aichernig et al. [2] and An et al. [3].

* Corresponding authors.
E-mail address: ebrahimi@tugraz.at (M. Ebrahimi).

https://doi.org/10.1016/j.ic.2023.105013

Grinchtein et al. [13,14] 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 clock 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 standard 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 the occurrence of two consecutive a's, the automaton no longer remembers when the first a has occurred; thus, it cannot ensure the occurrence of a timeout at the required moment in time.

Recently, Henry et al. [15] 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 [13,14], 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. [16,2], using genetic programming. They succeeded in learning timed automata models with multiple clocks for several industrial benchmarks.

Given the difficulties in inferring the guards and resets of timed automata, the question arises whether timed automata provide the proper 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 constrain the timing of events. However, 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 are observable. For many realistic applications, however, this reduced expressivity causes no problems. For instance, Kurose and Ross [17] use finite state machine models with timers to explain transport layer protocols. Caldwell et al. [18] 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 every transition resets. 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 sufficiently expressive models to describe the real-time behavior of many realistic applications that we can learn 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 we can easily obtain learning algorithms for MM1Ts via a reduction to the problem of learning Mealy machines and then apply established algorithms for learning Mealy machines like L^*_M [19] and TTT [20].

We describe an implementation of an MM1T learner on top of LearnLib, a state-of-the-art tool for learning Mealy machines [21], and compare its performance with the tools of Aichernig et al. [2] and An et al. [3] on several benchmarks: TCP connection setup, Android's Authentication & Key Management (AKM) service, and some industrial benchmarks taken from [2]. The tool of [3] can only learn the benchmarks with a "helpful" teacher that provides information about resets; without help, it is unable to learn the benchmarks. The tool of [2] is more generally applicable because it can learn timed automata with multiple clock variables. However, our implementation outperforms it with several orders of magnitude in terms of the total number of input symbols required to learn the benchmark models with a single clock. We only compare against that tool in a restricted setup where we learn Deterministic One-clock Timed Automata (DOTAs), a model that is closest to MM1Ts in terms of expressiveness.

2. Mealy machines with a single timer

This section introduces Mealy machines with a single timer (MM1Ts) and describes their semantics. 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 $f : X \rightharpoonup Y$ with the set of pairs $\{(x, y) \in X \times Y \mid f(x) = y\}$. If f and g are partial functions, then $f(x) = g(y)$ holds if either both $f(x)$ and $g(y)$ are undefined, or both $f(x)$ and $g(y)$ are defined and have the same result.

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 a timeout as an input event. This choice makes sense because the timeout event can be triggered by the environment by not providing a regular input to the system (or prevented by providing an input immediately).

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,

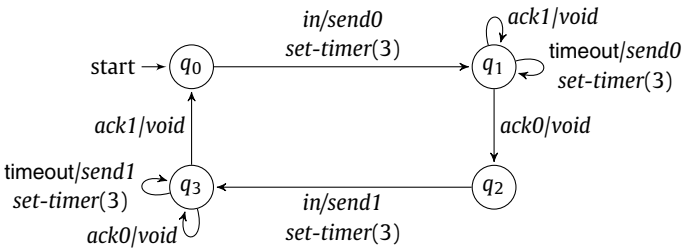

Fig. 1. MM1T model of alternating-bit protocol sender.

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

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

(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

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

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

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

(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 [17, 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 $Q_{off} = \{q_0, q_2\}$. 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 acknowledgment with the correct sequence number 0 (input "*ack0*") it stops the timer and jumps to state q_2 without generating visible output ("*void*"). Acknowledgment with the incorrect sequence number (input "*ack1*") is ignored. Likewise, in state q_1, input "*in*", and in state q_0 inputs "*ack0*" and "*ack1*" are ignored (for readability, we hid these self-loop transitions in the diagram). If no "*ack0*" input arrives within 3 time-units, 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.

Remark The output "*void*" in a transition corresponds to the *absence* of an observable output. When the environment offers an input to an MM1T and this does not trigger a visible output, then this is interpreted as a *void* output. Similarly, when the environment observes an output in a situation where it did not offer any input to the MM1T, it may conclude that a timeout has occurred. Thus the input "timeout" corresponds to the *absence* of an observable input in situations where an output is observed. For each transition with input i and output o we require that either $i \neq \mathsf{timeout}$ or $o \neq void$. This ensures that all i/o interactions of an MM1T can be observed.

We present two formal semantics for MM1Ts, an untimed and a timed one. An untimed word just collects the transition labels on finite paths in an MM1T, whereas a timed word also describes how much time may elapse in between the transitions of the automaton. Whereas the untimed semantics is close to the syntax of an MM1T model, the timed semantics describes the real-time observations that may be recorded during interaction with a physical system. For the MM1T of Fig. 1, an example of an untimed word is

$$(in, send0, 3)(ack1, void, \perp)(ack0, void, \perp)(in, send1, 3)(\mathsf{timeout}, send1, 3)$$

whereas an example of a timed word is

$$(59, in, send0)(1.1, ack1, void)(1.7, ack0, void)(2.2, in, send1)(3, \mathsf{timeout}, send1)$$

Here the real-number at the start of each triple denotes the time that elapses before the subsequent occurrence of the input and the output. Below we formally define the untimed and timed semantics.

Untimed semantics In the untimed semantics, we only 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

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

that begins with the initial state q_0 of $\mathcal{M}$ such that for each $j \le k$, $q_j \xrightarrow{i_j/o_j,n_j} q_{j+1}$ is a transition of $\mathcal{M}$. 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.

It will be useful to characterize untimed equivalence in terms of *bisimulations*. The fact that MM1Ts are deterministic allows us to simplify the second transfer condition (8).

Definition 2. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two MM1Ts with the same inputs, where $\mathcal{M}_j = (I, O_j, Q_j, q_0^j, \delta_j, \lambda_j, \tau_j)$, for $j = 1, 2$. A *bisimulation* between $\mathcal{M}_1$ and $\mathcal{M}_2$ is a relation $R \subseteq Q_1 \times Q_2$ satisfying, for every $q_1 \in Q_1$, $q_2 \in Q_2$ and $i \in I$,

$$q_0^1 \; R \; q_0^2 \tag{6}$$

$$q_1 \; R \; q_2 \wedge \delta_1(q_1, i) \downarrow \;\Rightarrow\; \delta_2(q_2, i) \downarrow \wedge \, \delta_1(q_1, i) \; R \; \delta_2(q_2, i) \wedge \lambda_1(q_1, i) = \lambda_2(q_2, i) \wedge \tau_1(q_1, i) = \tau_2(q_2, i) \tag{7}$$

$$q_1 \; R \; q_2 \wedge \delta_2(q_2, i) \downarrow \;\Rightarrow\; \delta_1(q_1, i) \downarrow \tag{8}$$

We write $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff there is a bisimulation R between $\mathcal{M}_1$ and $\mathcal{M}_2$.

The next lemma, which is easy to prove, is a variation of the classical result that trace equivalence and bisimulation coincide for deterministic systems [22].

Lemma 1. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be MM1Ts with the same inputs. Then $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff $\mathcal{M}_1 \approx_{untimed} \mathcal{M}_2$.*

Timed semantics The timed semantics, which is slightly more involved, describes the real-time behavior of an MM1T. It associates an infinite state transition system to an MM1T that describes all possible configurations and transitions between them. A *configuration* of an MM1T is a pair (q, t), where $q \in Q$ is a state and $t \in \mathbb{R}^{\ge 0} \cup \{\infty\}$ specifies the value of the timer. We require $t = \infty$ iff $q \in Q_{off}$. 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 $q \in Q$, $r \in Q_{off}$, $s, s' \in Q_{on}$, $i \in I$, $o \in O$, $t \in \mathbb{R}^{\ge 0} \cup \{\infty\}$, $d \in \mathbb{R}^{\ge 0}$ and $n \in \mathbb{N}^{>0}$,

$$\frac{d \le t}{(q, t) \xrightarrow{d} (q, t - d)} \tag{9}$$

$$\frac{q \xrightarrow{i/o,n} s, \quad i = \mathsf{timeout} \Rightarrow t = 0}{(q, t) \xrightarrow{i/o} (s, n)} \tag{10}$$

$$\frac{q \xrightarrow{i/o} r, \quad i = \mathsf{timeout} \Rightarrow t = 0}{(q, t) \xrightarrow{i/o} (r, \infty)} \tag{11}$$

$$\frac{s \xrightarrow{i/o} s', \quad i \neq \mathsf{timeout}}{(s, t) \xrightarrow{i/o} (s', t)} \tag{12}$$

Rule (9) 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 $d \in \mathbb{R}^{\ge 0}$. So when the timer is off, time may advance indefinitely. Rule (10) describes

events where the timer is (re)set; a timeout may occur only when the timer expires in the source state. Rule (11) describes events where the timer is off in the target state; again, a timeout may occur only when the timer expires in the source state. Finally, rule (12) 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, for each index j, $t_j \in \mathbb{R}^{\geq 0}$, $i_j \in I$, and $o_j \in O$. 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 applied 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

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

that begins with the initial configuration C_0 of $\mathcal{M}$ and where, for each $j \leq k$, $C_j \xrightarrow{t_j} C_j'$ and $C_j' \xrightarrow{i_j/o_j} C_{j+1}$ are transitions of $\mathcal{M}$. Since MM1Ts are deterministic, for each timed word w there is at most one timed run of $\mathcal{M}$ over w. We say w is a timed word of $\mathcal{M}$ iff there is a timed 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, the timed and untimed equivalence actually coincide. Below we show that untimed equivalence implies timed equivalence, a result that we need to prove the correctness of our learning algorithm. (The converse implication also holds, but we will not discuss this here, as it would distract from the main line of this article.)

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

Proof. Assume $\mathcal{M} \approx_{untimed} \mathcal{N}$ and assume $w = (i_0, o_0, t_0), (i_1, o_1, t_1) \cdots (i_k, o_k, t_k)$ is a timed word of $\mathcal{M}$. Since $\approx_{untimed}$ is symmetric, it suffices to prove that w is a timed word of $\mathcal{N}$.

Because w is a timed word of $\mathcal{M}$, $\mathcal{M}$ has a timed run over w:

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

Let $C_j = (q_j, u_j)$ and $C_j' = (q_j, u_j')$, for all j. Now observe that for each transition in this timed run, there is a unique rule (either (9), (10), (11) or (12)) that has been applied, with a unique untimed transition of $\mathcal{M}$ in the antecedent. Thus $\mathcal{M}$ has a corresponding untimed run

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

and thus $w' = (i_0, o_0, n_0), (i_1, o_1, n_1) \cdots (i_k, o_k, n_k)$ is an untimed word of $\mathcal{M}$. Since $\mathcal{M} \approx_{untimed} \mathcal{N}$, w' is also an untimed word of $\mathcal{N}$. Therefore, $\mathcal{N}$ has an untimed run over w':

$$r_0 \xrightarrow{i_0/o_0,n_0} r_1 \xrightarrow{i_1/o_1,n_1} \cdots \xrightarrow{i_k/o_k,n_k} r_{k+1},$$

with all r_j states of $\mathcal{N}$ and r_0 the initial state of $\mathcal{N}$. Let $D_j = (r_j, u_i)$ and $D_j' = (r_j, u_j')$, for all j. We claim that

$$D_0 \xrightarrow{t_0} D_0' \xrightarrow{i_0/o_0} D_1 \xrightarrow{t_1} D_1' \xrightarrow{i_1/o_1} D_2 \cdots \xrightarrow{t_k} D_k' \xrightarrow{i_k/o_k} D_{k+1}$$

is a timed run over w of $\mathcal{N}$. In order to prove this claim, we show by induction on j that the part of the sequence up to D_j is a timed run of $\mathcal{N}$, and moreover the timer is on in state q_j iff it is on in state r_j:

- Basis. Since q_0 is the initial state of $\mathcal{M}$ and r_0 is the initial state of $\mathcal{N}$, the timer is off in both states. Moreover, $u_0 = \infty$ and so D_0 is the initial configuration and a timed run of $\mathcal{N}$.
- Induction step. Suppose the statement holds for index $j \leq k$. By rule (9), $D_j \xrightarrow{t_j} D_j'$ is a timed step of $\mathcal{N}$. If $n_j \in \mathbb{N}^{>0}$, then by rule (3), the timer is on both in q_{j+1} and r_{j+1}, and by rule (10), $D_j' \xrightarrow{i_j/o_j} D_{j+1}$ is a timed step of $\mathcal{N}$. Otherwise, if $n_j = \perp$ and the timer is off in state q_j, then by induction hypothesis it is also off in state r_j. Hence, by rule (4), the timer is also off in states q_{j+1} and r_{j+1}. Therefore $u_j = u_{j+1} = \infty$, and $D_j' \xrightarrow{i_j/o_j} D_{j+1}$ is a timed step of $\mathcal{N}$. Finally, we consider the case where $n_j = \perp$ and the timer is on in state q_j. Then by induction hypothesis the timer is also on in state r_j. If $i_j = \mathsf{timeout}$ then by rule (5) the timer is off in states q_{j+1} and r_{j+1}. This means that $u_{j+1} = \infty$. Moreover, by rule (11), $u_j = 0$ and thus $D_j' \xrightarrow{i_j/o_j} D_{j+1}$ is a timed step of $\mathcal{N}$. Otherwise, if $i_j \neq \mathsf{timeout}$ then we claim that the timer is on in state q_{j+1} iff it is on in state r_{j+1}. Because suppose the timer is on in q_{j+1} and off in r_{j+1}. Then $\mathcal{M}$

has an untimed word $(i_0, o_0, n_0), (i_1, o_1, n_1) \cdots (i_j, o_j, n_j)(\mathsf{timeout}, o, n)$, for certain o and n. But $\mathcal{N}$ cannot possibly have such an untimed word, since the timer is off in r_{j+1}. This contradicts the assumption $\mathcal{M} \approx_{untimed} \mathcal{N}$. Via a symmetric argument we can derive a contradiction when we assume that the timer is off in q_{j+1} and on in r_{j+1}. If the timer is off in both q_{j+1} and r_{j+1}, then $D'_j \xrightarrow{i_j/o_j} D_{j+1}$ is a timed step of $\mathcal{N}$ by application of rule (11). If the timer is on in both q_{j+1} and r_{j+1}, then $D'_j \xrightarrow{i_j/o_j} D_{j+1}$ is a timed step of $\mathcal{N}$ by application of rule (12).

Thus w is a timed word of $\mathcal{N}$, as required. □

3. Learning MM1Ts

3.1. A MAT framework for MM1Ts

Many active learning algorithms have been designed following Angluin's approach of a *minimally adequate teacher (MAT)* [23]. In this approach, learning is viewed as a game in which a learner has to infer the behavior of an unknown state diagram by asking queries to a teacher. We present a natural timed adaptation of the MAT framework for learning MM1Ts, based on the timed semantics. In a *timed membership query (TMQ)*, the learner supplies a sequence of inputs with precise timing. In response, the teacher returns a timed word that also contains the timeouts and outputs that occur in response to these inputs, as well as their precise timing. Via a *timed equivalence query (TEQ)*, the learner asks whether a hypothesis MM1T that it has constructed accepts exactly the same timed words as the (unknown) MM1T of the teacher. If this is the case, the teacher's answer is 'yes'; otherwise, it is 'no' coupled with a timed word showing that the hypothesis is incorrect. The learner's task is to infer the unknown MM1T of the teacher via a finite number of timed membership and timed equivalence queries.

Formally, a timed membership query consists of a *timed input word*, which is an alternating sequence of delays in $\mathbb{R}^{\geq 0}$ and inputs from $I \setminus \{\mathsf{timeout}\}$, beginning and ending with a delay, that is, an element from $\mathbb{R}^{\geq 0}\,((I \setminus \{\mathsf{timeout}\})\,\mathbb{R}^{\geq 0})^*$.

The operation $\bullet$ concatenates two timed input words by putting them in sequence but adding the final delay of the first sequence and the initial delay of the second sequence:

$$(ud) \bullet (d'u') = u\,(d + d')\,u'.$$

We may associate a timed input word $tiw(w)$ to any timed word w by removing the outputs and occurrences of timeout, replacing consecutive numbers by their sum, and possibly placing 0 at the end of the sequence:

$$tiw(\epsilon) = 0$$
$$tiw((t, i, o)\ w) = \begin{cases} t \bullet tiw(w) & \text{if } i = \mathsf{timeout} \\ (t\ i\ 0) \bullet tiw(w) & \text{otherwise} \end{cases}$$

Thus, for instance,

$$tiw((7, i, o), (1, i, o), (1, \mathsf{timeout}, o')) = 7\ i\ 1\ i\ 1,$$
$$tiw((3, i_1, o_1), (1.1, i_2, o_2), (2, \mathsf{timeout}, o_3), (2.1, i_4, o_4)) = 3\ i_1\ 1.1\ i_2\ 4.1\ i_4\ 0.$$

If u and u' are timed input words, then we write $u \propto u'$ if u and u' are equal, except that the final delay of u is less than or equal to the final delay of u'. Let u be any timed input word over I.

A timed word w of $\mathcal{M}$ is an *outcome of running experiment* u on $\mathcal{M}$ if $tiw(w) \propto u$ and w is maximal in the sense that there is no timed word w' with w a proper prefix of w' and $tiw(w') \propto u$. For instance, if we run experiment 1 *in* 7 on the MMT of Fig. 1, then $(1, in, send0), (3, \mathsf{timeout}, send0), (3, \mathsf{timeout}, send0)$ is an outcome, but its prefix $(1, in, send0), (3, \mathsf{timeout}, send0)$ is not. Thus, if a timeout is still possible before the final delay of u expires, it must be included in an outcome.

Note that when we run an experiment u on $\mathcal{M}$ there will always be at least one outcome: inputs from $I \setminus \{\mathsf{timeout}\}$ are enabled in every state of an MM1T, we can always wait until the occurrence of the next timeout, and only finitely many timeouts can occur in any given interval of time. However, due to race conditions between inputs and timeouts, the outcome of an experiment is not always uniquely determined. For instance, the experiment 1 *in* 3 *ack0* 0 has two timed words as possible outcomes: $(0, in, send0), (3, ack0, void)$ and $(0, in, send0), (3, \mathsf{timeout}, send0), (0, ack0, void)$. The *timed membership query* function $tmq_{\mathcal{M}}$ assigns to each timed input word the set of all outcomes of running experiment u on $\mathcal{M}$. It is easy to see that two MM1Ts $\mathcal{M}$ and $\mathcal{N}$ are timed equivalent iff they behave the same in all experiments, that is, for all timed input words u, $tmq_{\mathcal{M}}(u) = tmq_{\mathcal{N}}(u)$.

3.2. From MM1Ts to Mealy machines and back

Below we explore the connection between MM1Ts and the classical notion of *Mealy machines* because this will allow us to reuse existing learning algorithms for Mealy machines [24,25] for learning MM1Ts. In this way, we may obtain a learning algorithm for MM1Ts almost for free. Essentially, a Mealy machine is just an MM1T in which the timer is off in all states. Thus, MM1Ts generalize Mealy machines. Conversely, we can view each MM1T as a Mealy machine of a special form.

Definition 3. A *Mealy machine (MM)* 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: $\delta(q, \epsilon) = q$ and $\delta(q, \sigma i) = \delta(\delta(q, \sigma), i)$, for $\sigma \in I^*$ and $i \in I$. The *membership query* function $mq_{\mathcal{M}} : I^+ \to O$ assigns to each nonempty sequence of inputs the final output when we apply these inputs from the initial state of $\mathcal{M}$: $mq_{\mathcal{M}}(\sigma i) = \lambda(\delta(q_0, \sigma), i)$. 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 $\sigma \in I^+$, $mq_{\mathcal{M}}(\sigma) = mq_{\mathcal{N}}(\sigma)$.

Note that the transition function of a Mealy machine is total, unlike the transition function of an MM1T, which is undefined for the timeout input in states where the timer is off. We say that Mealy machines are *input complete*. It will be useful to characterize equivalence of Mealy machines in terms of *bisimulations*. Since Mealy machines are input complete and deterministic, the definition is simpler than the usual definition of bisimulation on labeled transition systems.

Definition 4. Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two Mealy machines with the same inputs, where $\mathcal{M}_j = (I, O_j, Q_j, q_0^j, \delta_j, \lambda_j)$, for $j = 1, 2$. A *bisimulation* between $\mathcal{M}_1$ and $\mathcal{M}_2$ is a relation $R \subseteq Q_1 \times Q_2$ satisfying, for all $q_1 \in Q_1$, $q_2 \in Q_2$ and $i \in I$,

$$q_0^1 \; R \; q_0^2$$
$$q_1 \; R \; q_2 \;\Rightarrow\; \lambda_1(q_1, i) = \lambda_2(q_2, i) \wedge \delta_1(q_1, i) \; R \; \delta_2(q_2, i)$$

We write $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff there is a bisimulation relation between $\mathcal{M}_1$ and $\mathcal{M}_2$.

The next lemma, which like Lemma 1 is a variation of the classical result of [22], is again easy to prove.

Lemma 2. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Mealy machines with the same inputs. Then $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff $\mathcal{M}_1 \approx \mathcal{M}_2$.*

We associate a Mealy machine $\mathsf{Mealy}(\mathcal{M})$ to each MM1T $\mathcal{M}$ as follows. We keep the same states, initial state, inputs and transitions, but add timeout self-loops for each state in Q_{off} to make the Mealy machine input enabled. We associate a special output nil to each new timeout self-loop. The outputs of the other transitions of $\mathsf{Mealy}(\mathcal{M})$ are pairs consisting of the output from $\mathcal{M}$ together with the timer update. An inverse operation assigns a tuple $\mathsf{MM1T}(\mathcal{N})$ to each Mealy machine $\mathcal{N}$ with the right input and output alphabets. We will see that under certain assumptions $\mathsf{MM1T}(\mathcal{N})$ is indeed an MM1T.

Definition 5. Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda, \tau)$ be an MM1T. Then $\mathsf{Mealy}(\mathcal{M})$ is the Mealy machine $(I, O', Q, q_0, \delta', \lambda')$, where

$$O' = (O \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}$$
$$\delta'(q, i) = \begin{cases} \delta(q, i) & \text{if } \delta(q, i) \downarrow \\ q & \text{otherwise} \end{cases}$$
$$\lambda'(q, i) = \begin{cases} (\lambda(q, i), \tau(q, i)) & \text{if } \tau(q, i) \downarrow \\ (\lambda(q, i), \bot) & \text{if } \lambda(q, i) \downarrow \text{ and } \tau(q, i) \uparrow \\ \mathsf{nil} & \text{otherwise} \end{cases}$$

Conversely, suppose $\mathcal{N} = (I, O, Q, q_0, \delta, \lambda)$ is a Mealy machine with $\mathsf{timeout} \in I$ and $O \subseteq (\Omega \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}$, for some set Ω. Then we may reverse the above construction and define $\mathsf{MM1T}(\mathcal{N}) = (I, O', Q_{off} \cup Q_{on}, q_0, \delta', \lambda', \tau)$, where

$$O' = \{o \in \Omega \mid \exists n \in \mathbb{N} \cup \{\bot\} : (o, n) \in O\}$$
$$Q_{off} = \{q \in Q \mid \lambda(q, \mathsf{timeout}) = \mathsf{nil}\}$$
$$Q_{on} = Q \setminus Q_{off}$$
$$\delta'(q, i) = \begin{cases} \delta(q, i) & \text{if } q \in Q_{on} \text{ or } i \neq \mathsf{timeout} \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\lambda'(q, i) = \begin{cases} \pi_1(\lambda(q, i)) & \text{if } \lambda(q, i) \neq \mathsf{nil} \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\tau(q, i) = \begin{cases} \pi_2(\lambda(q, i)) & \text{if } \lambda(q, i) \in O' \times \mathbb{N}^{>0} \\ \text{undefined} & \text{otherwise} \end{cases}$$

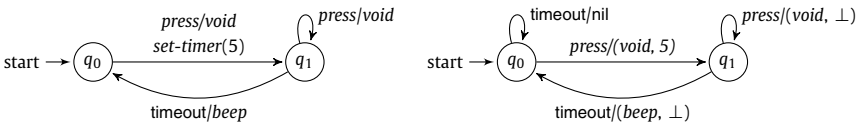

Fig. 2. MM1T $\mathcal{M}$ (left) and corresponding Mealy machine $\mathcal{N}$ (right).

where π_1 and π_2 project a pair to its first and second element, respectively.

Example 2. The translations between MM1Ts and Mealy machines are illustrated in Fig. 2 with an MM1T $\mathcal{M}$ and a Mealy machine $\mathcal{N}$ that translate to each other: $\mathsf{Mealy}(\mathcal{M}) = \mathcal{N}$ and $\mathsf{MM1T}(\mathcal{N}) = \mathcal{M}$. The MM1T $\mathcal{M}$ models a device that says "beep" exactly 5 time units after a button is pressed.

The following result, which follows from Definition 5, Theorem 1, Lemma 2 and Lemma 1, asserts that Mealy and MM1T act (in one direction) like adjoint operators.

Theorem 2. *Suppose $\mathcal{M}$ is an MM1T with inputs I and outputs $O \subseteq \Omega$, and $\mathcal{N}$ is a Mealy machine with all states reachable, inputs I and outputs $\hat{O} \subseteq (\Omega \times (\mathbb{N}^{>0} \cup \{\bot\})) \cup \{\mathsf{nil}\}$. Suppose $\mathsf{Mealy}(\mathcal{M}) \approx \mathcal{N}$. Then $\mathsf{MM1T}(\mathcal{N})$ is an MM1T and $\mathcal{M} \approx_{timed} \mathsf{MM1T}(\mathcal{N})$.*

Proof. Let

$$\begin{aligned} \mathcal{M} &= (I, O, Q, q_0, \delta, \lambda, \tau), \text{ where } Q = Q_{off} \cup Q_{on} \\ \mathsf{Mealy}(\mathcal{M}) &= \mathcal{N}' = (I, O', Q, q_0, \delta', \lambda') \\ \mathcal{N} &= (I, \hat{O}, \hat{Q}, \hat{q}_0, \hat{\delta}, \hat{\lambda}) \\ \mathsf{MM1T}(\mathcal{N}) &= \mathcal{M}' = (I, \hat{O}', \hat{Q}, \hat{q}_0, \hat{\delta}', \hat{\lambda}', \hat{\tau}), \text{ where } \hat{Q} = \hat{Q}_{off} \cup \hat{Q}_{on} \end{aligned}$$

(By Definition 5, $\mathcal{M}$ and $\mathcal{N}'$ have the same inputs, as well as $\mathcal{N}$ and $\mathcal{M}'$.)

Let R be a bisimulation relation between $\mathcal{N}'$ and $\mathcal{N}$ (R exists by Lemma 2). We first check that $\mathcal{M}' = \mathsf{MM1T}(\mathcal{N})$ is an MM1T:

1. Since $\mathcal{M}$ is an MM1T, I is finite and $\mathsf{timeout} \in I$.
2. By construction of $\mathsf{MM1T}(\mathcal{N})$, $\hat{Q}_{off}$ and $\hat{Q}_{on}$ are disjoint.
3. Suppose $(q, \hat{q}) \in R$. We show that $q \in Q_{off}$ iff $\hat{q} \in \hat{Q}_{off}$:

$$\begin{aligned} q \in Q_{off} &\Leftrightarrow \delta(q, \mathsf{timeout}) \uparrow && \text{(since } \mathcal{M} \text{ satisfies rule (1))} \\ &\Leftrightarrow \lambda(q, \mathsf{timeout}) \uparrow \wedge\, \tau(q, \mathsf{timeout}) \uparrow && \text{(since } \mathcal{M} \text{ satisfies rules (2) and (3))} \\ &\Leftrightarrow \lambda'(q, \mathsf{timeout}) = \mathsf{nil} && \text{(by definition Mealy)} \\ &\Leftrightarrow \hat{\lambda}(\hat{q}, \mathsf{timeout}) = \mathsf{nil} && \text{(since } R \text{ is a bisimulation)} \\ &\Leftrightarrow \hat{q} \in \hat{Q}_{off} && \text{(by definition MM1T)} \end{aligned}$$

4. Since $\mathcal{M}$ is an MM1T, $q_0 \in Q_{off}$. Since R is a bisimulation, $q_0\, R\, \hat{q}_0$. By the previous item, this implies $\hat{q}_0 \in \hat{Q}_{off}$.
5. $\mathcal{M}'$ satisfies rule (1) by definition of $\hat{\delta}'$.
6. We show that $\mathcal{M}'$ satisfies rule (2). Let $\hat{q} \in \hat{Q}$. Since all states of $\mathcal{N}$ are reachable and R is a bisimulation between $\mathcal{N}'$ and $\mathcal{N}$, there is a state $q \in Q$ with $q\, R\, \hat{q}$. Now we derive:

$$\begin{aligned} \hat{\lambda}'(\hat{q}, i) \downarrow &\Leftrightarrow \hat{\lambda}(\hat{q}, i) \neq \mathsf{nil} && \text{(by definition MM1T)} \\ &\Leftrightarrow \lambda'(q, i) \neq \mathsf{nil} && \text{(since } q\, R\, \hat{q}) \\ &\Leftrightarrow \lambda(q, i) \downarrow && \text{(by definition Mealy)} \\ &\Leftrightarrow \delta(q, i) \downarrow && (\mathcal{M} \text{ satisfies rule (2))} \\ &\Leftrightarrow q \in Q_{on} \vee i \neq \mathsf{timeout} && (\mathcal{M} \text{ satisfies rule (1))} \\ &\Leftrightarrow \hat{q} \in \hat{Q}_{on} \vee i \neq \mathsf{timeout} && \text{(item 3 above)} \\ &\Leftrightarrow \hat{\delta}'(\hat{q}, i) \downarrow && (\mathcal{M}' \text{ satisfies rule (1))} \end{aligned}$$

7. We show that $\mathcal{M}'$ satisfies rule (3). Let $\hat{q} \in \hat{Q}$ and $q \in Q$ with $q \; R \; \hat{q}$.

$$\begin{aligned}
\hat{\tau}(\hat{q}, i) \downarrow &\Rightarrow \hat{\lambda}(\hat{q}, i) \in \hat{O}' \times \mathbb{N}^{>0} && \text{(by definition MM1T)} \\
&\Rightarrow \lambda'(q, i) \in \hat{O}' \times \mathbb{N}^{>0} && \text{(since } q \; R \; \hat{q}\text{)} \\
&\Rightarrow \tau(q, i) \downarrow && \text{(by definition Mealy)} \\
&\Rightarrow \delta(q, i) \in Q_{on} && \text{(as } \mathcal{M} \text{ satisfies rule (3))} \\
&\Rightarrow \delta'(q, i) \in Q_{on} && \text{(by definition Mealy)} \\
&\Rightarrow \hat{\delta}(\hat{q}, i) \in \hat{Q}_{on} && \text{(by } q \; R \; \hat{q} \text{ and item 3 above)} \\
&\Rightarrow \hat{\delta}'(\hat{q}, i) \in \hat{Q}_{on} && \text{(by definition MM1T)}
\end{aligned}$$

8. We show that $\mathcal{M}'$ satisfies rule (4). Let $\hat{q} \in \hat{Q}$ and $q \in Q$ with $q \; R \; \hat{q}$.

$$\begin{aligned}
&\hat{q} \in \hat{Q}_{off} \wedge \hat{\delta}'(\hat{q}, i) \in \hat{Q}_{on} && \text{(by definition MM1T)} \\
\Rightarrow \; &\hat{q} \in \hat{Q}_{off} \wedge \hat{\delta}(\hat{q}, i) \in \hat{Q}_{on} && \text{(by } q \; R \; \hat{q} \text{ and item 3 above)} \\
\Rightarrow \; &q \in Q_{off} \wedge \delta'(q, i) \in Q_{on} && \text{(by definition Mealy)} \\
\Rightarrow \; &q \in Q_{off} \wedge \delta(q, i) \in Q_{on} && \text{(as } \mathcal{M} \text{ satisfies rule (4))} \\
\Rightarrow \; &\tau(q, i) \downarrow && \text{(by definition Mealy)} \\
\Rightarrow \; &\lambda'(q, i) \in O \times \mathbb{N}^{>0} && \text{(by } q \; R \; \hat{q}\text{)} \\
\Rightarrow \; &\hat{\lambda}(\hat{q}, i) \in \hat{O}' \times \mathbb{N}^{>0} && \text{(by definition MM1T)} \\
\Rightarrow \; &\hat{\tau}(\hat{q}, i) \downarrow
\end{aligned}$$

9. We show that $\mathcal{M}'$ satisfies rule (5). Let $\hat{q} \in \hat{Q}$ and $q \in Q$ with $q \; R \; \hat{q}$.

$$\begin{aligned}
&\hat{\delta}'(\hat{q}, \mathsf{timeout}) \in \hat{Q}_{on} && \text{(by definition MM1T)} \\
\Rightarrow \; &\hat{\delta}(\hat{q}, \mathsf{timeout}) \in \hat{Q}_{on} && \text{(by } q \; R \; \hat{q} \text{ and item 3 above)} \\
\Rightarrow \; &\delta'(q, \mathsf{timeout}) \in Q_{on} && \text{(by definition Mealy and } \mathcal{M} \text{ satisfies rule (1))} \\
\Rightarrow \; &\delta(q, \mathsf{timeout}) \in Q_{on} && \text{(as } \mathcal{M} \text{ satisfies rule (5))} \\
\Rightarrow \; &\tau(q, \mathsf{timeout}) \downarrow && \text{(by definition Mealy)} \\
\Rightarrow \; &\lambda'(q, \mathsf{timeout}) \in O \times \mathbb{N}^{>0} && \text{(by } q \; R \; \hat{q}\text{)} \\
\Rightarrow \; &\hat{\lambda}(\hat{q}, \mathsf{timeout}) \in \hat{O}' \times \mathbb{N}^{>0} && \text{(by definition MM1T)} \\
\Rightarrow \; &\hat{\tau}(\hat{q}, \mathsf{timeout}) \downarrow
\end{aligned}$$

Next we prove that R is also a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$.

1. Since R is a bisimulation between $\mathcal{N}'$ and $\mathcal{N}$, $q_0 \; R \; \hat{q}_0$.
2. Suppose $q \; R \; \hat{q}$. Then

$$\begin{aligned}
\delta(q, i) \downarrow &\Leftrightarrow q \in Q_{on} \vee i \neq \mathsf{timeout} && (\mathcal{M} \text{ satisfies rule (1)}) \\
&\Leftrightarrow \hat{q} \in \hat{Q}_{on} \vee i \neq \mathsf{timeout} && \text{(by } q \; R \; \hat{q} \text{ and item 3 above)} \\
&\Leftrightarrow \hat{\delta}'(\hat{q}, i) \downarrow && (\mathcal{M}' \text{ satisfies rule (1)})
\end{aligned}$$

3. Suppose $q \; R \; \hat{q} \wedge \delta(q, i) \downarrow$. Then, by the previous item, $\hat{\delta}'(\hat{q}, i) \downarrow$.
   - Since R is a bisimulation between $\mathcal{N}'$ and $\mathcal{N}$, $\delta'(q, i) \; R \; \hat{\delta}(\hat{q}, i)$. By the definition of Mealy, $\delta'(q, i) = \delta(q, i)$. By the definition of MM1T and because $\mathcal{M}'$ satisfies rule (1), $\hat{\delta}'(\hat{q}, i) = \hat{\delta}(\hat{q}, i)$. Thus, $\delta(q, i) \; R \; \hat{\delta}'(\hat{q}, i)$, as required.
   - Since $\mathcal{M}$ and $\mathcal{M}'$ satisfy rule (2), $\lambda(q, i) \downarrow$ and $\hat{\lambda}'(\hat{q}, i) \downarrow$. Then

$$\begin{aligned}
\lambda(q, i) &= \pi_1(\lambda'(q, i)) && \text{(by definition Mealy)} \\
&= \pi_1(\hat{\lambda}(\hat{q}, i)) && (R \text{ is a bisimulation between } \mathcal{N}' \text{ and } \mathcal{N}) \\
&= \hat{\lambda}'(\hat{q}, i) && \text{(by definition MM1T)}
\end{aligned}$$

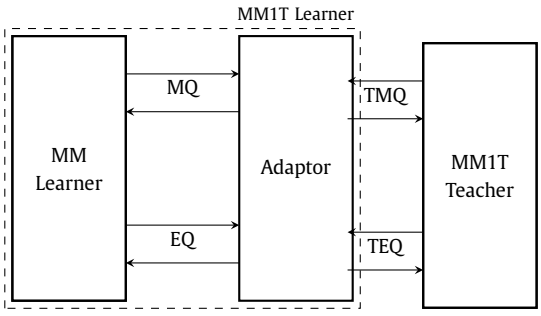

Fig. 3. Using a Mealy machine learner to construct an MM1T learner.

- In order to prove $\tau(q,i) = \hat{\tau}(\hat{q},i)$, we distinguish two cases.
  - If $\tau(q,i)\downarrow$ then

$$\begin{aligned}\tau(q,i) &= \pi_2(\lambda'(q,i)) && \text{(by definition Mealy)}\\ &= \pi_2(\hat{\lambda}(\hat{q},i)) && (R \text{ is a bisimulation between } \mathcal{N}' \text{ and } \mathcal{N})\\ &= \hat{\tau}(\hat{q},i) && \text{(by definition MM1T)}\end{aligned}$$

  - If $\tau(q,i)\uparrow$ then, by definition of Mealy, $\lambda'(q,i) \in O \times \{\bot\}$. Therefore, since R is a bisimulation between $\mathcal{N}'$ and $\mathcal{N}$, $\hat{\lambda}(\hat{q},i) \in O \times \{\bot\}$. Now the definition of MM1T implies $\hat{\tau}(\hat{q},i)\uparrow$.

Since R is a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$, $\mathcal{M} \simeq \mathcal{M}'$. Therefore, by Lemma 1, $\mathcal{M} \approx_{untimed} \mathcal{M}'$, and by Theorem 1, $\mathcal{M} \approx_{timed} \mathcal{M}'$, as required. □

3.3. Using a Mealy machine learner to build an MM1T learner

Theorem 2 allows us to reduce the problem of learning MM1Ts to the classical problem of learning Mealy machines. This simple reduction makes it possible to apply established and highly efficient/optimized algorithms for learning Mealy machines like L^*_M [19], TTT [20] and $L^\#$ [26].

In the MAT framework for Mealy machines, the task of the learner is to learn an unknown Mealy machine $\mathcal{M}$ through two types of queries: With a *membership query (MQ)*, the learner asks what the output is in response to an input sequence $\sigma \in I^*$. The teacher answers with output $mq_{\mathcal{M}}(\sigma)$. With an *equivalence query (EQ)*, the learner asks if a hypothesized Mealy machine $\mathcal{N}$ is correct, that is, whether $\mathcal{M} \approx \mathcal{N}$. The teacher answers 'yes' if this is true. Otherwise, she answers 'no' and supplies a counterexample $\sigma \in I^*$ that distinguishes $\mathcal{M}$ and $\mathcal{N}$.

To achieve the reduction, we place an *adaptor* between a Mealy machine learner and a teacher for an MM1T $\mathcal{M}$, as illustrated in Fig. 3. From the perspective of the Mealy machine learner, the adaptor behaves like a teacher for Mealy($\mathcal{M}$) that answers membership (MQ) and equivalence queries (EQ). In order to answer these queries, the adaptor poses timed membership queries (TMQ) and timed equivalence queries (TEQ) to the MM1T teacher for $\mathcal{M}$. When the learner has succeeded to learn a minimal Mealy machine $\mathcal{N}$ that is equivalent to Mealy($\mathcal{M}$), we know by Theorem 2 that $\mathcal{M} \approx_{timed}$ MM1T($\mathcal{N}$), and so the learner has learned an MM1T that is timed equivalent to $\mathcal{M}$. Effectively, the adaptor and the Mealy machine learner together act as an MM1T learner.

Thus, the *only* thing that we need to implement in order to obtain an active learning algorithm for MM1Ts is an adaptor that answers membership and equivalence queries for Mealy($\mathcal{M}$) by posing timed membership and timed equivalence queries to a teacher for $\mathcal{M}$. Actually, since our implementation uses the LearnLib tool as Mealy machine learner, and LearnLib uses membership queries to approximate equivalence queries, Theorem 2 implies that there is no need to implement equivalence queries in our setting. Below we describe how the adaptor answers membership queries for Mealy($\mathcal{M}$) by posing timed membership queries to $\mathcal{M}$.

3.4. Implementing membership queries

As its main data structure, the adaptor maintains an observation tree, which describes the part of the MM1T that has been explored thus far.

Definition 6. An *observation tree* $\mathcal{T}$ is an MM1T except that:

1. Inputs need not be enabled, that is, instead of rule (1) we only require the weaker rule $q \in Q^{\mathcal{T}}_{off} \wedge i = \text{timeout} \Rightarrow \delta^{\mathcal{T}}(q,i)\uparrow$.

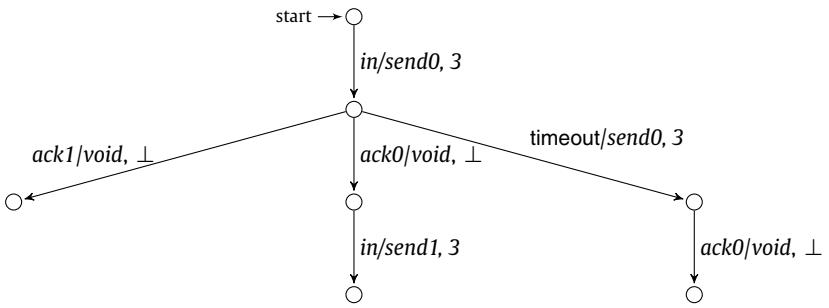

Fig. 4. Observation tree for the MM1T of Fig. 1.

2. Its graph is a tree, that is, there is a unique untimed run to each state.

We say that $\mathcal{T}$ is an *observation tree for* MM1T $\mathcal{M}$ iff there is a refinement function from $\mathcal{T}$ to $\mathcal{M}$, that is, a function $f : Q_{\mathcal{T}} \to Q_{\mathcal{M}}$ that preserves the initial state, transitions, and whether the timer is on/off. Formally, we require for every $q, q' \in Q^{\mathcal{T}}$, $i \in I$, $o \in O$ and $n \in \mathbb{N}^{>0} \cup \{\bot\}$,

$$\begin{aligned} & f(q_0^{\mathcal{T}}) = q_0^{\mathcal{M}} \\ q \xrightarrow{i/o,n} q' \;\Rightarrow\; & f(q) \xrightarrow{i/o,n} f(q') \\ q \in Q_{on}^{\mathcal{T}} \;\Leftrightarrow\; & f(q) \in Q_{on}^{\mathcal{M}} \end{aligned}$$

A refinement function is a (functional) bisimulation as in Definition 2, except that it does not have to satisfy transfer condition (8). Note that if $\mathcal{T}$ is an observation tree for $\mathcal{M}$, any untimed word of $\mathcal{T}$ is also an untimed word of $\mathcal{M}$. Fig. 4 shows an observation tree for our running example. There is an obvious refinement function that maps each state in this tree to a unique state of the MM1T of Fig. 1, that preserves the initial state, and such that each transition in the tree corresponds to a transition in the MM1T of Fig. 1.

The adaptor poses timed membership queries to $\mathcal{M}$ to construct an observation tree $\mathcal{T}$ for $\mathcal{M}$ which is then used to answer membership queries for Mealy($\mathcal{M}$). Initially, the adaptor starts with a trivial observation tree $\mathcal{T}$ with a single state with the timer off and no transitions. Trivially, the function that maps the single, initial state of $\mathcal{T}$ to the initial state of $\mathcal{M}$ is a refinement from $\mathcal{T}$ to $\mathcal{M}$. Tree $\mathcal{T}$ and refinement f are then extended one state at a time, using the following lemma.

Lemma 3. *Assume the adaptor knows an upper bound Δ on the maximal timer value occurring in $\mathcal{M}$. Let $\mathcal{T}$ be an observation tree for $\mathcal{M}$, let q be a state of $\mathcal{T}$ without an outgoing transition for $i \in I$ such that $i = \mathit{timeout} \Rightarrow q \in Q_{on}$. Then, via a single timed membership query, the adaptor may construct an observation tree $\mathcal{T}'$ for $\mathcal{M}$ that extends $\mathcal{T}$ with an outgoing i-transition for q.*

Proof. The adaptor first adds a new state q' to tree $\mathcal{T}$ and set $\delta(q, i) = q'$. In order to determine the values for $\lambda(q, i)$ and $\tau(q, i)$, and decide whether the timer should be on or off in q', the adaptor constructs a timed membership query. Let α be the unique untimed run of $\mathcal{T}$ leading to state q:

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

The adaptor constructs a corresponding timed run α' of $\mathcal{T}$:

$$\alpha' = C_0 \xrightarrow{d_0} C_0' \xrightarrow{i_0/o_0} C_1 \xrightarrow{d_1} C_1' \xrightarrow{i_1/o_1} C_2 \cdots \xrightarrow{d_k} C_k' \xrightarrow{i_k/o_k} C_{k+1},$$

with $C_j = (q_j, t_j)$ and $C_j' = (q_j, t_j')$. For indices j with $i_j \neq \mathsf{timeout}$, the adaptor chooses $d_j = 0$, that is, it performs these inputs as fast as possible. By definition of a timed run, this choice determines the values of all t_j's and t_j''s. For indices j with $i_j = \mathsf{timeout}$, the adaptor waits until the timeout occurs, so we have $d_j = t_j$ by definition of a timed run, and thus α' is fully determined. Note that each t_j is either a positive integer or ∞, for all j: if the timer is off in q_j then $t_j = \infty$, and if it is on then t_j still equals the value to which the timer was last set. This implies that $t_j' \geq 1$ for indices j with $i_j \neq \mathsf{timeout}$. Let w be the timed word corresponding to α'. The adaptor poses the following timed membership query u:

$$u = \begin{cases} tiw(w) \bullet (t_{k+1} + \Delta) & \text{if } i = \mathsf{timeout} \\ tiw(w) \bullet (\frac{1}{2}\, i\, \Delta) & \text{if } i \neq \mathsf{timeout} \end{cases}$$

Since there is a refinement from $\mathcal{T}$ to $\mathcal{M}$, timed input word $tiw(w)$ will drive $\mathcal{M}$ from its initial configuration with timed word w to a configuration (r, t_{k+1}), where r is the image of state q under the refinement. Actually, the timed run to state r in $\mathcal{M}$ is identical to the timed run α' of $\mathcal{T}$, except for the states. Since $t'_j \geq 1$ for indices j with $i_j \neq \mathsf{timeout}$, no race conditions will occur and the timed run of $\mathcal{M}$ is uniquely determined. Upon receiving timed membership query u, the teacher will return the unique timed word w' in $tmq_{\mathcal{M}}(u)$. The adaptor distinguishes between two cases.

1. If $i = \mathsf{timeout}$ then, since $q \in Q_{on}$, the timer is also on in state r. Therefore, after timed input word $tiw(w)$ and a subsequent delay of t_{k+1}, a timeout event will occur in $\mathcal{M}$ with some output o, leading to a configuration with state r'. The adaptor sets $\lambda(q, i) = o$. In this case, the refinement maps state q' to state r'. If w' contains no further timeout event then the adaptor sets $\tau(q, i) \uparrow$ and adds q' to set Q_{off}. If w' does contain a subsequent timeout event, then the adaptor measures the (integer valued) time delay n between the timeout event from state r and the timeout event from state r'. It sets $\tau(q, i) = n$ and adds q' to set Q_{on}. Note that in this case the timer is on in r', as required for a refinement function.
2. If $i \neq \mathsf{timeout}$ then, after timed input word $tiw(w)$ and a subsequent delay of $\frac{1}{2}$, an i-event will occur in w' with some output o, leading to a configuration with state r' in $\mathcal{M}$. The adaptor sets $\lambda(q, i) = o$. In this case, the refinement maps state q' to state r'. If w' contains no timeout event following the input i, the adaptor concludes the timer is off in state r'. It sets $\tau(q, i) \uparrow$ and adds q' to set Q_{off}. If w' does contain a subsequent timeout event, then the adaptor concludes the timer is on in r' and adds q' to set Q_{on}. The adaptor measures the time delay d between the i-event from state r and the subsequent timeout event from state r'. Either d will be equal to $t_{k+1} - \frac{1}{2}$ and it sets $\tau(q, i) \uparrow$ (this is the case where the timer remains on after the i-transition and is not reset), or d will be integer valued and the adaptor sets $\tau(q, i) = d$ (this is the case where the timer is reset after the i-transition). □

Theorem 3. *Assume there is a teacher who answers timed membership queries for some MM1T $\mathcal{M}$ posed by the adaptor. Assume the adaptor knows an upper bound Δ on the maximal timer value occurring in $\mathcal{M}$. Then the adaptor may answer membership queries for* Mealy($\mathcal{M}$).

Proof. Suppose the adaptor needs to answer a membership query $\sigma \in I^*$. The adaptor uses an observation tree $\mathcal{T}$ for $\mathcal{M}$ to accomplish this task, which is extended whenever needed. It may start from any observation tree, in particular the trivial observation tree with a single state and no transitions. The adaptor records: (1) the *current state* q, initialized to the initial state of $\mathcal{T}$, (2) the *remaining suffix* ρ, initialized to σ, and (3) the *computed answer* w, initialized to the empty sequence ϵ. Now basically, the adaptor may read out the value of $mq_{\mathsf{Mealy}(\mathcal{M})}(\sigma)$ from the observation tree. Since $\mathcal{T}$ is an observation tree for $\mathcal{M}$, there exists a refinement from $\mathcal{T}$ to $\mathcal{M}$. The refinement ensures that the answer to the membership query extracted from $\mathcal{T}$ will be the same as the answer obtained from $\mathcal{M}$:

- If the remaining suffix ρ is ϵ then it returns the computed answer w.
- Otherwise, if $\rho = \mathsf{timeout}\ \rho'$ and the timer is off in q, then it sets ρ to ρ' and appends nil to w.
- Otherwise, if $\rho = i\ \rho'$ then the adaptor first ensures, using Lemma 3, that state q has an outgoing i-transition. Next it sets ρ to ρ', appends the pair $(\lambda(q, i), \tau(q, i))$ to w, and updates the current state q to $\delta(q, i)$.

The corresponding pseudocode is displayed in Algorithm 1. □

Algorithm 1 Answering a membership query for Mealy($\mathcal{M}$).

```
1:  function MQ(σ)
2:      q ← q_0^T                                   ▷ initialize current state in T
3:      ρ ← σ                                       ▷ initialize remaining suffix
4:      w ← ϵ                                       ▷ initialize computed outputs
5:      while ρ ≠ ϵ do
6:          i ← Head(ρ)
7:          ρ ← Tail(ρ)
8:          if i = timeout and q ∈ Q_off^T then
9:              w ← append(w, nil)
10:         else
11:             if δ^T(q, i) ↑ then
12:                 Extend(T, q, i)                 ▷ procedure from the proof of Lemma 3
13:             w ← append(w, (λ^T(q, i), τ^T(q, i)))
14:             q ← δ^T(q, i)
15:     return w
```

Query complexity In order to add a new state to the observation tree, our algorithm needs one timed membership query on the MM1T. Thus, starting from the trivial observation tree, it will need at most m timed membership queries on the MM1T to answer a single membership query with m input symbols from a Mealy machine learner. The fastest existing

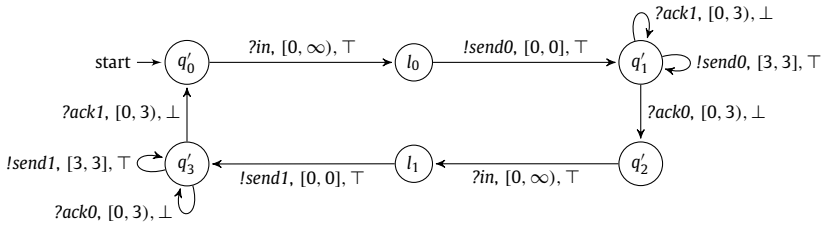

Fig. 5. DOTA model of alternating-bit protocol sender.

learning algorithms for Mealy machines (e.g. TTT [20] and $L^{\#}$ [26]) have a query complexity of $O(kn^2 + n \log m)$, where k is the number of inputs, n the number of states, and m bounds the length of the longest counterexample and the number of symbols in each query. This means that our algorithm needs $O(kmn^2 + mn \log m)$ timed membership queries to learn an MM1T with n states, k inputs, and up to m symbols in the longest counterexample. Thus, learning MM1Ts has a higher query complexity than learning Mealy machines, but 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 will be comparable in practice. It is interesting to compare the complexity of our algorithm with the complexity of the algorithm of An et al. [3] for learning DOTAs. The DOTA algorithm of An et al. [3] is exponential due to the need of guessing for each transition whether it resets the clock. In contrast, in the slightly more restrictive setting of MM1Ts, learning is polynomial because a single timed membership query suffices to determine whether a transition sets a timer.

Learning the maximum timer value In case the adaptor does not know a bound Δ to the maximal timer value occurring in $\mathcal{M}$, Δ can be assigned some arbitrary value in $\mathbb{N}^{>0}$. If this initial estimate is greater than or equal than the maximum timer value in $\mathcal{M}$, 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, the adaptor then updates Δ and start learning from scratch.

4. From MM1T to DOTA learning

In order to compare our approach to those of [2,3], we translate MM1Ts to Deterministic One-Clock Timed Automata (DOTAs). DOTAs are an extension of deterministic finite automata (DFAs) in which transitions are labeled with an action, a clock guard that is an interval on allowed clock values, and a Boolean that indicates whether or not the clock is reset to zero. In order to ensure determinism, for each state q and action a, the guards of the outgoing a-transitions of q must be disjoint. For the formal definition of DOTAs we refer to [3]. Here we illustrate our translation from MM1Ts to DOTAs with an example; for the full translation see Appendix A, where we also compare the expressiveness of MM1Ts and DOTAs. Fig. 5 shows an example of a DOTA, obtained as the translation of the alternating-bit protocol MM1T from Fig. 1.

In our translation, we split all transitions of the MM1T into an input and an output transition in the DOTA, except for the transitions with input timeout or output *void*, which are encoded by a single transition. For instance, we encode the transition $q_0 \to q_1$ into transitions $q'_0 \to l_0$ and $l_0 \to q'_1$. The guard $[0, \infty)$ indicates that the input transition can be taken at any time, and the $\top$ specifies that the clock is reset, which is needed to enforce that the output transition is taken right after the input transition.

Transitions of the MM1T with a *void* output, are translated to a single transition in the DOTA. The self loop on q_1 labeled *ack1*/*void*, for instance, is represented by a self-loop labeled *ack1* on q'_1 in the DOTA.

An MM1T transition that sets the timer corresponds to a DOTA transition that resets the clock and has appropriate clock guards on subsequent states where the timer is active. For instance, the transition $q_0 \to q_1$ sets the timer to 3. In the DOTA, this is reflected by a clock reset on the transition $l_0 \to q'_1$. Next, a timeout event will occur in q_1 at time 3, causing a *send*. The timeout-transition in the MM1T corresponds to a self loop on q'_1 in the DOTA with clock guard $[3, 3]$ and action *!send*.

5. Case studies

Through an adapter that maintains an observation tree, we reduced learning MM1Ts to learning Mealy machines; please see Fig. 3 and Sect. 3.4. We implemented an adaptor that interacts with LearnLib [24] so we can benefit from all optimizations already integrated into this well maintained tool for learning Mealy machines.[1] In this section, we first apply our implementation of MM1T learners on a real-world case study. Next, we compare the performance of MM1T learners against algorithms introduced in [2,3] in the context of DOTA learning.

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

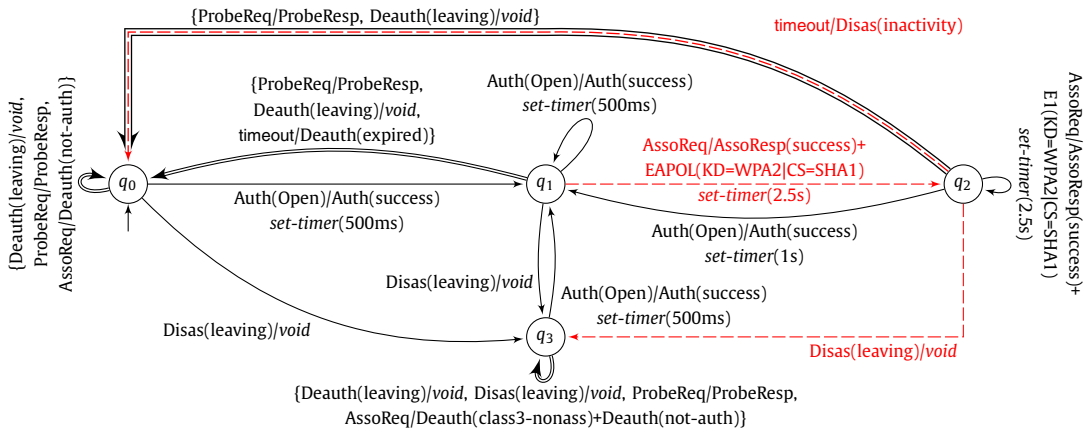

Fig. 6. MM1T of a Huawei Mate10-lite that captures granting uncontrolled port. Double and triple edges represent a set of transitions. We rounded timer values to the nearest 500 ms and marked specification violations with dashed red lines.

5.1. 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 [27, 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 [28], we focus on learning the AKM service using the following management frames: Auth(Open), AssoReq, Deauth(leaving), Disas(leaving), ProbeReq, and timeout [27, p. 45–49].

5.1.1. Learning setup

We instantiated our learner with L^*_M [19] using the MM1T membership oracle. For counterexample processing we used *Rivest & Schapire's* method [29]. We close tables using *close shortest* strategy. Finally, for equivalence oracle, we use *RandomWord* with 1000 words of variable length in $[4, 11]$.

Our learning experiments resulted in the MM1T shown in Fig. 6. 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.

5.2. Performance comparison against DOTA learners

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.

5.2.1. Learning setup

We used the MM1T membership oracle and for counterexample processing we used *Rivest & Schapire's* method [29]. We closed the tables using *close shortest* strategy. For a more thorough study, we considered two different Mealy learners and two different equivalence oracles. For the Mealy learners we chose L^*_M [19] and TTT [20]. Finally, for equivalence oracles, we used *RandomWord* with 1000 words of variable length in $[4, 11]$ and *Wp-Method* with depth 1 [30].

5.2.2. Benchmarks

Our benchmark set consists of the AKM (Section 5.1), the TCP Connection State Diagram ([31, p. 23]), a car alarm system (CAS) [2], and a particle counter (PC) [2]. 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.

Table 1
Benchmarks in terms of state-space ($|\mathbf{S}|$) and input size ($|\mathbf{\Sigma}|$).

Model	AKM		TCP		CAS		PC		Light		Train	
	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$	$\|\mathbf{S}\|$	$\|\mathbf{\Sigma}\|$
MM1T	4	5	11	8	8	4	8	8	4	2	6	3
DOTA	15	12	20	13	14	10	26	14	5	5	6	6

Table 2
Total number of SUL resets; **best** performance per experiment.

Learner	Eq. Oracle	AKM	TCP	CAS	PC	Light	Train
MM1T-L^*_M	RandomWord	5418	403	498	377	112	38
MM1T-L^*_M	Wp-Method	**461**	**257**	**161**	530	**28**	**34**
MM1T-TTT	RandomWord	5418	580	490	**325**	118	41
MM1T-TTT	Wp-Method	**461**	340	209	604	31	35
GTALearn	Random Walk	3147	1866	1500	2146	1044	1287
GTALearn*	Random Walk	3147	384	351	2146	32	83
OTALearn*	N/A	2103	2924	1448	10003	167	325
OTALearn	N/A	———————— timeout ————————					

5.2.3. Algorithms

Tables 2 and 3 show benchmark results for MM1T and DOTA learners. Following is the list of DOTA learners with which we compare:

- GTALearn represents the learning algorithm by Aichernig et al. [2] with a maximum of 1000 tests while performing equivalence checks.
- GTALearn* is the previous algorithm with the following optimizations:
  1. Performs minimal number of tests for equivalence checks.[2]
  2. Remembers an optimal set of counterexamples after each generation.
- OTALearn represents the learning algorithm by An et al. [3] using a normal teacher and timed out on all benchmarks.
- OTALearn* is the previous algorithm using a "smart" teacher that provides the clock reset information.

Equivalence oracles The choice of an equivalence oracle has a direct effect on the learner's performance. To show there is not much contrast between a naïve oracle and one that implements structural testing, we chose *RandomWord* and *Wp-Method*. Since OTALearn implements its own equivalence oracle, we do not report the metrics for equivalence check. GTALearn uses a *RandomWalk* method. For a fair comparison with the *RandomWord* equivalence oracle used by MM1T learners, we asked the authors of GTALearn to configure their equivalence oracle to perform 1000 words of desired length for each hypothesis.

Maximum timer value OTALearn* is provided with the exact clock reset values. GTALearn only requires a maximum clock value that is greater or equal to the SUL's maximal clock value. Meanwhile, MM1T learners can infer the maximum timer value aposteriori, but we provide the MM1T learners with a maximum timer value for fair comparison against the GTALearn algorithms.

5.2.4. Performance metrics

Since OTALearn implements its own equivalence oracle and GTALearn is not an Angluin style algorithm, we study the numbers of resets and inputs performed rather than the number of membership and equivalence queries. From a practical perspective, it is clear that the time required for realizing a query on the SUL grows at least linearly in its length. We therefore believe that the numbers of resets and inputs provide the best performance metric for comparing learning algorithms.

Table 2 reports the total number of SUL resets. We dedicated a row to each learning setup. The first column gives the learning algorithm, the second column the equivalence oracle it used, and we dedicated a column to each experiment in our benchmark. Note that, we ran GTALearn and MM1T experiments that use RandomWord equivalence oracle 10 times and report the average case instead. Table 3 has a similar layout and reports the total number of performed inputs.

Tables 4 to 9 show an in-depth performance breakdown for learning setups.[3] The number of membership queries posed by the learner (the equivalence oracle) is reported in the *Hypothesis Learning* (resp. *Equivalence Checking*) half of the table. The total number of membership queries is less than the number of SUL reset for MM1T learners. This is due to maintaining an observation tree; see the discussion on query complexity in Sect. 1. The observation tree can act as a caching mechanism,

[2] Once learned a correct hypothesis, GTALearn* halts and will not pose the last equivalence query requiring 1000 tests; hence, we compare against GTALearn.

[3] These tables denote by N/A and N/R, respectively, where performance criteria do not apply to an algorithm or are not reported.

Table 3
Total number of performed inputs; **best** performance per experiment.

Learner	Eq. Oracle	AKM	TCP	CAS	PC	Light	Train
MM1T-L^*_M	RandomWord	30072	1874	2599	1790	529	107
MM1T-L^*_M	Wp-Method	**1620**	**1096**	**672**	2709	**85**	**95**
MM1T-TTT	RandomWord	30127	2987	2561	**1538**	560	129
MM1T-TTT	Wp-Method	**1620**	1535	883	3125	96	98
GTALearn	RandomWalk	94735	28382	40273	23118	19887	18808
GTALearn*	RandomWalk	94735	13554	9564	23118	940	2338
OTALearn*	N/A	356762	86880	3791091	3540458	26545	29872
OTALearn	N/A	——————————— timeout ———————————					

Table 4
Performance breakdown for the AKM experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	174	174	**60**	66	N/R	N/R	2806
− Tree Queries	53	53	15	29	N/A	N/A	N/A
= SUL Queries	121	121	45	**37**	72	72	N/A
Equivalence Checking							
MQs	1000	**227**	1015	295	N/R	N/R	N/A
− Tree Queries	180	109	180	161	N/A	N/A	N/A
= SUL Queries	820	**118**	835	134	3147	3147	N/A
EQs	**1**	**1**	3	3	16	16	52
SUL Resets	5418	**461**	5418	**461**	3147	3147	2103
SUL Inputs	30072	**1620**	30127	**1620**	94735	94735	356762

Table 5
Performance breakdown for the TCP experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	999	999	**516**	585	N/R	N/R	4753
− Tree Queries	837	837	418	495	N/A	N/A	N/A
= SUL Queries	162	162	98	**90**	78	87	N/A
Equivalence Checking							
MQs	**1000**	3361	1735	6600	N/R	N/R	N/R
− Tree Queries	918	3313	1578	6458	N/A	N/A	N/A
= SUL Queries	82	**48**	157	142	1857	380	N/A
EQs	**1**	**1**	8	9	20	45	24
SUL Resets	403	**257**	580	340	1866	384	2924
SUL Inputs	1847	**1096**	2987	1535	28382	13554	86880

row *Tree Queries* reports the number of queries cached in the tree. *SUL Queries* reports on the actual number of effective queries performed on the SUL. Membership queries are not relevant for GTALearn, we instead considered them as SUL Queries.

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. The TTT learner performs the best in the number of required queries to learn a correct hypothesis. The GTALearn remembers 72 out of 3147 SUL queries it asks; meanwhile, the GTALearn* could not find a better setup for this experiment.

TCP has only one timeout transition; thus, the MM1T learners do not need to reset the SUL as often. 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. (With the exception of GTALearn*.) Table 5 shows that MM1T learners hugely benefit maintaining an observation tree.

CAS and PC show a slightly more sophisticated timed behavior than TCP. For both, the MM1T learners also significantly outperform the DOTA learners. Once more, GTALearn* failed to find a more optimal setup for PC experiment. Similarly, if considering the inputs performed, OTALearn* straggles by three orders of magnitude. Tables 6 and 7 show a similar pattern to that of TCP.

Light and Train are simpler systems with similar timed behavior. Tables 8 and 9 show that the MM1T learners again hugely benefit from maintaining an observation tree. In these experiments we see that both GTALearn and OTALearn* perform

Table 6
Performance breakdown for the CAS experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	245	245	**159**	186	N/R	N/R	2340
− Tree Queries	169	169	113	139	N/A	N/A	N/A
= SUL Queries	76	76	**46**	47	49	55	N/A
Equivalence Checking							
MQs	1000	**526**	1012	734	N/R	N/R	N/A
− Tree Queries	866	489	866	655	N/A	N/A	N/A
= SUL Queries	134	**37**	146	79	1477	311	N/A
EQs	**1**	**1**	3	4	13	9	26
SUL Resets	498	161	490	**209**	1500	351	1448
SUL Inputs	2599	672	2561	**883**	40273	9564	3791091

Table 7
Performance breakdown for the PC experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	890	890	**323**	399	N/R	N/R	16973
− Tree	660	660	214	317	N/A	N/A	N/A
= SUL Queries	230	230	109	82	**59**	**59**	N/A
Equivalence Checking							
MQs	**1000**	2750	1389	6219	N/R	N/R	N/A
− Tree	970	2579	1320	5935	N/A	N/A	N/A
= SUL Queries	**30**	171	69	284	2146	2146	N/A
EQs	**1**	**1**	5	6	16	16	36
SUL Resets	377	530	**325**	604	2146	2146	10003
SUL Inputs	1790	2709	**1538**	3125	23118	23118	3540458

Table 8
Performance breakdown for the Light experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	51	51	**34**	37	N/R	N/R	182
− Tree	35	35	25	27	N/A	N/A	N/A
= SUL Queries	16	16	**9**	10	18	**9**	N/A
Equivalence Checking							
MQs	1000	**85**	1003	118	N/R	N/R	N/A
− Tree	961	79	960	105	N/A	N/A	N/A
= SUL	39	**6**	43	13	1039	31	N/A
EQs	**1**	**1**	3	3	5	9	8
SUL Resets	112	**28**	118	31	1044	32	167
SUL Inputs	529	**85**	560	96	19887	940	26545

considerably better than they did in previous experiments; yet, OTALearn* is still behind by two orders of magnitude in the number of inputs.

Finally, in the context of learning DOTAs, the MM1T learners are faster than the DOTA learners often by orders of magnitude. We primarily attribute the performance of MM1T learners to the underlying observation tree and the compact formalism of MM1Ts compared to DOTAs. We backed this, by comparing MM1T learners with OTALearn*. Our benchmark revealed the potentials of GTALearn, especially when compared to OTALearn*; since both target TAs.[4]

[4] According to its authors, "GTALearn does not implement any caching mechanism, but its random walk equivalence can benefit from it. Optimizing the interaction with the SUL was not a primary objective while implementing GTALearn, therefore the learner re-executes all conformance queries on the SUL for each equivalence check".

Table 9
Performance breakdown for the Train experiment.

Learning Algorithm	L^*_M	L^*_M	TTT	TTT	GTALearn	GTALearn*	OTALearn
Equivalence Oracle	Random Word	Wp-Method	Random Word	Wp-Method	Random Walk	Random Walk	N/A
Hypothesis Learning							
MQs	124	124	**111**	116	N/R	N/R	408
− Tree Queries	95	95	92	97	N/A	N/A	N/A
= SUL Queries	29	29	**19**	**19**	16	7	N/A
Equivalence Checking							
MQs	1000	**337**	1012	657	N/R	N/R	N/A
− Tree Queries	994	332	1002	644	N/A	N/A	N/A
= SUL Queries	6	**5**	10	13	1279	80	N/A
EQs	**1**	**1**	3	5	5	7	12
SUL Resets	38	**34**	40	35	1287	83	325
SUL Inputs	107	**95**	125	98	18808	2338	29872

6. Conclusion & 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 [2] et al. and An et al. [3].

For our experiments we did not need to implement a timed equivalence oracle for MM1Ts, since we could use the approximation of an equivalence oracle for Mealy machines as implemented in LearnLib. It would be interesting to implement a timed equivalence oracle for MM1Ts directly, for instance using the timed testing tool Uppaal TRON [32], and compare its performance with the indirect implementation using LearnLib.

An obvious direction for future research is to extend our work to Mealy machines with multiple timers. In this setting, a challenge for the learner is to figure out which specific timers are started in a transition, as this information cannot be observed directly. We believe 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 [2] 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 gray-box techniques for model learning [33] 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 and software, which makes it important to push these techniques to their limits.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgment

We would like to thank Bernhard Aichernig, Andrea Pferscher and Miaomiao Zhang for their constructive comments and their help with running the benchmarks on their tools [2,3].

Appendix A. Translation from MM1Ts to DOTAs

Our translation only works for MM1Ts that satisfy a technical restriction (met by all benchmarks that we consider): if $q \xrightarrow{i/o,\perp} q'$ is a transition between two states $q, q' \in Q_{on}$ then $o = void$. Thus any transition between states in which the timer is on either resets the timer or has no observable output. We need this restriction to exclude MM1Ts such as the one from Fig. A.7, for which no equivalent DOTA exists. In this MM1T an event *begin* is followed by an event *end* after exactly 3 time

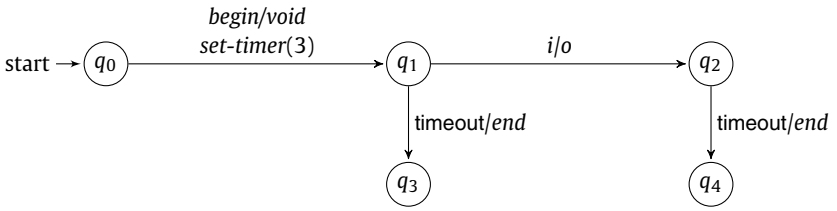

Fig. A.7. MM1T for which no equivalent DOTA exists.

units, and in between there may be an event i that is instantly followed by an event o. Such a behavior cannot be encoded in a DOTA: the only way to ensure that event i is instantly followed by event o is to reset the clock when i occurs and to add a guard $[0,0]$ on the subsequent o-transition. But then the DOTA can no longer enforce that the *end* event occurs exactly 3 time units after the *begin* event.

Let $\mathcal{M} = (I, O, Q, q_0, \delta, \lambda, \tau)$ be an MM1T that satisfies the above restriction. Then, generalizing the example of Fig. 5, we construct a corresponding DOTA $\mathcal{A}$ as follows:

1. The set Σ of actions of $\mathcal{A}$ consists of all inputs of $\mathcal{M}$ (except timeout) prefixed with '?' and all outputs of $\mathcal{M}$ (except *void*) prefixed with '!':

$$\Sigma = \{?i \mid i \in I \setminus \{\text{timeout}\}\} \cup \{!o \mid o \in O \setminus \{void\}\}$$

2. Let T be the set of all values that can be assigned to the timer by some transition of $\mathcal{M}$:

$$T = \{n \in \mathbb{N}^{>0} \mid \exists q \in Q\ \exists i \in I : \tau(q,i) = n\}$$

DOTA $\mathcal{A}$ has three types of states:
   (a) States from Q_{off}.
   (b) States from $Q_{on} \times T$ (when the timer is on, we remember the latest value that has been assigned to it as part of the state).
   (c) Intermediate states between an input and an observable output, needed as we split transitions of $\mathcal{M}$ into an input and an output part:

$$(Q_{off} \times (I \setminus \{\text{timeout}\})) \cup (Q_{on} \times T \times I)$$

3. All states of $\mathcal{A}$ are accepting states.
4. In the definition of the transitions of $\mathcal{A}$ we need to consider many cases, for any transition $q \xrightarrow{i/o,n} q'$ of $\mathcal{M}$, depending on whether or not $q \in Q_{off}$, $i = \text{timeout}$, $o = void$, $q' \in Q_{off}$, and $n = \perp$. Altogether DOTA $\mathcal{A}$ has thirteen types of transitions:
   (a) For each $q \in Q_{off}$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) \neq void$, a transition $q \xrightarrow{i,[0,\infty),\top} (q,i)$
   (b) For each $q \in Q_{off}$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = o \neq void$ and $\delta(q,i) = q' \in Q_{off}$, a transition $(q,i) \xrightarrow{o,[0,0],\perp} q'$
   (c) For each $q \in Q_{off}$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = o \neq void$, $\delta(q,i) = q' \in Q_{on}$ and $\tau(q,i) = n$, a transition $(q,i) \xrightarrow{o,[0,0],\top} (q',n)$
   (d) For each $q \in Q_{off}$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = void$ and $\delta(q,i) = q' \in Q_{off}$, a transition $q \xrightarrow{i,[0,\infty),\perp} q'$
   (e) For each $q \in Q_{off}$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = void$, $\delta(q,i) = q' \in Q_{on}$ and $\tau(q,i) = n$, a transition $q \xrightarrow{i,[0,\infty),\top} (q',n)$
   (f) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) \neq void$, a transition $(q,n) \xrightarrow{i,[0,n),\top} (q,n,i)$
   (g) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = o \neq void$ and $\delta(q,i) = q' \in Q_{off}$, a transition $(q,n,i) \xrightarrow{o,[0,0],\perp} q'$
   (h) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = o \neq void$, $\delta(q,i) = q' \in Q_{on}$ and $\tau(q,i) = n'$, a transition $(q,n,i) \xrightarrow{o,[0,0],\top} (q',n')$
   (i) For each $(q,n) \in Q_{on} \times T$ with $\lambda(q,\text{timeout}) = o$ and $\delta(q,\text{timeout}) = q' \in Q_{off}$, a transition $(q,n) \xrightarrow{o,[n,n],\perp} q'$
   (j) For each $(q,n) \in Q_{on} \times T$ with $\lambda(q,\text{timeout}) = o$, $\delta(q,\text{timeout}) = q' \in Q_{on}$ and $\tau(q,\text{timeout}) = n'$, a transition $(q,n) \xrightarrow{o,[n,n],\top} (q',n')$
   (k) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = void$ and $\delta(q,i) = q' \in Q_{off}$, a transition $(q,n) \xrightarrow{i,[0,n),\perp} q'$
   (l) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\text{timeout}\}$ with $\lambda(q,i) = void$, $\delta(q,i) = q' \in Q_{on}$ and $\tau(q,i) \uparrow$, a transition $(q,n) \xrightarrow{i,[0,n),\perp} (q',n)$

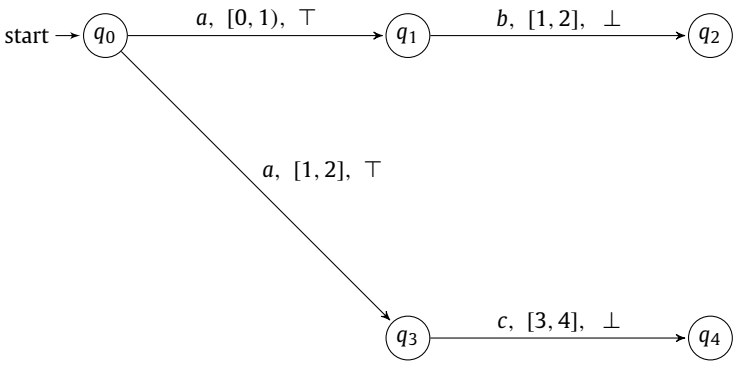

Fig. A.8. DOTA for which no equivalent MM1T exists.

(m) For each $(q,n) \in Q_{on} \times T$ and $i \in I \setminus \{\mathsf{timeout}\}$ with $\lambda(q,i) = void$, $\delta(q,i) = q' \in Q_{on}$ and $\tau(q,i) = n'$, a transition $(q,n) \xrightarrow{i,[0,n),\top} (q',n')$

Subtle differences between MM1Ts and DOTAs make it difficult to define a translation that fully preserves the semantics. For instance, due to race conditions an MM1T does not always have a unique timed run for a given timed input word. In contrast, a DOTA is deterministic by definition in the sense that for each timed input word (called delay-timed word in [3]) there is a unique (timed) run. In order to rule out nondeterminism, the guards of the *ack* transitions of the DOTA in Fig. 5 are right-open. As a result, this DOTA does not have a run corresponding to the following timed run of the MM1T of Fig. 1:

$$(q_0,\infty) \xrightarrow{0} (q_0,\infty) \xrightarrow{in/send0} (q_1,3) \xrightarrow{3} (q_1,0) \xrightarrow{ack0/void} (q_1,0)$$

When we exclude race conditions and only consider timed runs of $\mathcal{M}$ in which inputs (except timeout) never occur when the value of the timer is 0 then, for each of these timed runs, the DOTA $\mathcal{A}$ has a run (in the sense of [3]) with exactly the same observable events and exactly the same time delays between events.

There are also DOTAs for which no equivalent MM1Ts exists. In the initial state of an MM1T there is no timer running, and thus we cannot prevent an MM1T from staying in its initial state indefinitely. In DOTAs, however, guards may appear on the outgoing transitions of the initial state, enforcing real-time behavior. Also, an MM1T cannot express that an event will occur in an interval $[t_1,t_2]$ after arriving in a state when $0 < t_1 < t_2$. Thus for instance, unless we consider equivalences that abstract from internal transitions [34], it is not possible to translate the DOTA of Fig. A.8 to an equivalent MM1T.

References

[1] F. Vaandrager, R. Bloem, M. Ebrahimi, Learning Mealy machines with one timer, in: A. Leporati, C. Martín-Vide, D. Shapira, C. Zandron (Eds.), Language and Automata Theory and Applications - 15th International Conference, LATA 2021, Milan, Italy, March 1-5, 2021, Proceedings, in: Lecture Notes in Computer Science, vol. 12638, Springer, 2021, pp. 157–170.
[2] B.K. Aichernig, A. Pferscher, M. Tappler, From passive to active: learning timed automata efficiently, in: R. Lee, S. Jha, A. Mavridou (Eds.), NFM'20, in: LNCS, vol. 12229, Springer, 2020, pp. 1–19.
[3] J. An, M. Chen, B. Zhan, N. Zhan, M. Zhang, Learning one-clock timed automata, in: A. Biere, D. Parker (Eds.), TACAS'20, in: LNCS, vol. 12078, Springer, 2020, pp. 444–462.
[4] J.d. Ruiter, E. Poll, Protocol state fuzzing of TLS implementations, in: USENIX Security Symp., USENIX, 2015, pp. 193–206.
[5] P. Fiterău-Broştean, R. Janssen, F. Vaandrager, Combining model learning and model checking to analyze TCP implementations, in: S. Chaudhuri, A. Farzan (Eds.), Proceedings 28th International Conference on Computer Aided Verification (CAV'16), Toronto, Ontario, Canada, in: Lecture Notes in Computer Science, vol. 9780, Springer, 2016, pp. 454–471.
[6] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, P. Verleg, Model learning and model checking of SSH implementations, in: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017, ACM, New York, NY, USA, 2017, pp. 142–151.
[7] P. Fiterău-Broştean, F. Howar, Learning-based testing the sliding window behavior of TCP implementations, in: FMICS, in: LNCS, vol. 10471, 2017, pp. 185–200.
[8] P. Fiterău-Broştean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, J. Somorovsky, Analysis of DTLS implementations using protocol state fuzzing, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 2523–2540.
[9] T. Ferreira, H. Brewton, L. D'Antoni, A. Silva, Prognosis: closed-box analysis of network protocol implementations, in: F.A. Kuipers, M.C. Caesar (Eds.), ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021, ACM, 2021, pp. 762–774.
[10] F. Vaandrager, Model learning, Commun. ACM 60 (2) (2017) 86–95, https://doi.org/10.1145/2967606.
[11] F. Howar, B. Steffen, Active automata learning in practice, in: A. Bennaceur, R. Hähnle, K. Meinke (Eds.), Machine Learning for Dynamic Software Analysis: Potentials and Limits: International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24–27, 2016, Revised Papers, Springer International Publishing, 2018, pp. 123–148.
[12] R. Alur, D. Dill, A theory of timed automata, Theor. Comput. Sci. 126 (1994) 183–235.
[13] O. Grinchtein, B. Jonsson, P. Pettersson, Inference of event-recording automata using timed decision trees, in: CONCUR, in: LNCS, vol. 4137, 2006, pp. 435–449.
[14] O. Grinchtein, B. Jonsson, M. Leucker, Learning of event-recording automata, Theor. Comput. Sci. 411 (47) (2010) 4029–4054.

[15] L. Henry, T. Jéron, N. Markey, Active learning of timed automata with unobservable resets, in: N. Bertrand, N. Jansen (Eds.), Formal Modeling and Analysis of Timed Systems - 18th International Conference, FORMATS 2020, Vienna, Austria, September 1-3, 2020, Proceedings, in: Lecture Notes in Computer Science, vol. 12288, Springer, 2020, pp. 144–160.
[16] M. Tappler, B.K. Aichernig, K.G. Larsen, F. Lorber, Time to learn - learning timed automata from tests, in: É. André, M. Stoelinga (Eds.), FORMATS'19, in: Lecture Notes in Computer Science, vol. 11750, Springer, 2019, pp. 216–235.
[17] J.F. Kurose, K.W. Ross, Computer Networking: a Top-Down Approach, sixth edition, Pearson, 2013.
[18] B. Caldwell, R. Cardell-Oliver, T. French, Learning time delay Mealy machines from programmable logic controllers, IEEE Trans. Autom. Sci. Eng. 13 (2) (2016) 1155–1164.
[19] M. Shahbaz, R. Groz, Inferring Mealy machines, in: A. Cavalcanti, D. Dams (Eds.), FM 2009: Formal Methods, Second World Congress, Eindhoven, the Netherlands, November 2-6, 2009, Proceedings, in: Lecture Notes in Computer Science, vol. 5850, Springer, 2009, pp. 207–222.
[20] M. Isberner, F. Howar, B. Steffen, The ttt algorithm: a redundancy-free approach to active automata learning, in: B. Bonakdarpour, S.A. Smolka (Eds.), Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014, Proceedings, Springer International Publishing, Cham, 2014, pp. 307–322.
[21] H. Raffelt, B. Steffen, T. Berg, LearnLib: a library for automata learning and experimentation, in: FMICS '05: Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems, ACM Press, New York, NY, USA, 2005, pp. 62–71.
[22] D. Park, Concurrency and automata on infinite sequences, in: P. Deussen (Ed.), 5^{th} GI Conference, in: Lecture Notes in Computer Science, vol. 104, Springer-Verlag, 1981, pp. 167–183.
[23] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (2) (1987) 87–106.
[24] H. Raffelt, B. Steffen, T. Berg, T. Margaria, LearnLib: a framework for extrapolating behavioral models, Int. J. Softw. Tools Technol. Transf. 11 (5) (2009) 393–407.
[25] B. Steffen, F. Howar, M. Merten, Introduction to active automata learning from a practical perspective, in: M. Bernardo, V. Issarny (Eds.), Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011, Advanced Lectures, in: Lecture Notes in Computer Science, vol. 6659, Springer, 2011, pp. 256–296.
[26] F.W. Vaandrager, B. Garhewal, J. Rot, T. Wißmann, A new approach for active automata learning based on apartness, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 13243, Springer, 2022, pp. 223–243.
[27] IEEE Standard for Information technology—Telecommunications and information exchange between systems Local and metropolitan area networks—Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012). 2016, pp. 1–3534.
[28] C.M. Stone, T. Chothia, J. de Ruiter, Extending automated protocol state learning for the 802.11 4-way handshake, in: ESORICS'18, 2018.
[29] R. Rivest, R. Schapire, Inference of finite automata using homing sequences (extended abstract), in: Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing, 15-17 May 1989, Seattle, Washington, USA, ACM, 1989, pp. 411–420.
[30] S. Fujiwara, G.v. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, IEEE Trans. Softw. Eng. 17 (6) (1991) 591–603.
[31] J.E. Postel, Transmission Control Protocol, RFC 793. Sep. 1981, https://rfc-editor.org/rfc/rfc793.txt.
[32] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, A. Skou, Testing real-time systems using UPPAAL, in: R.M. Hierons, J.P. Bowen, M. Harman (Eds.), Formal Methods and Testing, an Outcome of the FORTEST Network, Revised Selected Papers, in: Lecture Notes in Computer Science, vol. 4949, Springer, 2008, pp. 77–117.
[33] F. Howar, B. Jonsson, F.W. Vaandrager, Combining black-box and white-box techniques for learning register automata, in: B. Steffen, G.J. Woeginger (Eds.), Computing and Software Science - State of the Art and Perspectives, in: Lecture Notes in Computer Science, vol. 10000, Springer, 2019, pp. 563–588.
[34] R.J. van Glabbeek, The linear time - branching time spectrum II, in: E. Best (Ed.), CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings, in: Lecture Notes in Computer Science, vol. 715, Springer, 1993, pp. 66–81.