Automata with Timers*

Véronique Bruyère[1], Guillermo A. Pérez[2], Gaëtan Staquet[1,2], and Frits W. Vaandrager[3]

[1] University of Mons, Belgium
{veronique.bruyere,gaetan.staquet}@umons.ac.be
[2] University of Antwerp – Flanders Make, Belgium
guillermo.perez@uantwerpen.be
[3] Radboud University, The Netherlands
f.vaandrager@cs.ru.nl

Abstract. In this work, we study properties of deterministic finite-state automata with timers, a subclass of timed automata proposed by Vaandrager et al. as a candidate for an efficiently learnable timed model. We first study the complexity of the configuration reachability problem for such automata and establish that it is PSPACE-complete. Then, as simultaneous timeouts (we call these, races) can occur in timed runs of such automata, we study the problem of determining whether it is possible to modify the delays between the actions in a run, in a way to avoid such races. The absence of races is important for modelling purposes and to streamline learning of automata with timers. We provide an effective characterization of when an automaton is race-avoiding and establish that the related decision problem is in 3EXP and PSPACE-hard.

Keywords: Timed systems · model checking · reachability

1 Introduction

Timed automata were introduced by Alur & Dill [4] as finite-state automata equipped with real-valued clock variables for measuring the time between state transitions. These clock variables all increase at the same rate when time elapses, they can be reset along transitions, and be used in guards along transitions and in invariant predicates for states. Timed automata have become a framework of choice for modeling and analysis of real-time systems, equipped with a rich theory, supported by powerful tools, and with numerous applications [8].

Interestingly, whereas the values of clocks in a timed automaton *increase* over time, designers of real-time systems (e.g. embedded controllers and network protocols) typically use timers to enforce timing constraints, and the values of these timers *decrease* over time. If an application starts a timer with a certain

* This work was supported by the Belgian FWO "SAILor" project (G030020N). Gaëtan Staquet is a research fellow (Aspirant) of the Belgian F.R.S.-FNRS. The research of Frits Vaandrager was supported by NWO TOP project 612.001.852 "Greybox learning of Interfaces for Refactoring Legacy Software (GIRLS)".

value t, then this value decreases over time and after t time units — when the value has become 0 — a timeout event occurs. It is straightforward to encode the behavior of timers using a timed automaton. Timed automata allow one to express a richer class of behaviors than what can be described using timers, and can for instance express that the time between two events is contained in an interval $[t - d, t + d]$. Moreover, timed automata can express constraints on the timing between arbitrary events, not just between start and timeout of timers.

However, the expressive power of timed automata entails certain problems. For instance, one can easily define timed automata models in which time stops at some point (timelocks) or an infinite number of discrete transitions occurs in a finite time (Zeno behavior). Thus timed automata may describe behavior that cannot be realized by any physical system. Also, learning [6, 15] of timed automata models in a black-box setting turns out to be challenging [5, 13, 14]. For a learner who can only observe the external events of a system and their timing, it may be really difficult to infer the logical predicates (invariants and guards) that label the states and transitions of a timed automaton model of this system. As a result, all known learning algorithms for timed automata suffer from combinatorial explosions, which severely limits their practical usefulness.

For these reasons, it is interesting to consider variations of timed automata whose expressivity is restricted by using timers instead of clocks, as introduced by Dill in [11]. In an automaton with timers (AT), a transition may start a timer by setting it to a certain value. Whenever a timer reaches zero, it produces an observable timeout symbol that triggers a transition in the automaton. Dill also shows that the space of timer valuations can be abstracted into a finite number of so-called *regions*. However, the model we study here is slightly different, as, unlike Dill, we allow a timer to be stopped before reaching zero. We also study the regions of our AT and give an upper bound on their number.

Vaandrager et al. [18] provide a black-box active learning algorithm for automaton with a single timer, and evaluate it on a number of realistic applications, showing that it outperforms the timed automata based approaches of Aichernig et al. [2] and An et al. [5]. However, whereas [18] only support a single timer, the genetic programming approach of [2] is able to learn models with multiple clocks/timers. If we want to extend the learning algorithm of [18] to a setting with multiple timers, we need to deal with the issue of *races*, i.e., situations where multiple timers reach zero (and thus timeout) simultaneously. If a race occurs, then (despite the automaton being deterministic!) the automaton can process the simultaneous timeouts in various orders, leading to nondeterministic behavior. This means that during learning of an automaton with multiple timers, a learner needs to offer the inputs at specific times in order to avoid the occurrence of races. As long as there are no races, the behavior of the automaton will be deterministic, and a learner may determine, for each timeout, by which preceding input it was caused by slightly *wiggling* the timing of inputs and check whether the timing timeout changes in a corresponding manner.

Contribution In this work, we take the one-timer definition from [18] and extend it to multiple timers while — to avoid overcomplicating the model — keeping the restriction that every transition can start or restart at most one timer. We first study the complexity of the configuration reachability problem for this model and establish that it is PSPACE-complete. Then, we turn our attention to the problem of determining whether it is possible to wiggle the delays between the inputs in a run, in a way to avoid races. The importance of the latter is twofold. First, automata with timers may not be an attractive modelling formalism in the presence of behaviors that do not align with those of the real-world systems they are meant to abstract. Second, the absence of races is a key property used in the learning algorithm for automata with a single timer. In this direction, we provide an effective characterization of when an automaton is race-avoiding and establish that the related decision problem is in 3EXP and PSPACE-hard. In a more pragmatic direction, while again leveraging our characterization, we show that with fixed input and timer sets, the problem is in PSPACE. Finally, we also give some simple yet sufficient conditions for an automaton to be race-avoiding. We refer to [9] for an extended version of our paper with all the proofs.

2 Preliminaries

An *automaton with timers* uses a finite set X of *timers*. Intuitively, a timer can be *started* to any integer value to become *active*. Subsequently, its value is decremented as time elapses (i.e., at the same fixed rate for all timers). When the value of a timer reaches 0, it *times out* and it is no longer active. Active timers can also be *stopped*, rendering them inactive, too. Such an automaton, along any transition, can stop a number of timers and update a *single* timer.

Some definitions are in order. We write $TO[X]$ for the set $\{to[x] \mid x \in X\}$ of *timeouts of* X. We denote by I a finite set of *inputs*, and by $\hat{I}$ the set $I \cup TO[X]$ of *actions*: either reading an input (an *input-action*), or processing a timeout (a *timeout-action*). Finally, $U = (X \times \mathbb{N}^{>0}) \cup \{\bot\}$ is the set of *updates*, where (x, c) means that timer x is started with value c, and $\bot$ stands for no timer update.

Definition 1 (Automaton with timers). *An automaton with timers (AT, for short) is a tuple $\mathcal{A} = (X, I, Q, q_0, \chi, \delta)$ where:*

- *X is a finite set of timers, I a finite set of inputs,*
- *Q is a finite set of states, with $q_0 \in Q$ the initial state,*
- *$\chi\colon Q \to \mathcal{P}(X)$, with $\chi(q_0) = \emptyset$, is a total function that assigns a finite set of active timers to each state,*
- *$\delta\colon Q \times \hat{I} \to Q \times U$ is a partial transition function that assigns a state and an update to each state-action pair, such that*
  - *$\delta(q, i)$ is defined iff either $i \in I$ or there is a timer $x \in \chi(q)$ with $i = to[x]$,*
  - *if $\delta(q, i) = (q', u)$ with $i = to[x]$ and $u = (y, c)$, then $x = y$ (when processing a timeout $to[x]$, only the timer x can be restarted).*

Moreover, any transition t of the form $\delta(q, i) = (q', u)$ must be such that

- *if $u = \bot$, then $\chi(q') \subseteq \chi(q)$ (all timers active in q' were already active in q in case of no timer update); moreover, if $i = to[x]$, then $x \notin \chi(q')$ (when the timer x times out and is not restarted, then x becomes inactive in q');*

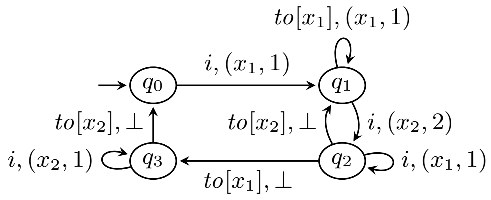


Fig. 1: An automaton with two timers x_1, x_2, such that $\chi(q_0) = \emptyset$, $\chi(q_1) = \{x_1\}$, $\chi(q_2) = \{x_1, x_2\}$, and $\chi(q_3) = \{x_2\}$.

– *if* $u = (x, c)$*, then* $x \in \chi(q')$ *and* $\chi(q') \setminus \{x\} \subseteq \chi(q)$ *((re)starting the timer* x *makes it active in* q'*).*

When a timer x *is active in* q *and* $i \neq to[x]$*, we say that the transition* t stops x *if* x *is inactive in* q'*, and that* t discards x *if* t *stops* x *or restarts* x*. We write* $q \xrightarrow[u]{i} q'$ *if* $\delta(q, i) = (q', u)$*.*

Example 1. An AT $\mathcal{A}$ is shown in Figure 1 with set $X = \{x_1, x_2\}$ of timers and with set $I = \{i\}$ of inputs. In the initial state q_0, no timer is active, while x_1 is active in q_1 and q_2, and x_2 is active in q_2 and q_3. That is, $\chi(q_0) = \emptyset, \chi(q_1) = \{x_1\}, \chi(q_2) = \{x_1, x_2\}$, and $\chi(q_3) = \{x_2\}$. Timer updates are shown in the transitions. For instance, x_1 is started with value 1 when going from q_0 to q_1. The transition looping on q_2 discards x_1 and restarts it with value 1.

2.1 Timed semantics

The semantics of an AT $\mathcal{A}$ is defined via an infinite-state labeled transition system that describes all possible configurations and transitions between them.

A *valuation* is a partial function $\kappa \colon X \to \mathbb{R}^{\geq 0}$ that assigns nonnegative real numbers to timers. For $Y \subseteq X$, we write $\mathsf{Val}(Y)$ for the set of all valuations κ such that $\mathsf{dom}(\kappa) = Y$.[4] A *configuration* of $\mathcal{A}$ is a pair (q, κ) where $q \in Q$ and $\kappa \in \mathsf{Val}(\chi(q))$. The *initial configuration* is the pair (q_0, κ_0) where κ_0 is the empty valuation since $\chi(q_0) = \emptyset$. If $\kappa \in \mathsf{Val}(Y)$ is a valuation in which all timers from Y have a value of at least $d \in \mathbb{R}^{\geq 0}$, then d units of time may elapse. We write $\kappa - d \in \mathsf{Val}(Y)$ for the valuation that satisfies $(\kappa - d)(x) = \kappa(x) - d$, for all $x \in Y$. The following rules specify the transitions between configurations $(q, \kappa), (q', \kappa')$.

$$\frac{\forall x \colon \kappa(x) \geq d}{(q, \kappa) \xrightarrow{d} (q, \kappa - d)} \tag{1}$$

$$\frac{q \xrightarrow[\perp]{i} q', \quad i = to[x] \Rightarrow \kappa(x) = 0, \quad \forall y \in \chi(q') \colon \kappa'(y) = \kappa(y)}{(q, \kappa) \xrightarrow[\perp]{i} (q', \kappa')} \tag{2}$$

[4] Notation $\mathsf{dom}(f)$ means the domain of the partial function f.

$$\frac{q \xrightarrow[(x,c)]{i} q', \quad i = \mathit{to}[x] \Rightarrow \kappa(x) = 0, \quad \forall y \in \chi(q')\colon \kappa'(y) = \begin{cases} c & \text{if } y = x \\ \kappa(y) & \text{otherwise} \end{cases}}{(q, \kappa) \xrightarrow[(x,c)]{i} (q', \kappa')} \tag{3}$$

Transitions of type (1) are called *delay transitions* (delay zero is allowed); those of type (2) and (3) are called *discrete transitions* (*timeout transitions* when $i = \mathit{to}[x]$ and *input transitions* otherwise). A *timed run* of $\mathcal{A}$ is a sequence of configurations such that delay and discrete transitions alternate, ending with a delay transition. The set $\mathit{truns}(\mathcal{A})$ of timed runs is defined inductively as follows.

- The sequence $(q_0, \kappa_0) \xrightarrow{d} (q_0, \kappa_0 - d)$ is in $\mathit{truns}(\mathcal{A})$.
- Suppose $\rho(q, \kappa)$ is a timed run ending with configuration (q, κ), then $\rho' = \rho(q, \kappa) \xrightarrow[u]{i} (q', \kappa') \xrightarrow{d} (q', \kappa' - d)$ is in $\mathit{truns}(\mathcal{A})$.

A timed run is also written as $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1\ \ldots\ d_n\ i_n/u_n\ d_{n+1}\ (q, \kappa)$ such that only the initial configuration (q_0, κ_0) and the last configurations (q, κ) of ρ are given. The *untimed trace* of a timed run ρ, denoted $\mathit{untime}(\rho)$, is the alternating sequence of states and actions from ρ, that is, $\mathit{untime}(\rho) = q_0\ i_1\ \ldots\ i_n\ q$ (we omit the valuations, the delays, and the updates).

Example 2. A sample timed run ρ of the AT of Example 1 is given below. Notice the transition with delay zero, indicating that two actions occur at the same time.

$$\rho = (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0) \xrightarrow[(x_2,2)]{i} (q_2, x_1 = 0, x_2 = 2)$$
$$\xrightarrow{0} (q_2, x_1 = 0, x_2 = 2) \xrightarrow[\perp]{\mathit{to}[x_1]} (q_3, x_2 = 2) \xrightarrow{2} (q_3, x_2 = 0) \xrightarrow[\perp]{\mathit{to}[x_2]} (q_0, \emptyset) \xrightarrow{0.5} (q_0, \emptyset).$$

The untimed trace of ρ is $\mathit{untime}(\rho) = q_0\ i\ q_1\ i\ q_2\ \mathit{to}[x_1]\ q_3\ \mathit{to}[x_2]\ q_0$.

2.2 Blocks and races

In this section, given an AT $\mathcal{A}$, we focus on its timed runs $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1 \cdots d_n\ i_n/u_n\ d_{n+1}\ (q, \kappa)$ such that their first and last delays are non-zero and no timer times out in their last configuration, i.e., $d_1 > 0, d_{n+1} > 0$ and $\kappa(x) \neq 0$ for all $x \in \chi(q)$.[5] Such runs are called *padded*, and we denote by $\mathit{ptruns}(\mathcal{A})$ the set of all padded timed runs of $\mathcal{A}$. To have a good intuition about padded timed runs, their decomposition into *blocks* is helpful and will be often used in the proofs. A block is composed of an input i that starts a timer x and of the succession of timeouts and restarts of x, that i induces inside a timed run. Let us formalize this notion. Consider a padded timed run $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1\ \ldots\ d_n\ i_n/u_n\ d_{n+1}\ (q, \kappa)$ of an AT. Let k, k' be such that $1 \leq k < k' \leq n$. We say that i_k *triggers* $i_{k'}$ if there is a timer x such that:

[5] The reason for this choice will be clarified at the end of this section.

– i_k (re)starts x, that is, $u_k = (x, c)$,
– $i_{k'}$ is the action $to[x]$, and
– there is no ℓ with $k < \ell < k'$ such that $i_\ell = to[x]$ or i_ℓ discards x.

Note that $i_{k'}$ may restart x or not, and if it does, x later times out or is discarded.

Definition 2 (Block). *Let $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1\ \dots\ d_n\ i_n/u_n\ d_{n+1}\ (q, \kappa)$ be a padded timed run of an AT. A* block *of ρ is a pair $B = (k_1 k_2 \dots k_m, \gamma)$ such that $i_{k_1}, i_{k_2}, \dots, i_{k_m}$ is a maximal subsequence of actions of ρ such that $i_{k_1} \in I$, i_{k_ℓ} triggers $i_{k_{\ell+1}}$ for all $1 \leq \ell < m$, and γ is the* timer fate *of B defined as:*

$$\gamma = \begin{cases} \bot & \text{if } i_{k_m} \text{ does not restart any timer,} \\ \bullet & \text{if } i_{k_m} \text{ restarts a timer which is discarded by some } i_\ell, \text{ with } k_m < \ell \leq n, \\ & \text{when its value is zero,} \\ \times & \text{otherwise.} \end{cases}$$

In the timer fate definition, consider the case where i_{k_m} restarts a timer x. For the purposes of Section 4.1, it is convenient to know whether x is later discarded or not, and in case it is discarded, whether this occurs when its value is zero ($\gamma = \bullet$). Hence, $\gamma = \times$ covers both situations where x is discarded with non-zero value, and x is still active in the last configuration (q, κ) of ρ. Notice that in the latter case, x has also non-zero value in (q, κ) as ρ is padded. When no confusion is possible, we denote a block by a sequence of inputs rather than the corresponding sequence of indices, that is, $B = (i_{k_1} i_{k_2} \dots i_{k_m}, \gamma)$. In the sequel, we use notation $i \in B$ to denote an action i belonging to the sequence of B.

By definition of an AT, recall that the same timer x is restarted along a block B. Hence we also say that B is an *x-block*. Note also that the sequence of a block can be composed of a single input $i \in I$.

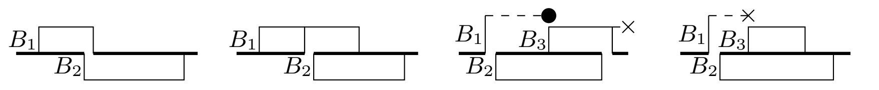
As this notion of blocks is not trivial but plays a great role in this paper, let us give several examples illustrating multiple situations.

Example 3. Consider the timed run ρ of Example 2 from the AT $\mathcal{A}$ depicted in Figure 1. It has two blocks: an x_1-block $B_1 = (i\ to[x_1], \bot)$ and an x_2-block $B_2 = (i\ to[x_2], \bot)$, both represented in Figure 2a.[6] In this visual representation of the blocks, time flows left to right and is represented by the thick horizontal line. A "gap" in that line indicates that the time is stopped, i.e., the delay between two consecutive actions is zero. We draw a vertical line for each action, and join together actions belonging to a block by a horizontal (non-thick) line.

Consider another timed run σ from $\mathcal{A}$:

$$\begin{aligned} \sigma = (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{1} (q_1, x_1 = 0) \xrightarrow[(x_1,1)]{to[x_1]} (q_1, x_1 = 1) \\ \xrightarrow{0} (q_1, x_1 = 1) \xrightarrow[(x_2,2)]{i} (q_2, x_1 = 1, x_2 = 2) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1) \\ \xrightarrow[\bot]{to[x_1]} (q_3, x_2 = 1) \xrightarrow{1} (q_3, x_2 = 0) \xrightarrow[\bot]{to[x_2]} (q_0, \emptyset) \xrightarrow{0.5} (q_0, \emptyset). \end{aligned}$$

[6] When using the action indices in the blocks, we have $B_1 = (1\,3, \bot)$ and $B_2 = (2\,4, \bot)$.



(a) Timed run ρ. (b) Timed run σ. (c) Timed run π. (d) Timed run τ.

Fig. 2: Block representations of four timed runs.

This timed run has also two blocks represented in Figure 2b, such that $B_1 = (i\ \mathit{to}[x_1]\ \mathit{to}[x_1], \bot)$ with x_1 timing out twice.

We conclude this example with two other timed runs, π and τ, such that some of their blocks have a timer fate $\gamma \neq \bot$. Let π and τ be the timed runs:

$$\begin{aligned}
\pi = {} & (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1) \xrightarrow[(x_2,2)]{i} (q_2, x_1 = 1, x_2 = 2) \\
& \xrightarrow{1} (q_2, x_1 = 0, x_2 = 1) \xrightarrow[(x_1,1)]{i} (q_2, x_1 = 1, x_2 = 1) \xrightarrow{1} (q_2, x_1 = 0, x_2 = 0) \\
& \xrightarrow[\bot]{\mathit{to}[x_2]} (q_1, x_1 = 0) \xrightarrow{0} (q_1, x_1 = 0) \xrightarrow[(x_1,1)]{\mathit{to}[x_1]} (q_1, x_1 = 1) \xrightarrow{0.5} (q_1, x_1 = 0.5) \\
\tau = {} & (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{0} (q_1, x_1 = 1) \xrightarrow[(x_2,2)]{i} (q_2, x_1 = 1, x_2 = 2) \\
& \xrightarrow{0.5} (q_2, x_1{=}0.5, x_2{=}1.5) \xrightarrow[(x_1,1)]{i} (q_2, x_1{=}1, x_2{=}1.5) \xrightarrow{1} (q_2, x_1{=}0, x_2{=}0.5) \\
& \xrightarrow[\bot]{\mathit{to}[x_1]} (q_3, x_2 = 0.5) \xrightarrow{0.5} (q_3, x_2 = 0) \xrightarrow[\bot]{\mathit{to}[x_2]} (q_0, \emptyset) \xrightarrow{0.5} (q_0, \emptyset).
\end{aligned}$$

The run π has three blocks $B_1 = (i, \bullet)$ (x_1 is started by i and then discarded while its value is zero), $B_2 = (i\ \mathit{to}[x_2], \bot)$, and $B_3 = (i\ \mathit{to}[x_1], \times)$ (x_1 is again started in B_3 but π ends before x_1 reaches value zero). Those blocks are represented in Figure 2c, where we visually represent the timer fate of B_1 (resp. B_3) by a dotted line finished by $\bullet$ (resp. $\times$). Finally, the run τ has its blocks depicted in Figure 2d. This time, x_1 is discarded before reaching zero, i.e., $B_1 = (i, \times)$.

As illustrated by the previous example, blocks satisfy the following property.[7]

Lemma 1. *Let $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1\ \dots\ d_n\ i_n/u_n\ d_{n+1}\ (q, \kappa)$ be a padded timed run of an AT. Then, the sequences of the blocks of ρ form a partition of the set of indices $\{1, \dots, n\}$ of the actions of ρ.*

Along a timed run of an AT $\mathcal{A}$, it can happen that a timer times out at the same time that another action takes place. This leads to a sort of *nondeterminism*, as $\mathcal{A}$ can process those concurrent actions in any order. This situation appears in Example 3 each time a gap appears in the time lines of Figure 2. We call these situations *races* that we formally define as follows.

Definition 3 (Race). *Let B, B' be two blocks of a padded timed run ρ with timer fates γ and γ'. We say that B and B'* participate in a race *if:*

[7] Recall that the sequence of a block can be composed of a single action.

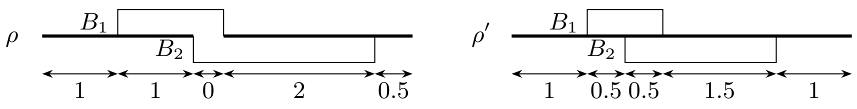


Fig. 3: Modifying the delays in order to remove a race.

- *either there exist actions $i \in B$ and $i' \in B'$ such that the sum of the delays between i and i' in ρ is equal to zero, i.e., no time elapses between them,*
- *or there exists an action $i \in B$ that is the first action along ρ to discard the timer started by the last action $i' \in B'$ and $\gamma' = \bullet$, i.e., the timer of B' (re)started by i' reaches value zero when i discards it.*

We also say that the actions i and i' participate in this race.

The first case of the race definition appears in Figure 2a, while the second case appears in Figure 2c (see the race in which blocks B_1 and B_3 participate). The nondeterminism is highlighted in Figures 2a and 2b where two actions (i and $to[x]$) occur at the same time but are processed in a different order in each figure. Unfortunately, imposing a particular way of resolving races (i.e. imposing a particular action order) may seem arbitrary when modelling real-world systems. It is therefore desirable for the set of sequences of actions along timed runs to be independent to the resolution of races.

Definition 4 (Race-avoiding). *An AT $\mathcal{A}$ is* race-avoiding *iff for all padded timed runs $\rho \in ptruns(\mathcal{A})$ with races, there exists some $\rho' \in ptruns(\mathcal{A})$ with no races such that $untime(\rho') = untime(\rho)$.*

Example 4. Let us come back to the timed run ρ of Example 2 that contains a race (see Figure 2a). By moving the second occurrence of action i slightly earlier in ρ, we obtain the timed run ρ':

$$\rho' = (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset) \xrightarrow[(x_1,1)]{i} (q_1, x_1 = 1) \xrightarrow{0.5} (q_1, x_1 = 0.5) \xrightarrow[(x_2,2)]{i} (q_2, x_1 = 0.5, x_2 = 2)$$
$$\xrightarrow{0.5} (q_2, x_1 = 0, x_2 = 1.5) \xrightarrow[\bot]{to[x_1]} (q_3, x_2 = 1.5) \xrightarrow{1.5} (q_3, x_2 = 0) \xrightarrow[\bot]{to[x_2]} (q_0, \emptyset) \xrightarrow{1} (q_0, \emptyset).$$

Notice that $untime(\rho') = q_0\ i\ q_1\ i\ q_2\ to[x_1]\ q_3\ to[x_2]\ q_0 = untime(\rho)$. Moreover, ρ' contains no races as indicated in Figure 3.

Notice that several blocks could participate in the same race. The notion of block has been defined for padded timed runs only, as we do not want to consider runs that end *abruptly* during a race (some pending timeouts may not be processed at the end of the run, for instance). Moreover, it is always possible for the first delay to be positive as no timer is active in the initial state. Finally, non-zero delays at the start and the end of the runs allow to move blocks as introduced in Example 4 and further detailed in Section 4.1.

In Section 4, we study whether it is decidable that an AT is race-avoiding, and how to eliminate races in a race-avoiding AT while keeping the same traces. Before, we study the (classical) reachability problem in Section 3.

3 Reachability

The *reachability problem* asks, given an AT $\mathcal{A}$ and a state q, whether there exists a timed run $\rho \in \mathit{truns}(\mathcal{A})$ from the initial configuration to some configuration (q, κ). In this section, we argue that this problem is PSPACE-complete.

Theorem 1. *The reachability problem for ATs is* PSPACE*-complete.*

For hardness, we reduce from the acceptance problem for linear-bounded Turing machines (LBTM, for short), as done for timed automata, see e.g. [1]. In short, given an LBTM $\mathcal{M}$ and a word w of length n, we construct an AT that uses n timers x_i, $1 \leq i \leq n$, such that the timer x_i encodes the value of the i-th cell of the tape of $\mathcal{M}$. We also rely on a timer x that is always (re)started at one, and is used to synchronize the x_i timers and the simulation of $\mathcal{M}$. The simulation is split into *phases*: The AT first seeks the symbol on the current cell i of the tape (which can be derived from the moment at which the timer x_i times out, using the number of times x timed out since the beginning of the phase). Then, the AT simulates a transition of $\mathcal{M}$ by restarting x_i, reflecting the new value of the i-th cell. Finally, the AT can reach a designated state iff $\mathcal{M}$ is in an accepting state. Therefore, the reachability problem is PSPACE-hard.

For membership, we follow the classical argument used to establish that the reachability problem for timed automata is in PSPACE: We first define *region automata* for ATs (which are a simplification of region automata for timed automata) and observe that reachability in an AT reduces to reachability in the corresponding region automaton. The region automaton is of size exponential in the number of timers and polynomial in the number of states of the AT. Hence, the reachability problem for ATs is in PSPACE via standard arguments.

We define region automata for ATs much like they are defined for timed automata [3, 4, 7]. Let $\mathcal{A} = (X, I, Q, q_0, \chi, \delta)$ be an AT. For a timer $x \in X$, c_x denotes the largest constant to which x is updated in $\mathcal{A}$. Let $C = \max_{x \in X} c_x$. Two valuations κ and κ' are said *timer-equivalent*, noted $\kappa \cong \kappa'$, iff $\mathsf{dom}(\kappa) = \mathsf{dom}(\kappa')$ and the following hold for all $x_1, x_2 \in \mathsf{dom}(\kappa)$: *(i)* $\lfloor \kappa(x_1) \rfloor = \lfloor \kappa'(x_1) \rfloor$, *(ii)* $\mathrm{frac}(\kappa(x_1)) = 0$ iff $\mathrm{frac}(\kappa'(x_1)) = 0$, *(iii)* $\mathrm{frac}(\kappa(x_1)) \leq \mathrm{frac}(\kappa(x_2))$ iff $\mathrm{frac}(\kappa'(x_1)) \leq \mathrm{frac}(\kappa'(x_2))$. A *timer region* for $\mathcal{A}$ is an equivalence class of timer valuations induced by $\cong$. We lift the relation to configurations: $(q, \kappa) \cong (q', \kappa')$ iff $\kappa \cong \kappa'$ and $q = q'$. Finally, $[\![(q, \kappa)]\!]_{\cong}$ denotes the equivalence class of (q, κ).

We are now able to define a finite automaton called the *region automaton* of $\mathcal{A}$ and denoted $\mathcal{R}$. The alphabet of $\mathcal{R}$ is $\Sigma = \{\tau\} \cup \hat{I}$ where τ is a special symbol used in non-zero delay transitions. Formally, $\mathcal{R}$ is the finite automaton (Σ, S, s_0, Δ) where:

- $S = \{(q, \kappa) \mid q \in Q, \kappa \in \mathsf{Val}(\chi(q))\}_{/\cong}$, i.e., the quotient of the configurations by $\cong$, is the set of states,
- $s_0 = [\![(q_0, \kappa_0)]\!]_{\cong}$ with κ_0 the empty valuation, is the initial state,
- the set of transitions $\Delta \subseteq S \times \Sigma \times S$ includes $([\![(q, \kappa)]\!]_{\cong}, \tau, [\![(q, \kappa')]\!]_{\cong})$ if $(q, \kappa) \xrightarrow{d} (q, \kappa')$ in $\mathcal{A}$ whenever $d > 0$, and $([\![(q, \kappa)]\!]_{\cong}, i, [\![(q', \kappa')]\!]_{\cong})$ if $(q, \kappa) \xrightarrow[u]{i} (q', \kappa')$ in $\mathcal{A}$.

It is easy to check that the timer-equivalence relation on configurations is a *(strong) time-abstracting bisimulation* [10,17]. That is, for all $(q_1, \kappa_1) \cong (q_2, \kappa_2)$ the following holds:

– if $(q_1, \kappa_1) \xrightarrow[u]{i} (q_1', \kappa_1')$, then there is $(q_2, \kappa_2) \xrightarrow[u]{i} (q_2', \kappa_2')$ with $(q_1', \kappa_1') \cong (q_2', \kappa_2')$,
– if $(q_1, \kappa_1) \xrightarrow{d_1} (q_1, \kappa_1')$, then there exists $(q_2, \kappa_2) \xrightarrow{d_2} (q_2, \kappa_2')$ where d_1, $d_2 > 0$ may differ such that $(q_1, \kappa_1') \cong (q_2, \kappa_2')$, and
– the above also holds if (q_1, κ_1) and (q_2, κ_2) are swapped.

Using this property, we can prove the following about $\mathcal{R}$.

Lemma 2. *Let $\mathcal{A} = (X, I, Q, q_0, \chi, \delta)$ be an AT and $\mathcal{R}$ be its region automaton.*

1. *The size of $\mathcal{R}$ is linear in $|Q|$ and exponential in $|X|$. That is, $|S|$ is smaller than or equal to $|Q| \cdot |X|! \cdot 2^{|X|} \cdot (C+1)^{|X|}$.*
2. *There is a timed run ρ of $\mathcal{A}$ that begins in (q, κ) and ends in (q', κ') iff there is a run ρ' of $\mathcal{R}$ that begins in $[\![(q, \kappa)]\!]_{\cong}$ and ends in $[\![(q', \kappa')]\!]_{\cong}$.*

Example 5. Let us consider the AT $\mathcal{A}$ of Figure 1 and the timed run π given in Example 3. The corresponding run π' in the region automaton $\mathcal{R}$ is

$$\begin{aligned}
&(q_0, [\![\emptyset]\!]_{\cong}) \xrightarrow{\tau} (q_0, [\![\emptyset]\!]_{\cong}) \xrightarrow{i} (q_1, [\![x_1 = 1]\!]_{\cong}) \xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 2]\!]_{\cong}) \\
&\xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 1]\!]_{\cong}) \xrightarrow{i} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 0]\!]_{\cong}) \\
&\xrightarrow{to[x_2]} (q_1, [\![x_1 = 0]\!]_{\cong}) \xrightarrow{to[x_1]} (q_1, [\![x_1 = 1]\!]_{\cong}) \xrightarrow{\tau} (q_1, [\![0 < x_1 < 1]\!]_{\cong}).
\end{aligned}$$

Notice that the transitions with delay zero of π do not appear in π'.

4 Race-avoiding ATs

In this section, we study whether an AT $\mathcal{A}$ is race-avoiding, i.e., whether for any padded timed run ρ of $\mathcal{A}$ with races, there exists another run ρ' with no races such that $\mathit{untime}(\rho) = \mathit{untime}(\rho')$. We are able to prove the next theorem.

Theorem 2. *Deciding whether an AT is race-avoiding is* PSPACE*-hard and in* 3EXP*. It is in* PSPACE *if the sets of actions I and of timers X are fixed.*

Our approach is, given $\rho \in \mathit{ptruns}(\mathcal{A})$, to study how to slightly move blocks along the time line of ρ in a way to get another $\rho' \in \mathit{ptruns}(\mathcal{A})$ where the races are eliminated while keeping the actions in the same order as in ρ. We call this action *wiggling*. Let us first give an example and then formalize this notion.

Example 6. We consider again the AT of Figure 1. We have seen in Example 4 and Figure 3 that the block B_2 of ρ can be slightly moved to the left to obtain the timed run ρ' with no race such that $\mathit{untime}(\rho) = \mathit{untime}(\rho')$. Figure 3 illustrates how to move B_2 by changing some of the delays.

In contrast, this is not possible for the timed run π of Example 3. Indeed looking at Figure 2c, we see that it is impossible to move block B_2 to the left due to its race with B_1 (remember that we need to keep the same action order). It is also not possible to move it to the right due to its race with B_3. Similarly, it is impossible to move B_1 neither to the right (due to its race with B_2), nor to the left (otherwise its timer will time out instead of being discarded by B_3). Finally, one can also check that block B_3 cannot be moved.

Given a padded timed run $\rho = (q_0, \kappa_0)\ d_1\ i_1/u_1\ \dots\ d_n\ i_n/u_n\ d_{n+1}(q, \kappa) \in \mathit{ptruns}(\mathcal{A})$ and a block $B = (k_1 \dots k_m, \gamma)$ of ρ participating in a race, we say that we can *wiggle* B if for some ϵ, we can move B to the left (by $\epsilon < 0$) or to the right (by $\epsilon > 0$), and obtain a run $\rho' \in \mathit{ptruns}(\mathcal{A})$ such that $\mathit{untime}(\rho) = \mathit{untime}(\rho')$ and B no longer participates in any race. More precisely, we have $\rho' = (q_0, \kappa_0)\ d'_1\ i_1/u_1\ \dots\ d'_n\ i_n/u_n\ d'_{n+1}\ (q, \kappa')$ such that

– for all $i_{k_\ell} \in B$ with $k_\ell > 1$, if $i_{k_\ell - 1} \notin B$ (the action before i_{k_ℓ} in ρ does not belong to B), then $d'_{k_\ell} = d_{k_\ell} + \epsilon$,
– if there exists $i_{k_\ell} \in B$ with $k_\ell = 1$ (the first action of B is the first action of ρ), then $d'_1 = d_1 + \epsilon$,
– for all $i_{k_\ell} \in B$ with $k_\ell < n$, if $i_{k_\ell + 1} \notin B$ (the action after i_{k_ℓ} in ρ does not belong to B), then $d'_{k_\ell + 1} = d_{k_\ell + 1} - \epsilon$,
– if there exists $i_{k_\ell} \in B$ with $k_\ell = n$ (the last action of B is the last action of ρ), then $d'_{n+1} = d_{n+1} - \epsilon$,
– for all other d'_k, we have $d'_k = d_k$.

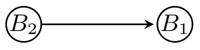
As $\rho' \in \mathit{ptruns}(\mathcal{A})$ and $\mathit{untime}(\rho) = \mathit{untime}(\rho')$, we must have $d'_k \geq 0$ for all k and $d'_1, d'_{n+1} > 0$. Observe that to wiggle B we move every action of B.

We say that we can wiggle ρ, or that ρ is *wigglable*, if it is possible to wiggle its blocks, *one block at a time*, to obtain $\rho' \in \mathit{ptruns}(\mathcal{A})$ with no races such $\mathit{untime}(\rho) = \mathit{untime}(\rho')$. Hence if all padded timed runs with races of an AT $\mathcal{A}$ are wigglable, then $\mathcal{A}$ is race-avoiding.

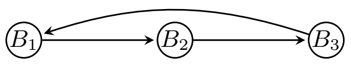
In the next sections, we first associate a graph with any $\rho \in \mathit{ptruns}(\mathcal{A})$ in a way to characterize when ρ is wigglable thanks to this graph. We then state the equivalence between the race-avoiding characteristic of an AT and the property that all $\rho \in \mathit{ptruns}(\mathcal{A})$ can be wiggled (Theorem 3). This allows us to provide logic formulas to determine whether an AT has an unwigglable run, and then to prove the upper bound of Theorem 2. We also discuss its lower bound. Finally, we discuss some sufficient hypotheses for a race-avoiding AT .

4.1 Wiggling a run

In this section, given an AT $\mathcal{A}$ and a padded timed run $\rho \in \mathit{ptruns}(\mathcal{A})$, we study the conditions required to be able to wiggle ρ. For this purpose, we define the following graph G_ρ associated with ρ. When two blocks B and B' of ρ participate in a race, we write $B \prec B'$ if there exist actions $i \in B$ and $i' \in B'$ such that i, i' participate in this race and, according to Definition 3:



(a) Graph G_ρ.



(b) Graph G_π.

Fig. 4: Block graphs of the timed runs ρ and π of Example 3.

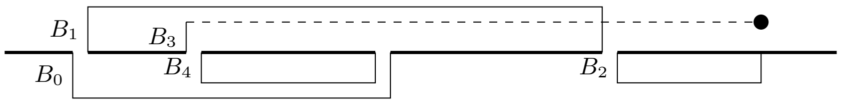


Fig. 5: Races of a padded timed run ρ with $B_\ell \prec B_{\ell+1 \bmod 5}$, $0 \leq \ell \leq 4$.

– either i occurs before i' along ρ and the total delay between i and i' is zero.
– or the timer of B' (re)started by i' reaches value zero when i discards it.

We define the *block graph* $G_\rho = (V, E)$ of ρ where V is the set of blocks of ρ and E has an edge (B, B') iff $B \prec B'$.

Example 7. Let $\mathcal{A}$ be the AT from Figure 1, and ρ and π be the timed runs from Example 3, whose block decomposition is represented in Figures 2a and 2c. For the run ρ, it holds that $B_2 \prec B_1$ leading to the block graph G_ρ depicted in Figure 4a. For the run π, we get the block graph G_π depicted in Figure 4b.

Notice that G_ρ is acyclic while G_π is cyclic. By the following proposition, this difference is enough to characterize that ρ can be wiggled and π cannot.

Proposition 1. *Let $\mathcal{A}$ be an AT and $\rho \in ptruns(\mathcal{A})$ be a padded timed run with races. Then, ρ can be wiggled iff G_ρ is acyclic.*

Intuitively, a block cannot be moved to the left (resp. right) if it has a predecessor (resp. successor) in the block graph, due to the races in which it participates. Hence, if a block has both a predecessor and a successor, it cannot be wiggled (see Figures 2c and 4b for instance). Then, the blocks appearing in a cycle of the block graph cannot be wiggled. The other direction holds by observing that we can do a topological sort of the blocks if the graph is acyclic. We then wiggle the blocks, one by one, according to that sort.

The next corollary will be useful in the following sections. It is illustrated by Figure 5 with the simple cycle $(B_0, B_1, B_2, B_3, B_4, B_0)$.

Corollary 1. *Let $\mathcal{A}$ be an AT and $\rho \in ptruns(\mathcal{A})$ be a padded timed run with races. Suppose that G_ρ is cyclic. Then there exists a cycle $\mathcal{C}$ in G_ρ such that*

– *any block of $\mathcal{C}$ participates in exactly two races described by this cycle,*
– *for any race described by $\mathcal{C}$, exactly two blocks of $\mathcal{C}$ participate in the race,*
– *the blocks $B = (k_1 \dots k_m, \gamma)$ of $\mathcal{C}$ satisfy either $m \geq 2$, or $m = 1$ and $\gamma = \bullet$.*

From the definition of wiggling, we know that if all padded timed runs with races of an AT $\mathcal{A}$ are wigglable, then $\mathcal{A}$ is race-avoiding. The converse also holds as stated in the next theorem. By Proposition 1, this means that an AT is race-avoiding iff the block graph of all its padded timed run is acyclic.

Theorem 3. *An AT $\mathcal{A}$ is race-avoiding*

- *iff any padded timed run $\rho \in ptruns(\mathcal{A})$ with races can be wiggled,*
- *iff for any padded timed run $\rho \in ptruns(\mathcal{A})$, its graph G_ρ is acyclic.*

Let us sketch the missing proof. By modifying $\rho \in ptruns(\mathcal{A})$ to explicitly encode when a timer is discarded, one can show the races of ρ cannot be avoided if the block graph of ρ is cyclic as follows. Given two actions i, i' of this modified run, it is possible to define the *relative elapsed time* between i and i', noted $\mathrm{reltime}(i, i')$, from the sum d of all delays between i and i': if i occurs before i', then $\mathrm{reltime}(i, i') = d$, otherwise $\mathrm{reltime}(i, i') = -d$. Lifting this to a sequence of actions from ρ is defined naturally. Then, one can observe that the relative elapsed time of a cyclic sequence of actions is zero, i.e., $\mathrm{reltime}(i_1, i_2, \dots, i_k, i_1) = 0$. Finally, from a cycle of G_ρ as described in Corollary 1, we extract a cyclic sequence of actions and prove, thanks to the concept of relative elapsed time, that any run ρ' such that $untime(\rho) = untime(\rho')$ must contain some races.

4.2 Existence of an unwigglable run

In this section, we give the intuition for the announced complexity bounds for the problem of deciding whether an AT $\mathcal{A}$ is race-avoiding (Theorem 2).

Let us begin with the 3EXP-membership. The crux of our approach is to use the characterization of the race-avoiding property given in Theorem 3, to work with a slight modification of the region automaton $\mathcal{R}$ of $\mathcal{A}$, and to construct a finite-state automaton whose language is the set of runs of $\mathcal{R}$ whose block graph is cyclic. Hence deciding whether $\mathcal{A}$ is race-avoiding amounts to deciding whether the language accepted by the latter automaton is empty. To do so, we construct a *monadic second-order* (MSO, for short; see [12,16] for an introduction) formula that is satisfied by words w labeling a run ρ of $\mathcal{R}$ iff the block graph of ρ is cyclic.

Our *modification* of $\mathcal{R}$ is best seen as additional annotations on the states and transitions of $\mathcal{R}$. We extend the alphabet Σ as follows: *(i)* we add a timer to each action $i \in \hat{I}$ to remember the updated timers; *(ii)* we also use new symbols $di[x]$, $x \in X$, with the intent of explicitly encoding in $\mathcal{R}$ when the timer x is discarded while its value is zero. Therefore, the modified alphabet is $\Sigma = \{\tau\} \cup (\hat{I} \times \hat{X}) \cup \{di[x] \mid x \in X\}$ where $\hat{X} = X \cup \{\bot\}$. As a transition in $\mathcal{A}$ can discard more than one timer, we store the set D of discarded timers in the states of $\mathcal{R}$, as well as outgoing transitions labeled by $di[x]$, for all discarded timers x. For this, the states of $\mathcal{R}$ become $S = \{(q, [\![\kappa]\!]_{\cong}, D) \mid q \in Q, \kappa \in \mathsf{Val}(\chi(q)), D \subseteq X\}$ and Δ is modified in the natural way so that D is updated as required. Note that the size of this modified $\mathcal{R}$ is only larger than what is stated in Lemma 2 by a factor of $2^{|X|}$.

Note that any x-block $(i_{k_1}, \dots, i_{k_m}, \gamma)$ of a timed run ρ in $\mathcal{A}$ is translated into the sequence of symbols $(i'_{k_1}, \dots, i'_{k_m}, \gamma')$ in the corresponding run ρ' of the modified $\mathcal{R}$ with an optional symbol γ' such that:

- $i'_{k_\ell} = (i_{k_\ell}, x)$, for $1 \leq \ell < m$,
- $i'_{k_m} = (i_{k_m}, \bot)$ if $\gamma = \bot$, and (i_{k_m}, x) otherwise,

– $\gamma' = \mathit{di}[x]$ if $\gamma = \bullet$, and γ' does not exist otherwise.

It follows that, instead of considering padded timed runs $\rho \in \mathit{ptruns}(\mathcal{A})$ and their block graph G_ρ, we work with their corresponding (padded) runs, blocks, and block graphs in the modified region automaton $\mathcal{R}$ of $\mathcal{A}$.

Example 8. The run π' of Example 5 becomes

$$\begin{aligned}
&(q_0, [\![\emptyset]\!]_{\cong}, \emptyset) \xrightarrow{\tau} (q_0, [\![\emptyset]\!]_{\cong}, \emptyset) \xrightarrow{(i,x_1)} (q_1, [\![x_1 = 1]\!]_{\cong}, \emptyset) \xrightarrow{(i,x_2)} (q_2, [\![x_1 = 1, x_2 = 2]\!]_{\cong}, \emptyset) \\
&\quad \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 1]\!]_{\cong}, \emptyset) \xrightarrow{(i,x_1)} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}, \{x_1\}) \\
&\quad \xrightarrow{\mathit{di}[x_1]} (q_2, [\![x_1 = 1, x_2 = 1]\!]_{\cong}, \emptyset) \xrightarrow{\tau} (q_2, [\![x_1 = 0, x_2 = 0]\!]_{\cong}, \emptyset) \\
&\quad \xrightarrow{(\mathit{to}[x_2],\bot)} (q_1, [\![x_1 = 0]\!]_{\cong}, \emptyset) \xrightarrow{(\mathit{to}[x_1],x_1)} (q_1, [\![x_1 = 1]\!]_{\cong}, \emptyset) \xrightarrow{\tau} (q_1, [\![0 < x_1 < 1]\!]_{\cong}, \emptyset).
\end{aligned}$$

The transition with label $\mathit{di}[x_1]$ indicates that the timer x_1 is discarded in the original timed run while its value equals zero (see the race in which blocks B_1 and B_3 participate in Figure 2c). The three blocks of π become $B'_1 = ((i, x_1), \mathit{di}[x_1])$, $B'_2 = ((i, x_2), (\mathit{to}[x_2], \bot))$, and $B'_3 = ((i, x_1), (\mathit{to}[x_1], x_1))$ in π'. The fact that in π, B_1 and B_2 participate in a race (with a zero-delay between their respective actions i and i), is translated in π' with the non existence of the τ-symbol between the symbols (i, x_1) and (i, x_2) in B'_1 and B'_2 respectively.

Lemma 3. *Let $\mathcal{A}$ be an AT and $\mathcal{R}$ be its modified region automaton. We can construct an MSO formula Φ of size linear in I and X such that a word labeling a run ρ of $\mathcal{R}$ satisfies Φ iff ρ is a padded run that cannot be wiggled. Moreover, the formula Φ, in prenex normal form, has three quantifier alternations.*

The formula Φ of this lemma describes the existence of a cycle $\mathcal{C}$ of blocks $B_0, B_1, \ldots, B_{k-1}$ such that $B_\ell \prec B_{\ell+1 \bmod k}$ for any $0 \le \ell \le k-1$, as described in Corollary 1 (see Figure 5). To do so, we consider the actions (i.e., symbols of the alphabet Σ of $\mathcal{R}$) participating in the races of $\mathcal{C}$: $i_0, i_1, \ldots, i_{k-1}$ and $i'_0, i'_1, \ldots, i'_{k-1}$ such that for all ℓ, i_ℓ, i'_ℓ belong to the same block, and $i'_\ell, i_{\ell+1 \bmod k}$ participate in a race. One can write MSO formulas expressing that two actions participate in a race (there is no τ transition between them), that two actions belong to the same block, and, finally, the existence of these two action sequences.

From the formula Φ of Lemma 3, by the Büchi-Elgot-Trakhtenbrot theorem, we can construct a finite-state automaton whose language is the set of all words satisfying Φ. Its size is triple-exponential. We then compute the intersection $\mathcal{N}$ of this automaton with $\mathcal{R}$ — itself exponential in size. Finally, the language of $\mathcal{N}$ is empty iff each padded timed run of $\mathcal{A}$ can be wiggled, and emptiness can be checked in polynomial time with respect to the triple-exponential size of $\mathcal{N}$, thus showing the 3EXP-membership of Theorem 2. Notice that when we fix the sets of inputs I and of timers X, the formula Φ becomes of constant size. Constructing $\mathcal{N}$ and checking its emptiness can be done "on the fly", yielding a nondeterministic decision procedure which requires a polynomial space only.

We thus obtain that, under fixed inputs and timers, deciding whether an AT is race-avoiding is in PSPACE.

The complexity lower bound of Theorem 2 follows from the PSPACE-hardness of the reachability problem for ATs (see the intuition given in Section 3). We can show that any run in the AT constructed from the given LBTM and word w can be wiggled. Once the designated state for the reachability reduction is reached, we add a widget that forces a run that cannot be wiggled. Therefore, as the only way of obtaining a run that cannot be wiggled is to reach a specific state (from the widget), the problem whether an AT is race-avoiding is PSPACE-hard. Notice that this hardness proof is no longer valid if we fix the sets I and X.

4.3 Sufficient hypotheses

Let us discuss some sufficient hypotheses for an AT $\mathcal{A}$ to be race-avoiding.

1. If every state in $\mathcal{A}$ has at most one active timer, then $\mathcal{A}$ is race-avoiding. Up to renaming the timers, we actually have a single-timer AT in this case.
2. If we modify the notion of timed run by imposing non-zero delays everywhere in the run, then $\mathcal{A}$ is race-avoiding. Indeed, the only races that can appear are when a zero-valued timer is discarded, and it is impossible to form a cycle in the block graph with only this kind of races. Imposing a non-zero delay before a timeout is debateable. Nevertheless, imposing a non-zero delay before inputs only is not a sufficient hypothesis.
3. Let us fix a total order $<$ over the timers, and modify the semantics of an AT to enforce that, in a race, any action of an x-block is processed before an action of a y-block, if $x < y$ (x is preemptive over y). Then the AT is race-avoiding. Towards a contradiction, assume there are blocks $B_0, B_1, \ldots, B_{k-1}$ forming a cycle as described in Corollary 1, where each B_i is an x_i-block. By the order and the races, we get $x_0 \leq x_1 \leq \ldots \leq x_{k-1} \leq x_0$, i.e., we have a single timer (as in the first hypothesis). Hence, it is always possible to wiggle, which is a contradiction.

5 Conclusion and future work

In this paper, we studied automata with timers. We proved that the reachability problem for ATs is PSPACE-complete. Moreover, given a padded timed run in an AT, we defined a decomposition of its actions into blocks, and provided a way to remove races (concurrent actions) inside the run by wiggling blocks one by one. We also proved that this notion of wiggling is necessary and sufficient to decide whether an AT is race-avoiding. Finally, we showed that deciding whether an AT is race-avoiding is in 3EXP and PSPACE-hard.

For future work, it may be interesting to tighten the complexity bounds for the latter decision problem, both when fixing the sets I and X and when not. A second important direction, which we plan to pursue, is to work on a learning algorithm for ATs, as initiated in [18] with Mealy machines with one timer. This would allow us to construct ATs from real-world systems, such as network protocols, in order to verify that these systems behave as expected.

References

1. Aceto, L., Laroussinie, F.: Is your model checker on time? on the complexity of model checking for timed modal logics. J. Log. Algebraic Methods Program. **52-53**, 7–51 (2002). https://doi.org/10.1016/S1567-8326(02)00022-X
2. Aichernig, B.K., Pferscher, A., Tappler, M.: From Passive to Active: Learning Timed Automata Efficiently. In: Lee, R., Jha, S., Mavridou, A. (eds.) NFM'20. LNCS, vol. 12229, pp. 1–19. Springer (2020)
3. Alur, R.: Timed automata. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. pp. 8–22. Springer (1999)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
5. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 444–462. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_25
6. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
7. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)
8. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Ouaknine, J., Worrell, J.: Model checking real-time systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1001–1046. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_29
9. Bruyère, V., Pérez, G.A., Staquet, G., Vaandrager, F.W.: Automata with timers. CoRR **abs/2305.07451** (2023). https://doi.org/10.48550/arXiv.2305.07451
10. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
11. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings. Lecture Notes in Computer Science, vol. 407, pp. 197–212. Springer (1989). https://doi.org/10.1007/3-540-52148-8_17
12. Grädel, E., Thomas, W., Wilke, T.: Automata, logics, and infinite games: a guide to current research, vol. 2500. Springer (2003)
13. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theor. Comput. Sci. **411**(47), 4029–4054 (2010). https://doi.org/10.1016/j.tcs.2010.07.008
14. 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, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4137, pp. 435–449. Springer (2006). https://doi.org/10.1007/11817949_29
15. Howar, F., Steffen, B.: Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27,

2016, Revised Papers. Lecture Notes in Computer Science, vol. 11026, pp. 123–148. Springer (2018). https://doi.org/10.1007/978-3-319-96562-8_5

16. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, Volume 3: Beyond Words, pp. 389–455. Springer (1997). https://doi.org/10.1007/978-3-642-59126-6_7
17. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Formal Methods Syst. Des. **18**(1), 25–68 (2001). https://doi.org/10.1023/A:1008734703554
18. Vaandrager, F., Ebrahimi, M., Bloem, R.: Learning Mealy machines with one timer. Information and Computation p. 105013 (2023). https://doi.org/https://doi.org/10.1016/j.ic.2023.105013