






# Learning Symbolic Timed Models from Concrete Timed Data<sup>\*</sup>

Simon Dierl<sup>2</sup>, Falk Maria Howar<sup>2</sup>, Sean Kauffman<sup>1</sup>, Martin Kristjansen<sup>1</sup>,  
Kim Guldstrand Larsen<sup>1</sup>, Florian Lorber<sup>1</sup>, and Malte Mauritz<sup>2</sup>

<sup>1</sup> Aalborg Universitet, Aalborg, Denmark

<sup>2</sup> TU Dortmund University, Dortmund, Germany

**Abstract.** We present a technique for learning explainable timed automata from passive observations of a black-box function, such as an artificial intelligence system. Our method accepts a single, long, timed word with mixed input and output actions and learns a Mealy machine with one timer. The primary advantage of our approach is that it constructs a symbolic observation tree from a concrete timed word. This symbolic tree is then transformed into a human comprehensible automaton. We provide a prototype implementation and evaluate it by learning the controllers of two systems: a brick-sorter conveyor belt trained with reinforcement learning and a real-world derived smart traffic light controller. We compare different model generators using our symbolic observation tree as their input and achieve the best results using  $k$ -tails. In our experiments, we learn smaller and simpler automata than existing passive timed learners while maintaining accuracy.

## 1 Introduction

In recent years, machine learning has been integrated into more and more areas of life. However, the safety of such systems often cannot be verified due to their complexity and unknown internal structure. For such black-box systems, model learning can provide additional information. Model learning [14] typically deduces an executable representation either by monitoring the System Under Learning (SUL) (*passive learning*), or by prompting the SUL (*active learning*). Either approach produces a model consistent with the observations. These models can be used for verification methods like model checking, but often simply obtaining a graphical illustration of the internal workings of the system can provide an increase in confidence that it works as intended. The approach is especially useful for artificial intelligence (AI) systems, where a function is constructed from training data and no human-readable explanation might exist.

---

<sup>\*</sup> This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: [https://doi.org/10.1007/978-3-031-33170-1\\_7](https://doi.org/10.1007/978-3-031-33170-1_7). Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use <https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms>

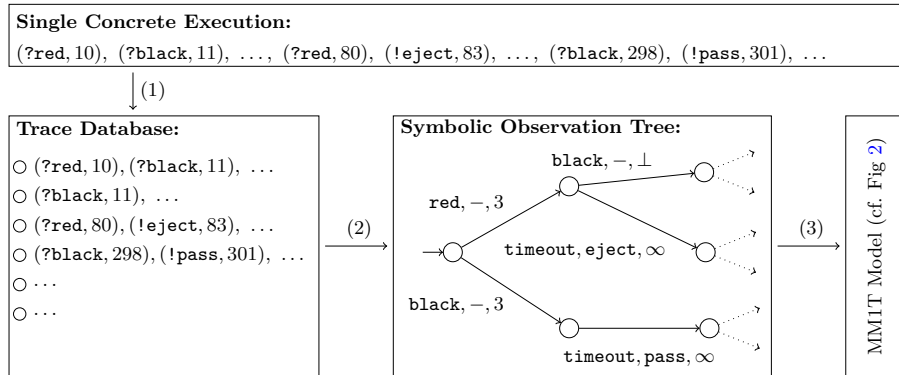


Fig. 1: Inference workflow: Find initial conditions and split into trace database (1), compute symbolic observation tree from database (2), generate MMIT (3).

Active algorithms like Angluin’s  $L^*$  [4] have shown promising results. However, they can be difficult to apply in practice, as many systems exist that provide no way for a learner to interact with them. An example of such a system could be the controller of a smart traffic light, where the inputs are the arrival of cars in a street lane. Luckily, in the modern era of big data, many systems are monitored throughout their deployment in the real world, producing log files that can span over months or years. Passive learning algorithms like (timed)  $k$ -tails [22, 23] take large numbers of such traces, convert them into a tree-like structure, and apply state-merging techniques to collapse them into cyclic automata.

We propose the generation of a symbolic oracle, which can be used as a pre-processing step to apply both active and passive learning algorithms. Figure 1 shows our proposed workflow. The first step in creating the oracle is to instantiate a *trace database* from a *single concrete execution trace* by defining symbolic conditions on what constitutes an initial location, i.e., where the system has reset to its default configuration. This approach supports learning when only a single trace is available (e.g., a long traffic log for the controller of a specific intersection). Next, we show how to build symbolic representations of the traces. These can be exhaustively enumerated to build a *symbolic observation tree*.

By constructing a trace database and symbolic observation tree from a log of system operation, we enable both active and passive learning approaches. We demonstrate active learning using the *Mealy machine learner* by Vaandager et al. [26]. Instead of the learner interacting with the system directly, it answers *membership queries* using the symbolic observation tree. For *equivalence queries* we use random runs from the trace database for estimating the correctness of the hypothesis. Since learning algorithms designed for use with a complete oracle might ask queries that the trace database cannot answer, the inferred models will, in general, not be minimal (distinguishing states for missing information). We minimize the resulting *Mealy Machine with a Single Timer (MMIT)* via a *partial Mealy minimization* algorithm that greedily merges compatible states.

For passive learning, we directly use the symbolic tree, that, by construction, contains all the information in the database. It can be transformed into a final model by existing approaches like *k-tails* [22,23], or via *partial Mealy minimization*.

**Contribution.** First, we show how to turn one long trace into a trace database of short traces starting with an initial condition. We show how to identify initial conditions along the trace, based on symbolic queries, and discuss what might constitute an initial condition in a black-box system. Then, we present how to derive a symbolic observation tree from these traces, which seems to carry the right level of abstraction for human readability. Finally, we discuss and compare several post-processing methods to retrieve human-readable automata. One of these methods shows how active learning algorithms can be applied in a passive setting. We demonstrate how the readability and explainability of the produced models provide a significant advantage over previous approaches.

**Case Studies.** We will use a brick sorter and a smart traffic controller for an intersection as demonstrating examples. The brick sorter is inspired by [17]. It randomly receives either red or black blocks. These are scanned by a color sensor and transported along a conveyor belt for three seconds. Finally, a controller will eject red bricks, and let black bricks pass through. We use a timed automaton controller, trained using reinforcement learning with UPPAAL STRATEGO, as a basis for the experiments. The inputs to the SUL are  $\{red, black\}$  and the outputs are  $\{eject, pass\}$ . The SUL contains an intentional bug: if two blocks arrive within three seconds, the variable storing the scanned color will be overwritten.

The traffic controller is based on the control system of a real intersection located in the city of Vejle in Denmark. The intersection is a four way crossing equipped with radar sensors that report the arrival of incoming cars, and can switch between five modes for the lights. Inputs are cars arriving at the different lanes of the streets. Outputs are the active traffic lights, e.g.,  $a_1 + a_2$  when the main road on both sides has a green light. We use real-world traffic data gathered over seven consecutive days, combined with outputs generated from a digital twin of the intersection. The digital twin is a model created in the tool UPPAAL according to a detailed specification of the traffic light controller.

## 2 Preliminaries

We denote the non-negative real number at which an action occurred as its *timestamp*. We refer to a finite set of actions, also known as an alphabet, as  $\Sigma = \mathcal{I} \cup \mathcal{O}$  where  $\mathcal{I}$  is the set of input actions and  $\mathcal{O}$  is the set of output actions where  $\mathcal{I} \cap \mathcal{O} = \emptyset$ . The special actions  $\top_{\mathcal{I}}$  and  $\top_{\mathcal{O}}$  are used in symbolic queries and represent *any* input or output, respectively. The partial order  $\sqsubseteq$  relates actions in  $\mathcal{I} \cup \{\top_{\mathcal{I}}\}$  such that any pair of inputs  $i_1, i_2 \in \mathcal{I}$  are incomparable when  $i_1 \neq i_2$  and  $\top_{\mathcal{I}}$  is an upper bound of  $\mathcal{I}$  ( $\forall i \in \mathcal{I}. i \sqsubseteq \top_{\mathcal{I}}$ ) and the same applies for  $\mathcal{O} \cup \{\top_{\mathcal{O}}\}$ . Given a finite alphabet  $\Sigma$ , a timed word is a pair  $\rho = \langle \sigma, \tau \rangle$  where  $\sigma$  is a non-empty finite word over the alphabet  $\Sigma$ , and  $\tau$  is a strictly increasing sequence of timestamps with the same length as  $\sigma$ . We call the set of

all finite timed words  $T\Sigma^*$ . We also write a sequence of pairs of actions in  $\Sigma$  and timestamps to represent a timed word:  $(\sigma_0, \tau_0), (\sigma_1, \tau_1), \dots, (\sigma_n, \tau_n)$ .

A constrained symbolic timed word is a pair  $\langle \mathcal{S}, \varphi \rangle$  where  $\mathcal{S}$  is a symbolic timed word and  $\varphi$  is a boolean combination of constraints on the symbolic timestamps of  $\mathcal{S}$ . The set of all symbolic timestamps is  $\mathbb{V}$ . A symbolic timed word over the finite alphabet  $\Sigma$  is a pair  $\langle \sigma, v \rangle$  where  $\sigma$  is a finite word over  $\Sigma$ , and  $v \in \mathbb{V}^*$  is a sequence of symbolic timestamps the same length as  $\sigma$ . We also write a sequence of pairs  $(\sigma_0, v_0), (\sigma_1, v_1), \dots, (\sigma_n, v_n)$  for a constrained symbolic timed word of length  $n + 1$ . We write  $\tau_i/v_i$  to denote that  $v_i$  takes the value  $\tau_i$ .

We say that a concrete timed word *models* (written  $\models$ ) a constrained symbolic timed word when both sequences of actions are equal, and the constraints on the symbolic timestamps are satisfied by the corresponding concrete timestamps. We use a partial order to relate actions instead of strict equality so that we can reuse the model's definition later with  $\top_{\mathcal{I}}$  and  $\top_{\mathcal{O}}$  as possible symbolic actions.

**Definition 1 (Modeling of Symbolic Word).** *Given a concrete timed word  $\rho = \langle \sigma, \tau \rangle$  and a constrained symbolic timed word  $\langle \mathcal{S}, \varphi \rangle$  where  $\mathcal{S} = \langle \sigma', v \rangle$ , we say that  $\rho \models \langle \mathcal{S}, \varphi \rangle$  iff for all indices  $i$  we have  $\sigma_i \sqsubseteq \sigma'_i$  and  $\tau_0/v_0 \dots \tau_n/v_n \models \varphi$ .*

*Example 1.* In the brick-sorter example, suppose a constrained symbolic timed word  $\langle \mathcal{S}, \varphi \rangle = \langle (black, v_0), (eject, v_1), v_0 + 3 \leq v_1 \rangle$  and a timed word that models it  $\rho = (black, 0), (eject, 3)$ . We see that  $\rho \models \langle \mathcal{S}, \varphi \rangle$  since  $\sigma_0 = black \sqsubseteq \mathcal{S}_0 = black$ ,  $\sigma_1 = eject \sqsubseteq \mathcal{S}_1 = eject$ , and  $0 + 3 \leq 3$ .

We concatenate constrained symbolic timed words  $\langle \mathcal{S}_1, \varphi_1 \rangle$  and  $\langle \mathcal{S}_2, \varphi_2 \rangle$  by concatenating the symbolic timed words  $\mathcal{S}_1$  and  $\mathcal{S}_2$  and conjoining the time constraints  $\varphi_1 \wedge \varphi_2$ , letting  $\langle \mathcal{S}_1, \varphi_1 \rangle \cdot \langle \mathcal{S}_2, \varphi_2 \rangle \equiv \langle \mathcal{S}_1 \cdot \mathcal{S}_2, \varphi_1 \wedge \varphi_2 \rangle$ .

### Mealy Machines with One Timer.

We learn Mealy machine models with one timer. The timer can be reset to values in  $\mathbb{N}$  on transitions. We assume a special input **timeout** that triggers the expiration of the set timer and the corresponding change of a machine's state. We also assume a special output “-” that indicates no output on a transition. We first define the structure of these Mealy machines and then specify their semantics. Our definition extends the original definition by Vaandrager et al. [26] to model explicitly when a timer is reset or disabled and to allow a timer to be set already in the initial state. For a partial function  $f : X \rightarrow Y$ , we write  $f(x) \downarrow$  to indicate that  $f$  is defined for  $x$  and  $f(x) \uparrow$  to indicate that  $f$  is not defined for  $x$ . We fix a set of actions  $\Sigma = \mathcal{I} \cup \mathcal{O}$  and use  $\mathcal{I}_{\mathbf{to}}$  as a shorthand for  $\mathcal{I} \cup \{\mathbf{timeout}\}$ .

**Definition 2 (Mealy Machine with One Timer).** *A Mealy machine with one timer (MM1T) is a tuple  $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda, \kappa, t_0 \rangle$  with*

- finite set  $\mathcal{I}$  of inputs,
- finite set  $\mathcal{O}$  of outputs disjoint from  $\mathcal{I}_{\mathbf{to}}$ ,
- set of states  $Q = Q_{\text{off}} \cup Q_{\text{on}}$ , partitioned into states with and without a timer ( $Q_{\text{off}} \cap Q_{\text{on}} = \emptyset$ ), respectively, with initial state  $q_0 \in Q$ ,

- transition function  $\delta : Q \times \mathcal{I}_{\mathbf{to}} \rightarrow Q$
- output function  $\lambda : Q \times \mathcal{I}_{\mathbf{to}} \rightarrow \mathcal{O} \cup \{-\}$ ,
- timer reset  $\kappa : Q \times \mathcal{I}_{\mathbf{to}} \rightarrow \{\infty, \perp\} \cup \mathbb{N}$ , satisfying
  - $\delta(q, i) \in Q_{\text{off}} \iff \kappa(q, i) = \infty$ , where no timer is set,
  - $q \in Q_{\text{off}} \wedge \delta(q, i) \in Q_{\text{on}} \iff \kappa(q, i) \in \mathbb{N}$ , where the timer is set,
  - $q \in Q_{\text{on}} \wedge \delta(q, i) \in Q_{\text{on}} \iff \kappa(q, i) \in \{\perp\} \cup \mathbb{N}$ , where the timer either continues or is set to a new value,
  - $\delta(q, \mathbf{timeout}) \in Q_{\text{on}} \iff q \in Q_{\text{on}}$ , where timeouts only happen if the timer is running, and
- initial timer  $t_0 \in \{\infty\} \cup \mathbb{N}$  s.t.  $t_0 = \infty$  if  $q_0 \in Q_{\text{off}}$  and  $t_0 \in \mathbb{N}$  if  $q_0 \in Q_{\text{on}}$ .

The transition function, output function, and timer reset have identical domains, i.e.,  $\delta(q, i) \uparrow \iff \lambda(q, i) \uparrow \iff \kappa(q, i) \uparrow$  for  $q \in Q$  and  $i \in \mathcal{I}$ .

For a MM1T  $\langle \mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda, \kappa, t_0 \rangle$  we write  $q \xrightarrow{i, o, t} q'$  for a transition from state  $q$  to  $q'$  for input  $i$ , output  $o$ , and new timer value  $t$ . If the given transition is possible, we must have that  $\delta(q, i) \downarrow$ .

*Example 2.* Figure 2 shows the MM1T for the brick-sorter example, as learned by our experiments, and illustrates the typical concepts of an MM1T. It shows MM1T  $\mathcal{M} = \langle \mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda, \kappa, t_0 \rangle$  with  $\mathcal{I} = \{\text{black}, \text{red}\}$ ,  $\mathcal{O} = \{\text{pass}, \text{eject}\}$ , and  $Q = \{q_0, q_1, q_2\}$ , where  $q_0 \in Q_{\text{off}}$  and  $q_1, q_2 \in Q_{\text{on}}$ . Additionally, we have that  $\delta(q_0, \text{red}) = q_1$ ,  $\lambda(q_0, \text{red}) = -$ ,  $\kappa(q_0, \text{red}) = 3$ , and  $t_0 = \infty$ . We omit the remainder of transition, output, and timer reset functions for readability.

**Untimed Semantics.** The untyped semantics maps an untyped run to the last observed output on that run: The partial function  $\llbracket \mathcal{M} \rrbracket : \mathcal{I}_{\mathbf{to}}^+ \rightarrow \mathcal{O} \times (\{\infty, \perp\} \cup \mathbb{N})$  represents the behavior of the machine at the abstract level of how the timer is affected by the inputs. The function is defined for an *untimed word*  $w = i_0, \dots, i_n \in \mathcal{I}_{\mathbf{to}}^*$  if there exists a corresponding sequence of transitions in  $\mathcal{M}$ , i.e., let  $\llbracket \mathcal{M} \rrbracket(w) \downarrow$  if transitions  $q_j \xrightarrow{i_j, o_j, t_j} q_{j+1}$  exist for  $0 \leq j < n$ , where  $q_0$  is the initial state of  $\mathcal{M}$ . We call a sequence of such transitions an *untimed run*, as there is no information on exactly when transitions are taken. Finally, if we have that  $\llbracket \mathcal{M} \rrbracket(w) \downarrow$ , the  $n^{\text{th}}$  step of a sequence  $w$  can be

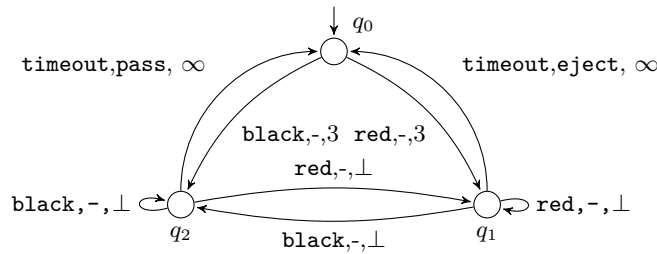


Fig. 2: Learned MM1T of the brick-sorter conveyor-belt example.

found by  $\llbracket \mathcal{M} \rrbracket(w) = (o_n, t_n)$ . Input-enabledness (i.e., totality of  $\llbracket \mathcal{M} \rrbracket$ ) can easily be achieved by fixing a special *undefined* output, allowing us to use the active learning algorithms for Mealy machines in LearnLib [16].

**Symbolic Runs.** Before defining timed semantics on concrete timed words, we define symbolic runs as an intermediate construct that we will also use when generating MM1Ts from concrete traces. We need one auxiliary concept that we define inductively over the complete sequence of transitions: the current symbolic timer value  $\theta_i$  at every transition. This has to be defined inductively as on some transitions a running timer is not reset. The initial timeout is  $\theta_0 = v_0 + t_0$ , for the start of the run at symbolic time  $v_0$ . Then, let

$$\theta_i = \begin{cases} \theta_{i-1} & \text{for } t_i = \perp \text{ (no reset)} \\ v_i + t_i & \text{otherwise (reset).} \end{cases}$$

The symbolic run is constructed from an untimed run of the form

$$q_0 \xrightarrow{i_1, o_1, t_1} q_1, \dots, q_{n-1} \xrightarrow{i_n, o_n, t_n} q_n$$

The symbolic run is then represented by a constrained symbolic timed word and constructed as follows. In every step, we define a short constrained symbolic timed word  $\langle \mathcal{S}_i, \varphi_i \rangle$  and concatenate these to form a constrained symbolic timed word for the whole sequence of transitions. For transition  $q_{j-1} \xrightarrow{i_j, o_j, t_j} q_j$  let

$$\langle \mathcal{S}_j, \varphi_j \rangle = \begin{cases} \langle (o_j, v_j), v_j = \theta_{j-1} \rangle & \text{if } i_j = \mathbf{timeout} \\ \langle (i_j, v_j), v_{j-1} < v_j < \theta_{j-1} \rangle & \text{else if } o_j = - \\ \langle (i_j, v_j) (o_j, v'_j), v_{j-1} < v_j = v'_j < \theta_{j-1} \rangle & \text{else.} \end{cases}$$

The complete symbolic run is then  $\langle \mathcal{S}_1, \varphi_1 \rangle \cdot \dots \cdot \langle \mathcal{S}_n, \varphi_n \rangle$ . Finally, every word in  $w \in \mathcal{I}_{\mathbf{to}}^*$  with  $\llbracket \mathcal{M} \rrbracket(w) \downarrow$  takes a unique sequence of transitions in  $\mathcal{M}$  and consequently has a unique symbolic run, denoted by  $\mathbf{sr}(w)$ . We extend  $\mathbf{sr}(\cdot)$  to sets of words by word-wise application and write  $\mathbf{sr}(\mathcal{M})$  for the the set of symbolic runs for words in the domain of  $\llbracket \mathcal{M} \rrbracket$ .

**Timed Semantics.** Our definition of the concrete timed semantics of  $\mathcal{M}$  leverages symbolic runs. The concrete timed semantics of  $\mathcal{M}$  then is the set  $\mathbf{traces}(\mathcal{M}) \subseteq T\Sigma^*$  of timed words over actions  $\Sigma = \mathcal{I} \cup \mathcal{O}$  such that

$$\rho \in \mathbf{traces}(\mathcal{M}) \Leftrightarrow \rho \models \langle \mathcal{S}, \varphi \rangle \text{ for some } \langle \mathcal{S}, \varphi \rangle \in \mathbf{sr}(\mathcal{M}).$$

*Example 3.* For the brick sorter, an untimed run could be

$$q_0 \xrightarrow{red, -, 3} q_1 \xrightarrow{black, -, \perp} q_2 \xrightarrow{timeout, pass, \infty} q_3$$

Then, assuming  $v_0 = 0$  and  $t_0 = \infty$  would lead to the symbolic run

$$\langle (red, v_1), 0 < v_1 < 0 + \infty \rangle \cdot \langle (black, v_2), v_1 < v_2 < v_1 + 3 \rangle \cdot \langle (pass, v_3), v_3 = v_1 + 3 \rangle$$

### 3 Trace Database

To learn an MM1T  $\mathcal{M}$ , we must find concrete timed words produced by the SUL that model constrained symbolic timed words in the symbolic runs of  $\mathcal{M}$ . However, we want to learn a model of a long-running black-box system to support more realistic logs. As such, we assume only a single timed word from which to learn<sup>3</sup>, and we do not assume that the word begins or ends in an initial configuration. This makes learning more difficult since we cannot simply iterate over the recorded words starting from the initial state. Instead, we construct a *trace database* that is instantiated with a query to find *initial conditions*. The sub-words that model the initial conditions mark the positions in the timed word where the SUL has been reset to its initial state.

A symbolic query is an extension of a constrained symbolic timed word that supports queries for unknown inputs or outputs, and that can be modeled by timed words with *stuttering actions*. A stuttering action might occur in a concrete word several times in a row, all of which may be matched by one action in a symbolic query. We require stuttering to learn models where the initial state has transitions leading back to itself. These self-loops are common in many controllers, including our traffic controller case study, where an output is periodically triggered by a timer without any new inputs. We now describe symbolic queries before explaining how they are used to configure initial conditions.

Formally, a symbolic query is a triple  $\langle \mathcal{S}, \varphi, \gamma \rangle$  where  $\mathcal{S}$  is a pair  $\langle \sigma, v \rangle$ ,  $\varphi$  is a constraint on the symbolic timestamps of  $\mathcal{S}$ , and  $\gamma \in \mathbb{B}^*$  is a sequence of Booleans the same length as  $\sigma$  where truth indicates that stuttering is allowed for the action at the same index. Here,  $\sigma$  is a finite word over the alphabet  $\Sigma \cup \{\top_{\mathcal{I}}, \top_{\mathcal{O}}\}$  and  $v$  is a series of symbolic timestamps the same length as  $\sigma$ . The special symbols  $\top_{\mathcal{I}}$  and  $\top_{\mathcal{O}}$  represent any *input* or *output* action, respectively. We use the convention that omitting  $\gamma$  when we write a symbolic query (i.e., writing  $\langle \mathcal{S}, \varphi \rangle$ ) means that  $\gamma$  is *false* for all actions in  $\mathcal{S}$ .

We now define when a concrete trace models a symbolic query. The difference between a symbolic query and a constrained symbolic timed word is that a symbolic query may include the symbolic actions  $\top_{\mathcal{I}}$  and  $\top_{\mathcal{O}}$  and may permit stuttering on actions. Given an action  $\sigma_i$  and a symbolic timestamp  $v_i$ , the function  $\mathbf{repeat}(\sigma_i, v_i) = \{(\sigma_i, v_{i,0}), \dots, (\sigma_i, v_{i,j}), \bigvee v_{i,j} = v_i \mid j \geq 0\}$  produces a set of all constrained symbolic timed words with  $\sigma_i$  repeated finitely many times and constraints that require one symbolic timestamp  $v_{i,j}$  to be equal to  $v_i$ .

**Definition 3 (Modeling of Symbolic Query).** *Given a concrete timed word  $\rho$  and a symbolic query  $\langle \mathcal{S}, \varphi, \gamma \rangle$  where  $\mathcal{S} = \langle \sigma, v \rangle$ ,  $\rho \models \langle \mathcal{S}, \varphi, \gamma \rangle$  when there exists a constrained symbolic timed word  $u$  such that  $\rho \models u$ , where*

$$u \in \left\{ \langle \mathcal{S}'_0, \varphi'_0 \rangle \cdots \langle \mathcal{S}'_n, \varphi'_n \rangle \mid \langle \mathcal{S}'_i, \varphi'_i \rangle = \begin{cases} \mathbf{repeat}(\sigma_i, v_i) & \text{if } \gamma_i \\ (\sigma_i, v_i), \varphi & \text{otherwise} \end{cases} \right\}$$

**Initial Conditions.** A trace database is a timed word instantiated with an initial condition that serves to break it into smaller pieces and a special query to

<sup>3</sup> Note that the approach also works with more than one word.

Table 1: Action statistics for our case studies. For the brick sorter, inputs are detected brick colors and outputs are sorting actions. For the intersection, inputs are lanes with detected cars and outputs are signalling configuration changes.

(a) Brick sorter		(b) Intersection		(c) Timing			
Input	Freq. [%]	Input	Freq. [%]	Case	Symbols	Time	Distance [s]
<i>red</i>	49.5	$A_2$	41.8	brick sorter inputs			$11.1 \pm 9.8$
<i>black</i>	50.5	$A_1$	39.2	brick sorter outputs			$13.2 \pm 9.8$
		$B_{\text{Left}}$	8.3	intersection inputs			$6.6 \pm 161.7$
Output	Freq. [%]	Output	Freq. [%]	intersection outputs			$10.0 \pm 0.0$
<i>pass</i>	49.5	$a_1 + a_2$	83.9				
<i>reject</i>	50.5	$b_1 + b_{\text{turn}}$	7.7				

prepend to requests from the learner. The initial conditions are specified using a symbolic query and each index of the word is tested against the query to see if it satisfies the condition. If the query is satisfied, then the index is marked as a starting index for a word in the database. Subsequently, when the learner submits a symbolic query, it has no information about the initial conditions of the trace database. As such, queries from the learner must be modified before testing against the trace database: they must be augmented with any additional actions matched by the initial condition.

Formally, a trace database is a triple  $\mathcal{D} = \langle \rho, \langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle, \langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle \rangle$  where  $\rho$  is a concrete timed word,  $\langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle$  is a symbolic query that defines initial conditions, and  $\langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle$  is a symbolic query that is prepended to trace database queries to match the initial conditions and define the beginning of a word. We require that  $\langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle$  defines the symbolic timestamp  $v_0$ , thereby providing a relative timestamp from which offsets may be computed.

We can now define a function *query* that computes a response to a symbolic query from an instantiated trace database. Note that more than one sub-word of the database may match a query and, in that case, one such matching sub-word will be chosen non-deterministically. In practice, the choice of word does not matter since any fulfill the constraints of the query.

**Definition 4.** Given a trace database  $\mathcal{D} = \langle \rho, \langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle, \langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle \rangle$  where  $\rho = (\sigma, \tau)$  and a symbolic query  $\langle \mathcal{S}, \varphi, \gamma \rangle$ , we define  $\text{query}(\mathcal{D}, \langle \mathcal{S}, \varphi, \gamma \rangle) = \rho_i, \dots, \rho_n$  where  $i \in \{j \mid \exists k > j. \rho_j, \dots, \rho_k \models \langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle\}$  and  $\rho_i, \dots, \rho_n \models \langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle \cdot \langle \mathcal{S}, \varphi, \gamma \rangle$ .

The initial conditions for a trace database are specific to each SUL and are a form of prior knowledge about which the learner has no information. However, in many cases the initial conditions can be safely assumed to be whatever happens when the SUL is fed no inputs for a long period of time. In this case, the only information provided by a human is how long to wait before the SUL can be assumed to have reset. This number need not be precise, only long enough that a reset occurs and short enough that the condition is met sufficiently often.

*Example 4.* Table 1 shows frequency and timing information for the most frequent inputs and outputs of our two case studies. Table 1a shows that bricks arrived



and either passed or were rejected a similar number of times. Table 1c shows that the mean times between brick-sorter inputs and outputs diverged by about 2s, but the standard deviation was the same. This tells us that the output timing is probably closely related to the input timing. We set initial conditions  $\langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle = \langle (\top_{\mathcal{O}}, v_0), (\top_{\mathcal{I}}, v_1), v_0 + 10 < v_1, (true, false) \rangle$  meaning that we search for any output (possibly stuttering) followed by any input after 10 ( $\sim 9.8$ ) seconds. We set the word beginnings with  $\langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle = \langle (\top_{\mathcal{O}}, v_0), true, (true) \rangle$ . Table 1c shows the timing of inputs and outputs for the intersection appear largely unrelated, with outputs occurring at a fixed interval, and Table 1b shows that one output action dominates the others. We set initial conditions  $\langle \mathcal{S}_I, \varphi_I, \gamma_I \rangle = \langle (a_1 + a_2, v_0), (\top_{\mathcal{I}}, v_1), v_0 + 10 < v_1, (true, false) \rangle$  meaning that we search for an  $a_1 + a_2$  output (possibly stuttering) followed by any input after 10 seconds. We set the word beginnings with  $\langle \mathcal{S}_P, \varphi_P, \gamma_P \rangle = \langle (a_1 + a_2, v_0), true, (true) \rangle$ .

## 4 From Concrete Traces to Symbolic Runs

We use symbolic queries to construct a *symbolic observation tree* from a trace database. A symbolic observation tree is a tree-shaped MM1T. For a given set of actions  $\Sigma = \mathcal{I} \cup \mathcal{O}$ , we generate input sequences  $w \in \mathcal{I}_{\mathbf{to}}^+$  and for these try to infer the corresponding symbolic runs of the target MM1T from which the trace database was recorded. The trace database provides a concrete timed word  $\rho = (\sigma, \tau)$  for the symbolic query  $\langle \mathcal{S}, \varphi, \gamma \rangle$  where we can use wildcards  $\top_{\mathcal{I}}$  and  $\top_{\mathcal{O}}$  to be matched by any input action and any output action, respectively. We do not use action stuttering when constructing the symbolic observation tree as this feature is needed only for specifying trace database initial conditions. As such, we omit  $\gamma$  when writing symbolic queries in this section. Intuitively, we mimic the inference process (i.e., interacting with the oracle) that is used for constructing an observation tree in [26]. However, while Vaandrager et al. can derive concrete timed queries for the symbolic relations and values they want to infer, we have to find adequate traces in the database, not having full control over timing. We leverage that, in general, a symbolic run is modeled by many timed words, most of which can be used interchangeably. One notable difference from an active learning setting is that the trace database may be incomplete. In this section, we focus on showing that the generated symbolic runs may be incomplete but will be consistent with all the information in the trace database. The quality of inferred models will depend on the quality of data in the database.

We initialize the symbolic observation tree  $\langle \mathcal{I}, \mathcal{O}, Q, q_0, \delta, \lambda, \kappa, t_0 \rangle$  with initial state  $q_0$ , i.e., initially  $Q = \{q_0\}$ , and use a timed symbolic query  $\langle (\top_{\mathcal{O}}, v_1), v_0 < v_1 \rangle$  to observe the initial timer  $\tau_1$  from timed word  $(o_1, \tau_1)$ , setting  $t_0 = \tau_1$ . Recall that  $v_0$  will be defined in the initial conditions of the trace database. If we cannot find a concrete trace in the database, we assume that no timer is set initially, setting  $t_0 = \infty$ . This is consistent with the database as in this case all traces in the database start with an input.

Now, assume the path from the root  $q_0$  of the tree leads to an unexplored state  $q$ , along already inferred transitions  $q_0 \xrightarrow{i_1, o_1, t_1} q_1 \dots q_{k-1} \xrightarrow{i_k, o_k, t_k} q_k = q$

(or empty sequence of transitions in the case of the initially unexplored state  $q_0$ ). Let  $\langle \mathcal{S}_q, \varphi_q \rangle$  denote the corresponding symbolic run. For  $q_0$ , we use symbolic run  $\langle \varepsilon, true \rangle$ , where  $\varepsilon$  denotes the empty word. The currently active symbolic timer (cf. Section 2) after the run is  $\theta_q = v_i + t_i$  for the most recent set timer, i.e., such that  $t_i \neq \perp$  and  $t_j = \perp$  if  $i < j$ . If the sequence of transitions is empty or no such  $\tau_i$  exists, then  $\theta_q = v_0 + t_0$ .

We generate a series of queries to the trace database and extend the symbolic tree based on the responses, adding new transitions from  $q$  to newly created states based on every input  $i \in \mathcal{I}_{\text{to}}$ . We distinguish two basic cases: transitions for regular inputs and transitions on timeouts.

**Timeouts.** For  $i = \text{timeout}$ , the symbolic tree only needs to be extended with a new state if a timer is currently running, i.e., if  $t_i \neq \infty$  in active timer  $\theta_q$ . In this case, we want to add new state  $r$  and new transition  $q \xrightarrow{\text{timeout}, o, t} r$  and need to compute  $o$  and  $t$ . In the best case, both values can be computed from symbolic query  $\langle \mathcal{S}_q, \varphi_q \rangle \cdot \langle (\top_{\mathcal{O}}, v_{k+1}), v_{k+1} = \theta_q \rangle \cdot \langle (\top_{\mathcal{O}}, v_{k+2}), v_{k+1} < v_{k+2} \rangle$ .

If a corresponding concrete timed word exists in the database, then it ends with  $\dots, (o_{k+1}, \tau_{k+1}), (o_{k+2}, \tau_{k+2})$ . The word immediately provides  $o = o_{k+1}$  and  $t = \tau_{k+2} - \tau_{k+1}$  is the time observed between the two subsequent timeouts. If no such word can be found, we can ask for the shorter  $\langle \mathcal{S}_q, \varphi_q \rangle \cdot \langle (\top_{\mathcal{O}}, v_{k+1}), v_{k+1} = \theta_q \rangle$ . A corresponding concrete timed word provides  $o$  and we assume that no new timer is set, i.e., that  $t = \infty$ . Here, we conflate the case that we do not have complete information with the case that no new timer is set. This is consistent with the trace database by the same argument given above: there can only be continuations with an input as the next action in the database.

In case the trace database also does not contain a concrete word for the second query, we do not add a new transition to the symbolic tree. Since we do not have enough information in the database for computing the transition, we stop exploring in this direction.

**Regular Inputs.** For  $i \neq \text{timeout}$ , we want to add new state  $r$  and new transition  $q \xrightarrow{i, o, t} r$  and need to observe  $o$  and infer  $t$ . This case is slightly more complex than timeouts since we have to account for the immediate output of the transition and the fact that input transitions can either continue the existing timer, reset it, or disable it. We start by asking symbolic query

$$\langle \mathcal{S}_q, \varphi_q \rangle \cdot \langle (i, v_{k+1}), v_{k+1} < \theta_q \rangle \cdot \langle (\top_{\mathcal{O}}, v'_{k+1}), v'_{k+1} = v_{k+1} \rangle \cdot \langle (\top_{\mathcal{O}}, v_{k+2}), v_{k+2} \neq \theta_q \rangle$$

which, answered with a timed word ending in  $\dots, (i, \tau_{k+1}), (o_{k+1}, \tau'_{k+1}), (o_{k+2}, \tau_{k+2})$ , provides enough information. We set  $o = o_{k+1}$  and  $t = \tau_{k+2} - \tau'_{k+1}$ . Since we observed a timeout that cannot be explained by the currently running timer (as  $v_{k+2} \neq \theta_q$ ), we can infer that the new transition sets a timer.

If no matching timed word is found and there is a currently running timeout, i.e., if  $t_i \neq \infty$  in  $\theta_q$ , we alter the query slightly to

$$\langle \mathcal{S}_q, \varphi_q \rangle \cdot \langle (i, v_{k+1}), v_{k+1} < \theta_q \rangle \cdot \langle (\top_{\mathcal{O}}, v'_{k+1}), v'_{k+1} = v_{k+1} \rangle \cdot \langle (\top_{\mathcal{O}}, v_{k+2}), v_{k+2} = \theta_q \rangle$$

and try to observe the already running timer expiring. If a corresponding timed word is found, we set  $o$  as before and  $t = \perp$ . This may actually be wrong: the transition we observe could have reset the timer to a value that (accidentally) equals the remaining time on the previously running timer. However, from the unsuccessful first query, we know that our choice is consistent with the database.

If the second query also does not produce a timed word, we try the shorter query  $\langle \mathcal{S}_q, \varphi_q \rangle \cdot \langle (i, v_{k+1}), v_{k+1} < \theta_q \rangle \cdot \langle (\top_{\mathcal{O}}, v'_{k+1}), v'_{k+1} = v_{k+1} \rangle$ . If successful, we observe  $o$  as above and assume  $t = \infty$ , which, again, conflates missing information and disabling the timer but is consistent with the information in the trace database.

In case all three queries fail, we do not add a transition or new state.

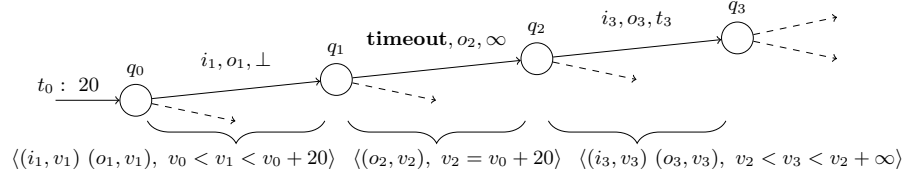


Fig. 3: Symbolic observation tree with symbolic run to  $q_3$ .

*Example 5.* Figure 3 shows a fragment of a symbolic observation tree and the corresponding symbolic run from the root  $q_0$  to inner node  $q_3$ : The initial timer is set to 20. The transition from  $q_0$  to  $q_1$  does not reset the timer, the transition from  $q_1$  to  $q_2$  disables the timer, and the transition from  $q_2$  to  $q_3$  sets a timer to  $t_3$ . The corresponding symbolic run consists of all observed inputs and outputs along the sequence of transitions and constrains symbolic times to obey the active initial timer that is triggered by the second transition.

**Consistency.** For a trace database that could have been generated by a MM1T, i.e., with consistent timer behavior, the symbolic observation tree is consistent with the trace database: we only stop extending the tree when no concrete continuations to traces exist and in every single step we ensure that the symbolic representation is consistent with the trace database.

## 5 Application in Model Learning Scenarios

To evaluate the utility of our proposed symbolic abstraction in different learning pipelines, we define five pipelines and execute them on symbolic observations generated from single logs for the brick-sorter model and the intersection controller.<sup>4</sup>

<sup>4</sup> A note on the experiment design: since the symbolic abstraction is not learned (i.e., does not extrapolate beyond certain knowledge), we do not evaluate its performance but focus on the utility in model learning. We fix adequate initial conditions for computing the trace databases.

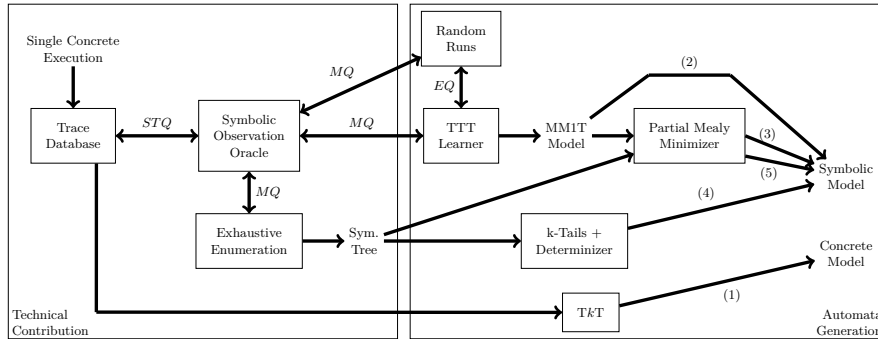


Fig. 4: The five evaluated learning pipelines.

We report on their quantitative performance and discuss the human-readability of the created models.

**Evaluation Setup.** To identify the best learning setup, we assembled five learning pipelines, illustrated in Fig. 4: timed  $k$ -tails ( $TkT$ ) without post-processing (1),  $TTT$  without post-processing (2),  $TTT$  with subsequent refinement (3), symbolic tree recovery with subsequent  $k$ -tails (4), and symbolic tree recovery with subsequent refinement (5). We use the following three approaches to generate an initial model:

The  $TkT$  (baseline) approach performs passive learning on the trace database without symbolic abstraction using a modified version of the  $TkT$  algorithm [22, 23]. We use a single non-resetting timer, no end events,  $k = 2$ , and a relaxed merge criterion (states are equal if one’s  $k$ -tails are a subset of the other’s).<sup>5</sup>

The  $TTT$  approach performs active learning of a Mealy machine using the novel symbolic abstraction oracle described in Section 4. We use the  $TTT$  [15] algorithm provided by LearnLib [16] and approximate equivalence queries with randomly generated runs with a fixed maximal length.

The *Symbolic Tree Recovery* queries the symbolic abstraction oracle exhaustively to recover a symbolic tree from the inputs.

We also implemented two post-processing steps that can be used in conjunction with the latter two learning approaches. The  $k$ -Tails approach performs an additional passive learning step on a set of symbolic runs by applying the  $k$ -tails passive learning algorithm [5]. Again, we use  $k = 2$  and the relaxed merge criterion described above and follow this with a determinization step. The *Partial Mealy Minimizer* post-processing step performs greedy, partition-refinement-inspired minimization on a partial model.

**Quantitative Evaluation.** We executed the pipelines described above on both the brick-sorter and intersection logs. For the intersection, we also consider sce-

<sup>5</sup> A problem with applying  $TkT$  to the intersection’s logs is that an unbounded number of inputs can occur before a relevant output (i.e., cars being detected before the signal switches). As a result, no  $k$  can be chosen that would avoid overfitting.

Table 2: Performance of our learning pipelines on the different scenarios.

Scenario	TTT						Symbolic Tree					
	Symb. Tree		TkT		No Post-pr.		Part. Ref.		$k$ -Tails		Part. Ref.	
	$ \mathcal{I} $	$d$	$ \ell $	$ \mathcal{Q} $	acc. [%]	$ \mathcal{Q} $	acc. [%]	$ \mathcal{Q} $	acc. [%]	$ \mathcal{Q} $	acc. [%]	$ \mathcal{Q} $
Brick Sorter	2	7	139	7	100	3	100	3	100	3	100	3
Intersection( $A_i, B_i$ )	4	6	581	n/a <sup>1</sup>	97	5	98	2	98	6	99	2
Intersection( $A_i, B_{Left}$ )	3	6	638	n/a <sup>1</sup>	99	2	99	2	100	4	86	19
Intersection( $A_i, B_i, B_{Left}$ )	5	6	970	n/a <sup>1</sup>	86	9	83	7	97	11	81	29
Intersection( $A_i, B_i, A_{Turn}$ )	8	6	1,351	n/a <sup>1</sup>	93	7	93	3	97	10	77	37
Intersection(complete)	11	6	1,839	749	81	13	81	13	94	23	77	50

<sup>1</sup> TkT cannot be used: projection of concrete trace to subset of inputs not obvious.

narios with a reduced input alphabet. E.g., “Intersection( $A_i, B_i$ )” only considers vehicles on straight lanes. TkT does not support such scenarios since it operates on the concrete log where it is not obvious how to project to a subset of inputs.

The results of the evaluation are shown in Table 2. For each scenario, the table provides some information about the symbolic tree: the number of inputs  $|\mathcal{I}|$ , the depth  $d$  to which the symbolic tree was explored and the number of resulting leaves  $|\ell|$  (i.e., unique symbolic traces). For each learning pipeline, the accuracy in the model learning step (acc. [%]), i.e., the percentage of symbolic runs in the recovered symbolic tree that is correctly represented by the final model, and size of the resulting automaton  $|\mathcal{Q}|$  w.r.t. the traces contained in the symbolic tree are shown in the order described in the last section. By design, TkT always yields perfect accuracy, so this information is omitted.

The accuracy of the TTT-based approach degrades faster than using  $k$ -tails for post-processing in experiments with more inputs and sparser logs. There is a trade-off between learning extra states, distinguished from other states by missing information and querying the trace database more extensively during equivalence queries. As the experiments show, greedy minimization cannot effectively mitigate missing information (either on the models inferred with TTT or on the symbolic tree directly): obtained models are often less accurate than the original models.

Summarizing, post-processing the symbolic tree yields the best results: the approach scales to the complete intersection and the automata are not too large, while preserving very high accuracy.

**Explainability.** We also examined the human-readability of the generated models. To judge human-readability, we rendered the learned models using GraphViz [11] and examined them manually.

Since TkT operates on log entries, input and output actions are independent in the automaton and timeouts have to be inferred from the timing intervals. The resulting edge labels are non-symbolic, e.g., `<pass_through, [17,32]>`, indicating that the system can be expected to let the brick pass in 17–32 time units. Additionally, the number of generated locations is far greater than the underlying model’s number of states, indicating that no semantic meaning can be assigned to the states. As a result, the model (available in the repeatability package [9]) is not easily comprehensible for a human reader.

In contrast, the workflows using the symbolic tree fully recover the correct automaton shown in Figure 2. The symbolic tree generation yields MM1T transitions that combine input, output and timing behavior (e.g., `Timeout`, `pass_through` // `timer off`). One can comprehend the system’s behavior “at a glance”, e.g., the bug in the brick sorter’s behavior can be seen in the generated model by following the execution path for insertion of two different-colored bricks.

These results extend to the intersection scenarios. One can see how the intersection controller behaves by inspection of the learned model: the signals change to accommodate arriving cars on blocked lanes. While the automata generated, e.g., by symbolic tree recovery with  $k$ -tails refinement do increase in size (up to 23 states) when the input alphabet is enlarged, their comprehensibility surpasses the 749-state automaton generated by TkT.

**Threats to Validity.** On the conceptual level, we assume that our scenarios can a) be correctly modeled using MM1Ts and b) initial conditions can be identified via queries. Vaandrager et al. [26] argue that MM1Ts are applicable to many real-life scenarios and we have anecdotally found that the quality of final learned models is not very sensitive to the precision of initial conditions. The primary internal threat is the parameterization of our  $k$ -tails, especially the choice of  $k = 2$ . We selected that value based on its frequent selection in the literature for similar use cases, e.g. in [6, 7, 10]. External validity may be threatened by our approach overfitting the two case studies. We designed our method to be as general as possible, and selected two dissimilar case studies to mitigate this risk.

## 6 Related Work

Many passive timed model learning methods construct and minimize trees. Pastore et al. [22, 23] proposed the TkT algorithm for inference of timed automata with multiple clocks. It normalizes traces, turns them into Timed Automata (TAs) trees and merges locations to gain a general structure. Verwer et al. [28] proposed the RTI algorithm for learning deterministic real-time automata. It forms an “augmented” prefix tree with accepting/rejecting states, and merges and splits them to compress the trees into automata. Maier et al. [19] learned TAs online by constructing a prefix tree automaton and merging its states. Unlike RTI, their method does not require negative examples. Recently, Coranguer et al. [8] constructed tree-shaped automata, merged states (ignoring timing constraints) and then used timing information to split states. They presented promising results in comparison to TkT (in some respects) and RTI. Grinchtein et al. [12] learned event-recording TAs. They built timed decision trees, and then folded them into a compact automaton. Henry et al. [13] also proposed a method for learning event-recording TAs where not all transitions must reset clocks. Dematos et al. [20] presented a method and proof of concept for learning stochastic TAs by identifying an equivalence relation between states and merging them.

Other formalism used in passive approaches are as follows: Narayan et al. introduced a method to mine TAs using patterns expressed as Timed Regular Expressions (TREs) [21]. The technique passively mines variable bindings from

system traces for TREs templates provided by a user. Verwer et al. [27] present an algorithm for the identification of deterministic one-clock TAs. The algorithm is efficient in the limit, i.e., it requires polynomial time to identify the learned model. Tappler et al. [24] used genetic algorithms for learning TAs based on passive traces. Later, the approach was adapted to the active setting [2]. An et al. [3] proposed two methods for actively learning TAs. In one the learner needs to guess which transitions carry clock resets, and in the other the teacher has a method of telling if clock-resets occur. Recently, Tappler et al. [25] proposed a learning method for TAs that is based on SMT-solving. SMT-solving can handle cases with partial information (as experienced in the traffic controller), and still provide solutions satisfying the given constraints. However, the long traces we are dealing with will likely introduce too many variables and formulas to scale.

Aichernig et al. [1] compared active and passive learning approaches in a network protocol setting. They show that passive learning is competitive when utilizing sparse data, a result matching our observations when comparing TTT to passive learning-based symbolic tree post-processing.

A recent work by Vaandrager et al. defined MM1T and then used an adapter interface to actively mine them using existing Mealy machine mining algorithms in LearnLib [26]. This technique is used as one of the post-processing steps in our approach, and we use an equivalent definition of MM1Ts. We complement this approach via our symbolic observation oracle.

Jeppu et al. [18] recently introduced a method to construct automata from long traces by extracting counterexamples from attempting to prove that no such model exists. It found smaller models than traditional state-merging methods.

## 7 Conclusion

We presented a novel technique for abstracting a single concrete log of a timed system into a MM1T. The abstraction can be used as an oracle by active learning algorithms following the MAT framework such as TTT or to create a symbolic tree view of the system. We evaluate four approaches for learning a model of the system based on this abstraction in combination with different post-processing methods on two real-world-derived use cases, a brick sorting system and a traffic intersection signaling controller. We examined if our approach can be used to provide explainability for complex or machine learned black-box systems. We found that the proposed symbolic trees in combination with post-processing via  $k$ -tails yields concise and symbolic human-readable automata.

We plan to apply our approach to more use cases to verify its performance in more scenarios and on different automata classes (e.g., automata with data). Moreover, we can not yet formally relate the quality of our models to the input’s completeness and aim at finding such a relation in future work.

**Acknowledgements.** This work was supported by the S40S Villum Investigator Grant (37819) from VILLUM FONDEN, the ERC Advanced Grant LASSO, DIREC, and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 495857894 (STING).

## References

1. Aichernig, B.K., Muškardin, E., Pferscher, A.: Active vs. passive: A comparison of automata learning paradigms for network protocols. *Electronic Proceedings in Theoretical Computer Science* **371**, 1–19 (Sep 2022). <https://doi.org/10.4204/eptcs.371.1>, FMAS/ASYDE 2022
2. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: Learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) *NASA Formal Methods*. LNCS, vol. 12229, pp. 1–19. Springer International Publishing (2020). [https://doi.org/10.1007/978-3-030-55754-6\\_1](https://doi.org/10.1007/978-3-030-55754-6_1), NFM 2020
3. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'20)*. LNTCS, vol. 12078, pp. 444–462. Springer (2020). [https://doi.org/10.1007/978-3-030-45190-5\\_25](https://doi.org/10.1007/978-3-030-45190-5_25)
4. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
5. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers* **C-21**(6), 592–597 (Jun 1972). <https://doi.org/10.1109/TC.1972.5009015>
6. Busany, N., Maoz, S., Yulazari, Y.: Size and accuracy in model inference. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. pp. 887–898 (Nov 2019). <https://doi.org/10.1109/ASE.2019.00087>, ASE 2019
7. Cohen, H., Maoz, S.: The confidence in our k-tails. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. pp. 605–610. Association for Computing Machinery, New York, NY, USA (Sep 2014). <https://doi.org/10.1145/2642937.2642944>, ASE '14
8. Cornanguer, L., Largouët, C., Rozé, L., Termier, A.: TAG: Learning timed automata from logs. *Proceedings of the AAAI Conference on Artificial Intelligence* **36**(4), 3949–3958 (June 2022). <https://doi.org/10.1609/aaai.v36i4.20311>
9. Dierl, S., Howar, F.M., Kauffman, S., Kristjansen, M., Larsen, K.G., Lorber, F., Mauritz, M.: Learning symbolic timed models from concrete timed data – data and replication package (Mar 2023). <https://doi.org/10.5281/zenodo.7766789>
10. Gabor, U.T., Dierl, S., Spinczyk, O.: Spectrum-based fault localization in deployed embedded systems with driver interaction models. In: Romanovsky, A., Troubitsyna, E., Bitsch, F. (eds.) *Computer Safety, Reliability, and Security*. Lecture Notes in Computer Science, vol. 11698, pp. 97–112. Springer International Publishing, Cham (2019). [https://doi.org/10.1007/978-3-030-26601-1\\_7](https://doi.org/10.1007/978-3-030-26601-1_7), SAFECOMP 2019
11. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. *Software: Practice and Experience* **30**(11), 1203–1233 (2000). [https://doi.org/10.1002/1097-024X\(200009\)30:11<1203::AID-SPE338>3.0.CO;2-N](https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N)
12. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) *CONCUR 2006 – Concurrency Theory*. Lecture Notes in Computer Science, vol. 4137, pp. 435–449. Springer, Berlin, Heidelberg (2006). [https://doi.org/10.1007/11817949\\_29](https://doi.org/10.1007/11817949_29), CONCUR 2006
13. Henry, L., Jéron, T., Markey, N.: Active learning of timed automata with unobservable resets. In: *Formal Modeling and Analysis of Timed Systems (FOR-*



- MATS'20). LNTCS, vol. 12288, pp. 144–160. Springer (2020). [https://doi.org/10.1007/978-3-030-57628-8\\_9](https://doi.org/10.1007/978-3-030-57628-8_9)
14. Howar, F., Steffen, B.: Active automata learning in practice. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, Lecture Notes in Computer Science, vol. 11026, pp. 123–148. Springer International Publishing, Cham (2018). [https://doi.org/10.1007/978-3-319-96562-8\\_5](https://doi.org/10.1007/978-3-319-96562-8_5)
  15. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) *Runtime Verification*. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer International Publishing, Cham (2014). [https://doi.org/10.1007/978-3-319-11164-3\\_26](https://doi.org/10.1007/978-3-319-11164-3_26), RV 2014
  16. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) *Computer Aided Verification*. Lecture Notes in Computer Science, vol. 9206, pp. 487–495. Springer International Publishing, Cham (2015). [https://doi.org/10.1007/978-3-319-21690-4\\_32](https://doi.org/10.1007/978-3-319-21690-4_32), CAV 2015
  17. Iversen, T.K., Kristoffersen, K.J., Larsen, K.G., Laursen, M., Madsen, R.G., Mortensen, S.K., Pettersson, P., Thomasen, C.B.: Model-checking real-time control programs: verifying lego mindstorms tm systems using uppaal. In: *Proceedings 12th Euromicro Conference on Real-Time Systems*. Euromicro RTS 2000. pp. 147–155. IEEE (2000)
  18. Jeppu, N.Y., Melham, T., Kroening, D., O’Leary, J.: Learning concise models from long execution traces. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. pp. 1–6 (2020). <https://doi.org/10.1109/DAC18072.2020.9218613>
  19. Maier, A.: Online passive learning of timed automata for cyber-physical production systems. In: *IEEE International Conference on Industrial Informatics (INDIN’14)*. pp. 60–66. IEEE (2014). <https://doi.org/10.1109/INDIN.2014.6945484>
  20. de Matos Pedro, A., Crocker, P.A., de Sousa, S.M.: Learning stochastic timed automata from sample executions. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLA’12)*. LNTCS, vol. 7609, pp. 508–523. Springer (2012). [https://doi.org/10.1007/978-3-642-34026-0\\_38](https://doi.org/10.1007/978-3-642-34026-0_38)
  21. Narayan, A., Cutulenco, G., Joshi, Y., Fischmeister, S.: Mining timed regular specifications from system traces. *ACM Trans. Embed. Comput. Syst.* **17**(2), 46:1–46:21 (1 2018). <https://doi.org/10.1145/3147660>
  22. Pastore, F., Micucci, D., Guzman, M., Mariani, L.: Tkt: Automatic inference of timed and extended pushdown automata. *IEEE Transactions on Software Engineering* **48**(2), 617–636 (Feb 2022). <https://doi.org/10.1109/TSE.2020.2998527>
  23. Pastore, F., Micucci, D., Mariani, L.: Timed k-tail: Automatic inference of timed automata. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. pp. 401–411. IEEE, New York (Mar 2017). <https://doi.org/10.1109/ICST.2017.43>, ICST 2017
  24. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Time to learn – learning timed automata from tests. In: *Formal Modeling and Analysis of Timed Systems*. pp. 216–235. Springer (2019). [https://doi.org/10.1007/978-3-030-29662-9\\_13](https://doi.org/10.1007/978-3-030-29662-9_13)
  25. Tappler, M., Aichernig, B.K., Lorber, F.: Timed automata learning via SMT solving. In: *NASA Formal Methods*. LNCS, vol. 13260, pp. 489–507. Springer (2022). [https://doi.org/10.1007/978-3-031-06773-0\\_26](https://doi.org/10.1007/978-3-031-06773-0_26)
  26. Vaandrager, F., Bloem, R., Ebrahimi, M.: Learning mealy machines with one timer. In: *Language and Automata Theory and Applications*. pp. 157–170. No. 12638 in LNCS, Springer (2021). [https://doi.org/10.1007/978-3-030-68195-1\\_13](https://doi.org/10.1007/978-3-030-68195-1_13)

27. Verwer, S., de Weerd, M., Witteveen, C.: One-clock deterministic timed automata are efficiently identifiable in the limit. In: Language and Automata Theory and Applications. LNTCS, vol. 5457, pp. 740–751. Springer (2009). [https://doi.org/10.1007/978-3-642-00982-2\\_63](https://doi.org/10.1007/978-3-642-00982-2_63)
28. Verwer, S., de Weerd, M., Witteveen, C.: Efficiently identifying deterministic real-time automata from labeled data. Machine Learning **86**(3), 295–333 (Mar 2012). <https://doi.org/10.1007/s10994-011-5265-4>