

# The monitoring problem for timed automata

**Alejandro Grez**

Pontificia Universidad Católica de Chile, Chile  
ajgrez@uc.cl

**Filip Mazowiecki**

Max Planck Institute for Software Systems, Germany  
filipm@mpi-sws.org

**Michał Pilipczuk**

University of Warsaw, Poland  
michal.pilipczuk@mimuw.edu.pl

**Gabriele Puppis**

University of Udine, Italy  
gabriele.puppis@uniud.it

**Cristian Riveros**

Pontificia Universidad Católica de Chile, Chile  
cristian.riveros@uc.cl

---

## Abstract

We study a variant of the classical membership problem in automata theory, which consists of deciding whether a given input word is accepted by a given automaton. We do so under a different perspective, that is, we consider a dynamic version of the problem, called *monitoring problem*, where the automaton is fixed and *the input is revealed as in a stream, one symbol at a time following the natural order on positions*. The goal here is to design a dynamic data structure that can be queried about whether the word consisting of symbols revealed so far is accepted by the automaton, and that can be efficiently updated when the next symbol is revealed. We provide complexity bounds for this monitoring problem, by considering timed automata that process symbols interleaved with timestamps. The main contribution is that *monitoring of a one-clock timed automaton, with all its components but the clock constants fixed, can be done in amortised constant time per input symbol*.

**2012 ACM Subject Classification** Theory of computation → Models of computation

**Keywords and phrases** timed automata, monitoring problem, data stream, dynamic data structure

**Digital Object Identifier** 10.4230/LIPIcs.???-???-??

**Funding** *Michał Pilipczuk*: This work is a part of project TOTAL that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 677651.



© Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis and Cristian Riveros;  
licensed under Creative Commons License CC-BY

???

Editors: ???; Article No. ??; pp. ??:1–??:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 1 Introduction

We consider the problem of *monitoring* streams of data against a property. Precisely, the problem assumes that there is a fixed property  $P \subseteq \Sigma^*$  over a data domain  $\Sigma$  and an arbitrary stream  $w = d_1d_2d_3\cdots \in \Sigma^\omega$ , and the goal is to tell, at each step  $n$ , whether the  $n$ -element prefix  $d_1d_2\cdots d_n$  of the stream verifies the property  $P$ . For example, in the simpler form of the problem, the property  $P$  may be a regular language given by a finite state automaton. In this case the monitoring problem could be thought of as a natural variant of the membership problem in automata theory. In general, the monitoring problem requires to devise, for a given property  $P$ , a *dynamic data structure* that maintains the information whether the current stream prefix satisfies  $P$  or not. This data structure should be updated whenever the next symbol of the input stream arrives, and we would like to perform these update operations *efficiently*: possibly in constant time, and thus independently of the input stream and of the amount of data consumed at any moment. Of course, the complexity of the operations may depend on the property  $P$ , but this is usually considered fixed, since the focus here is on the asymptotic complexity for monitoring streams of arbitrary length.

In this paper we consider the monitoring problem for “regular” properties enhanced with time constraints. Specifically, we consider properties given by *timed automata* [4], and streams of data that are timed words, that is, sequences of letters from a finite alphabet interleaved with timestamps. The main contribution is that monitoring of a one-clock timed automaton  $\mathcal{A}$  can be done in amortised constant time per consumed input element, assuming that  $\mathcal{A}$  is fixed beforehand. In fact, we provide a finer complexity result, as we design a dynamic data structure that supports monitoring with time complexity that does not depend on the magnitude of the clock constants that appear in the transitions of  $\mathcal{A}$ . We also show that when the stream is discrete (when the timestamps increase by exactly one unit of time) then monitoring in constant (non-amortised) time is possible.

We complement these algorithmic results with a complexity lower bound, in which we consider timed automata with multiple clocks and additive constraints [7]. We show that, subject to the 3SUM Conjecture, the resulting family of properties cannot be monitored in amortised constant time, and not even in amortised strongly sub-linear time. The 3SUM Conjecture is among the most popular hypotheses considered in computational geometry and in fine-grained complexity theory; see e.g. an overview in [2, Appendix A].

The considered timed variant of the monitoring problem turns out to be particularly meaningful in the context of data streaming algorithms [5, 10] and complex event recognition and processing [9]. There the input stream is produced by some sensors, or other similar systems, and the importance of each data item rapidly decays over time. In this setting, one would expect that the relevant data comes with an implicit time horizon, implying that the properties to be verified can be often relativised to one or more time windows. For example, in complex event processing, one can imagine a specification language that extends classical regular expressions with constructs of the form  $E$  **within**  $X$ , requiring that a factor of the stream exists that satisfies the regular expression  $E$  and that spans over at most  $X$  time units. Some properties are easily captured by one-clock timed automata, and thus can be efficiently monitored using our dynamic data structure.

**Related work.** The monitoring problem we study here somewhat resembles the context of *streaming algorithms*; see e.g. [5, 10, 18] for works with a similar motivation. In this context, a typical problem is to compute (possibly approximately) some statistic or aggregate over a data sequence, where the main point is to assume a severe restriction on the space usage. Note that in our setting, we focus on obtaining low time complexity of updates and queries, rather

than studying the space complexity, so our work leans more towards the area of dynamic query evaluation [8, 19]. For Boolean properties (like in our monitoring problem), several papers [21, 22, 6] have considered streaming algorithms for testing membership in regular and context-free languages. Another variant of the problem was considered in [15, 14, 13], where the regular property is verified on the last  $N$  letters of the stream instead of the entire prefix up to the current position.

The closest to our setting is the work [24], which studies the monitoring problem for finite automata over a sliding window. The authors provide a structure that takes constant time to update. We explain in Section 3 that this is a special case of our results for timed automata. In Appendix A we give other application examples of our result.

## 2 Preliminaries

**Finite automata.** A *finite automaton* is a tuple  $\mathcal{A} = (\Sigma, Q, I, E, F)$ , where  $\Sigma$  is a finite alphabet,  $Q$  is a finite set of states,  $E \subseteq Q \times \Sigma \times Q$  is a transition relation, and  $I, F \subseteq Q$  are the sets of initial and final states. A run of  $\mathcal{A}$  on a word  $w = a_1 \dots a_n \in \Sigma^*$  is a sequence  $\rho = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ , where  $(q_{i-1}, a_i, q_i) \in E$  for all  $i = 1, \dots, n$ . Moreover,  $\rho$  is a *successful* run if  $q_0 \in I$  and  $q_n \in F$ . The language *recognized* by  $\mathcal{A}$  is the set  $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* : \text{there is a successful run of } \mathcal{A} \text{ on } w\}$ .

**Timed automata.** Let  $X$  be a finite set of clocks, usually denoted by  $x, y, \dots$ . A *clock valuation* is any function  $\nu : X \rightarrow \mathbb{R}_{\geq 0}$  from clocks to non-negative real numbers. *Clock conditions* are formulas defined by the grammar  $C_X := \text{true} \mid x < c \mid x > c \mid x = c \mid C_X \wedge C_X \mid C_X \vee C_X$ , where  $x \in X$  and  $c \in \mathbb{R}_{\geq 0}$ . By a slight abuse of notation, we denote by  $C_X$  the set of all clock conditions over the clock set  $X$ . Given a clock condition  $\gamma$  and a valuation  $\nu$ , we say that  $\nu$  *satisfies*  $\gamma$ , and write  $\nu \models \gamma$ , if the arithmetic formula obtained from  $\gamma$  by substituting each clock  $x$  with its value  $\nu(x)$  evaluates to true.

A *timed automaton* is a tuple  $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ , where  $Q, \Sigma, I, F$  are defined exactly as for finite automata,  $X$  is a finite set of clocks, and  $E \subseteq Q \times \Sigma \times C_X \times Q \times 2^X$  is a transition relation. We say that  $c \in \mathbb{R}_{\geq 0}$  is a *clock constant* of  $\mathcal{A}$  if  $c$  appears in some clock condition of a transition from  $E$ . A *configuration* of  $\mathcal{A}$  is a pair  $(q, \nu)$ , where  $q \in Q$  and  $\nu$  is a clock valuation. Recall that finite automata process words over a finite alphabet  $\Sigma$ ; likewise, timed automata process timed words over an alphabet of the form  $\Sigma \uplus \mathbb{R}_{>0}$ , with  $\Sigma$  finite.<sup>1</sup> Given a timed automaton  $\mathcal{A}$  and a timed word  $w = e_1 \dots e_n \in (\Sigma \uplus \mathbb{R}_{>0})^*$ , a run of  $\mathcal{A}$  on  $w$  is any sequence  $\rho = (q_0, \nu_0) \xrightarrow{e_1} (q_1, \nu_1) \xrightarrow{e_2} \dots \xrightarrow{e_n} (q_n, \nu_n)$ , where each  $(q_i, \nu_i)$  is a configuration and

- if  $e_i \in \mathbb{R}_{>0}$ , then  $q_{i+1} = q_i$  and  $\nu_{i+1}(x) = \nu_i(x) + e_i$  for all  $x \in X$ ;
- if  $e_i \in \Sigma$ , then there is a transition  $(q_i, e_i, \gamma, q_{i+1}, Z) \in E$  such that  $\nu_i \models \gamma$  and either  $\nu_{i+1}(x) = 0$  or  $\nu_{i+1}(x) = \nu_i(x)$  depending on whether  $x \in Z$  or  $x \in X \setminus Z$ .

Thus, the set  $Z$  in a transition  $(q_i, e_i, \gamma, q_{i+1}, Z) \in E$  corresponds to the subset of clocks that are reset when firing the transition. Note that the values of the other clocks stay unchanged.

A run  $\rho$  as above is *successful* if  $q_0 \in I$ ,  $\nu_0(x) = 0$  for all  $x \in X$ , and  $q_n \in F$ . The language *recognised* by  $\mathcal{A}$  is the set  $\mathcal{L}(\mathcal{A}) = \{w \in (\Sigma \uplus \mathbb{R}_{>0})^* : \text{there is a successful run of } \mathcal{A} \text{ on } w\}$ .

**Size of an automaton.** The size of a finite automaton  $\mathcal{A} = (\Sigma, Q, I, E, F)$  is defined as  $|\mathcal{A}| = |Q| + |E|$ . This is asymptotically equivalent to essentially every possible definition of

<sup>1</sup> In the literature, timed words are often defined as words over  $\Sigma \times \mathbb{R}_{>0}$ , meaning that every letter is tagged with a timestamp. Here, instead, we define a timed word as a sequence of letters possibly interleaved with time spans from  $\mathbb{R}_{>0}$ . In this case, the timestamp of a letter is implicitly determined by the sum of the time spans before it. The current presentation enables a simpler definition of transitions.

size of a finite automaton that can be found in the literature. The size of a timed automaton  $\mathcal{A} = (\Sigma, Q, X, I, E, F)$  is instead defined as  $|\mathcal{A}| = |Q| + |X| + \sum_{(p,a,\gamma,q,Z) \in E} |\gamma|$ , where  $|\gamma|$  is the number of atomic expressions (i.e. expressions of the form  $x < c$ ,  $x > c$ ,  $x = c$ ) appearing in the clock condition  $\gamma$ . *Note that the size of a timed automaton does not take into account the magnitude of the clock constants that appear in its clock conditions.* The reader may consider these values as external parameters, provided to the monitoring algorithm on initialisation.

**Computation model.** In our setting, both the clock constants and the time spans read from the input stream can be arbitrary real numbers. For this reason, we will use the real RAM model of computation, which is a model widely adopted in computational geometry and in data structures handling floating-point data. In this model, we have integer memory cells that can store integers and floating-point memory cells that can store real numbers. There are no bounds on the bit length or precision of the stored numbers. We assume that all basic arithmetic operations — negation, addition, subtraction, multiplication, and division — can be performed in unit time. Note that modulo arithmetics and rounding is not included in the model; in fact, we will not use multiplication and division on real numbers either.

We remark that if one assumes that all the numbers on input, such as clock constants or numbers appearing in the stream, are integers of bit length at most  $M$ , then in our algorithms we may rely only on integers of bit length  $\mathcal{O}(M + \log N)$ , where  $N$  is an upper bound on the total length of the stream that we expect to process. Therefore, in this case we may assume the standard word RAM model with words of bit length  $\mathcal{O}(M + \log N)$ .

### 3 The monitoring problem

In the algorithmic setting that we are going to consider, a timed automaton processes an arbitrarily long and possibly infinite timed word, hereafter called *stream*. It does so by consuming one element of the stream at a time (be it a letter or a time span), while updating its configuration. We would like to design a monitoring algorithm for an arbitrary stream processed by a fixed timed automaton  $\mathcal{A}$ , that readily answers the following query:

*Does the time automaton accept the current prefix of the stream?*

In answering the following query, the monitoring algorithm can make use of a suitable data structure, that is initialised beforehand and is maintained consistent along the computation. Of course the goal here is to do so efficiently, that is, in amortised constant time assuming that  $\mathcal{A}$  is fixed. We explain below what we mean precisely by this.

By saying that  $\mathcal{A}$  is *fixed* we mean that all its components (e.g. input alphabet, sets of states, etc.) are fixed, with the only exception of the constants that appear in the clock conditions of the transitions of  $\mathcal{A}$ . Such constants are instead treated as formal parameters, and their actual values, represented by a tuple  $\bar{c}$ , are provided as input to the algorithm upon initialisation. Note that the definition of the size  $|\mathcal{A}|$  of  $\mathcal{A}$  (cf. Section 2) reflects the above assumption, in the sense that it takes into account only the components of  $\mathcal{A}$  that are considered to be fixed.

Once  $\mathcal{A}$  is fixed, we can define the problem of *monitoring*  $\mathcal{A}$  as the problem of designing a suitable dynamic data structure that supports certain operations efficiently upon reading the input stream. Precisely, the data structure should support the following operations:

- **init**( $\bar{c}$ ), that initialises the data structure for the timed automaton  $\mathcal{A}$  with a tuple  $\bar{c}$  of clock constants;
- **accepted**(), that queries whether the prefix of the stream consumed up to the current moment is accepted by  $\mathcal{A}$ ;

- **read**( $e$ ), that consumes the next element  $e$  from the input stream, be it a letter from  $\Sigma$  or a time span from  $\mathbb{R}_{>0}$ , and updates the data structure accordingly.

The running time of each of these operations needs to be as low as possible, possibly bounded by a constant that is independent of the input stream, of the number of stream elements consumed so far, but also of the clock constants of  $\mathcal{A}$ . However, as we assume that  $\mathcal{A}$  is fixed, this constant running time may (and will) depend on  $|\mathcal{A}|$ . Formally, we say that a data structure *supports monitoring  $\mathcal{A}$  in constant time* if the first operation **init**( $\bar{c}$ ) and every subsequent execution of **accepted**() or **read**( $e$ ) take time  $\mathcal{O}(1)$ . Similarly, we say that a data structure *supports monitoring  $\mathcal{A}$  in amortised time  $f(n)$*  if the first operation **init**( $\bar{c}$ ) and every subsequent execution of **accepted**() take time  $\mathcal{O}(1)$ , whereas if the execution of the  $i$ -th **read**( $e$ ) operation takes time  $t_i$ , then for all  $n$  we have  $\frac{1}{n} \cdot \sum_{i=1}^n t_i = \mathcal{O}(f(n))$ . In particular, we have *amortised constant time* (resp. *amortised strongly sub-linear time*) if in the previous equation we can let  $f(n) = 1$  (resp.  $f(n) = n^{1-\delta}$  for any  $\delta > 0$ ).

We remark, for readers familiar with parametrised complexity, that it may be convenient to consider the monitoring problem as a problem parametrised by the size of the fixed automaton  $\mathcal{A}$ . Then the “constants” hidden in the  $\mathcal{O}(\cdot)$  notation are nothing but computable functions of the parameter  $|\mathcal{A}|$ . This inscribes our work into the setting of *dynamic parameterised algorithms*, which is still a under-developed area; see e.g. the discussion in [3].

**Results.** In this paper we describe a data structure that supports monitoring in either constant time or amortised constant time, depending on the form of the input stream. More precisely, we say that a stream  $w$  is *discrete* if its elements range over  $\Sigma \uplus \{1\}$ , that is, if all time spans coincide with the time unit 1. We will prove the following theorem:

► **Theorem 1.** *Fix a timed automaton  $\mathcal{A}$  with a single clock. There is a data structure that supports monitoring  $\mathcal{A}$*

- *in constant time on discrete streams,*
- *in amortised constant time on arbitrary streams.*

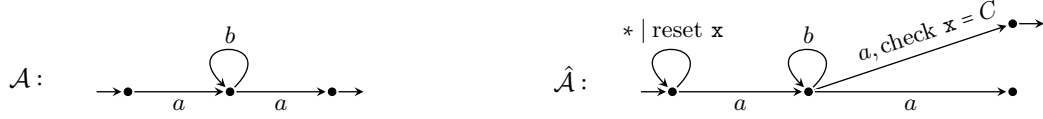
The proof of Theorem 1 is deferred to Section 4. We do not know whether this theorem can be generalised to timed automata with more than one clock without sacrificing the amortised constant time complexity of updates.

On the other hand, we are able to establish a negative result for a slightly more powerful model of timed automata, called timed automata with additive constraints (see e.g. [7]). Formally, a *timed automaton with additive constraints* is defined exactly as a timed automaton – that is, as a tuple  $\mathcal{A} = (\Sigma, Q, X, I, E, F)$  consisting of an input alphabet, a set of states, a set of clocks, etc. — but clock conditions are now allowed to satisfy an extended grammar obtained by adding new rules of the form  $(\sum_{x \in Z} x) \sim c$ , where  $Z \subseteq X$  and  $\sim \in \{<, >, =\}$ .

Our lower bound further relies on a complexity theoretical assumption related to the 3SUM problem. Recall that in the 3SUM problem we are given a set  $S$  of positive real numbers and the question is to determine whether there exist  $a, b, c \in S$  satisfying  $a + b = c$ . It is easy to solve the problem in time  $\mathcal{O}(n^2)$ , where  $n = |S|$ . Our lower bound is based on the hypothesis that the quadratic running time for 3SUM cannot be significantly improved.

► **Conjecture 2 (3SUM Conjecture).** *In the real RAM model, the 3SUM problem cannot be solved in strongly sub-quadratic time, that is, in time  $\mathcal{O}(n^{2-\delta})$  for any  $\delta > 0$ , where  $n$  is the number of numbers on the input.*

The 3SUM Conjecture is among the most popular hypotheses considered in computational geometry and fine-grained complexity theory; see e.g. an overview in [2, Appendix A]. It was introduced by Gajentaan and Overmars [11, 12] in a stronger form, which postulated the non-existence of *sub-quadratic* algorithms, that is, achieving running time  $o(n^2)$ . This



■ **Figure 1** Reducing the sliding window membership problem to the monitoring problem.

formulation was refuted by Grønlund and Pettie [17], who gave an algorithm for 3SUM with running time  $\mathcal{O}(n^2/(\log n/\log \log n)^{2/3})$  in the real RAM model, which can be improved to  $\mathcal{O}(n^2(\log \log n)^2/\log n)$  when randomisation is allowed. However, the existence of a strongly sub-quadratic algorithm is widely open and conjectured to be hard.

We can now state a lower bound for a variant of the monitoring problem:

► **Theorem 3.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton  $\mathcal{A}$  with additive constraints such that there is no data structure supporting monitoring  $\mathcal{A}$  in amortised strongly sub-linear time, and hence also not in amortised constant time.*

The proof of the above theorem is in the appendix, together with further discussion about the 3SUM Conjecture. Again, we do not know whether a negative result similar the above one also holds for plain timed automata (without additive constraints).

Before presenting the data structure for Theorem 1, we show an application of this result.

**An application example.** Here we consider only streams that are discrete. In fact, we will enforce a slightly more restricted form for such streams: we will assume that they belong to the language  $(\{1\} \cdot \Sigma)^\omega$ , namely, that the letters from  $\Sigma$  are interleaved by the time unit 1. Let  $\mathcal{A} = (\Sigma, Q, I, E, F)$  be a finite automaton and  $C > 0$  a number defining the width of a window. We consider the membership problem in the *sliding window model* (see, for instance [13]), that is, we process an arbitrary input  $w = a_1a_2a_3 \dots$  over  $\Sigma$ , consuming one letter at a time from left to right, while maintaining the answer to the following query: is the sequence of the last  $C$  consumed letters accepted by  $\mathcal{A}$  or not? Below, we explain how this problem can be reduced to our monitoring problem for timed automata and discrete streams.

We map  $w = a_1a_2a_3 \dots$  to a corresponding discrete stream  $\widehat{w} = 1a_11a_21a_3 \dots$ , and modify  $\mathcal{A}$  to obtain a corresponding timed automaton  $\widehat{\mathcal{A}}$ , as follows. We introduce a new state  $\widehat{q}$ , which will be the only final state in  $\widehat{\mathcal{A}}$ , and a clock  $x$ . We then replace every transition  $(q, a, q')$  of  $\mathcal{A}$  with the transition  $(q, a, \text{true}, q', \emptyset)$ . Note that these transitions have a vacuous clock condition, hence they are applicable in  $\widehat{\mathcal{A}}$  whenever the original transitions of  $\mathcal{A}$  are so. In addition, when the former transition  $(q, a, q')$  reaches a final state  $q' \in F$ , we also have a transition  $(q, a, x = C, \widehat{q}, \emptyset)$  in  $\widehat{\mathcal{A}}$ . Finally, we add looping transitions on the initial states that reset the clock, that is, transitions of the form  $(q, a, \text{true}, q, \{x\})$ , with  $q \in I$  and  $a \in \Sigma$ . Figure 1 shows the timed automaton  $\widehat{\mathcal{A}}$  corresponding to an automaton  $\mathcal{A}$  recognising  $ab^*a$ .

From the above construction it is clear that  $\widehat{\mathcal{A}}$  accepts a prefix  $1a_1 \dots 1a_n$  of  $\widehat{w}$  if and only if  $\mathcal{A}$  accepts the  $C$ -letter factor  $a_{n-C+1} \dots a_n$  of  $w$ . Thus, the acceptance problem for  $\mathcal{A}$  in the width- $C$  sliding window model is reduced to the monitoring problem for  $\widehat{\mathcal{A}}$  over the stream  $\widehat{w}$ . By Theorem 1, we know that there is a data structure that supports monitoring in constant time. This means that we can process one letter at a time from a word  $w$ , while answering in constant time whether  $\mathcal{A}$  accepts the sequence of the last  $C$  consumed letters. Note that the complexity here is independent of the choice of  $C$ .

We also remark that there is a similar reduction from the sliding window problem to the monitoring problem, where the timed automaton  $\widehat{\mathcal{A}}$  uses a clock condition, say  $x = 1$ , that is fixed prior to the choice of the window length, and where the input timed word is of the form  $\delta a_1 \delta a_2 \delta a_3 \dots$ , with  $\delta = \frac{1}{C}$ . This intuitively shows that, even if we fix the clock constants together with the timed automaton, the monitoring problem does not trivialise.



#### 4 Amortised constant-time data structure

We prove Theorem 1 by describing a data structure for monitoring a given timed automaton.

**Notation and definitions.** Let us fix, once and for all, a timed automaton  $\mathcal{A} = (\Sigma, Q, X, I, E, F)$  with a single clock  $x$ . By adding a non-accepting sink state, if necessary, we may assume that for every  $q \in Q$  and  $a \in \Sigma$ , some transition over letter  $a$  can be always applied at  $q$ . Since  $\mathcal{A}$  uses only one clock, every configuration of  $\mathcal{A}$  can be written simply as a pair  $(q, t)$ , where  $q \in Q$  is the state and  $t \in \mathbb{R}_{\geq 0}$  is the value of the clock  $x$ .

Note that by fixing  $\mathcal{A}$  we also fix the number of constants that appear in the clock conditions in  $E$ . Let us enumerate them as  $0 = C_0 < C_1 < \dots < C_k$ , where, without loss of generality, we assume that the clock constant  $C_0 = 0$  is always present. For simplicity we also let  $C_{k+1} = \infty$ , assuming that  $t < \infty$  for every  $t \in \mathbb{R}$ . As  $\mathcal{A}$  is fixed, we consider  $|\Sigma|$ ,  $|Q|$ , and  $k$  as fixed constants.

Consider now an arbitrary stream  $w \in (\Sigma \cup \mathbb{R}_{>0})^\omega$ . For every  $n \in \mathbb{N}$ , let  $w_n = w[1 \dots n]$  be the  $n$ -element prefix of  $w$ . Recall that  $w_n$  can be thought of as the stream prefix that is disclosed after  $n$  operations  $\text{read}(e)$ . We say that a configuration  $(q, t)$  is *active* at step  $n$  if there is a run of  $\mathcal{A}$  on  $w_n$  that starts in a configuration  $(q_0, 0)$  for some  $q_0 \in I$  and ends in  $(q, t)$ . We let  $K_n$  be the set of all configurations  $(q, t)$  that are active at step  $n$ .

**Partitioning the problem.** It is clear that the monitoring problem essentially boils down to designing an efficient data structure that maintains  $K_n$  under reading consecutive elements from the stream. This data structure should offer a query on whether  $K_n$  contains an accepting configuration. The main observation is that configurations with clock values behaving in the same way with respect to the clock constants  $C_1, \dots, C_k$  satisfy exactly the same clock conditions in  $E$ . Precisely, let us consider the partition of  $\mathbb{R}_{\geq 0}$  into intervals

$$\begin{aligned} J_0 &= [C_0, C_0], & J_1 &= (C_0, C_1), & J_2 &= [C_1, C_1], & J_3 &= (C_1, C_2), & J_4 &= [C_2, C_2], \\ J_5 &= (C_2, C_3), & \dots & J_{2k-1} &= (C_{k-1}, C_k), & J_{2k} &= [C_k, C_k], & J_{2k+1} &= (C_k, C_{k+1}). \end{aligned}$$

The following assertion is clear: for any two configurations  $(q, t)$ ,  $(q, t')$ , with  $t, t' \in J_i$  for some  $0 \leq i \leq 2k+1$ , exactly the same transitions are available in  $(q, t)$  as in  $(q, t')$ .

For  $n \in \mathbb{N}$  and  $i \in \{0, \dots, 2k+1\}$ , let  $K_n[i] = \{(q, t) \in K_n : t \in J_i\}$ . The idea is to maintain each set  $K_n[i]$  in a separate data structure. Each of these data structures follows the same design, which we call the *inner data structure* and which is outlined below.

**Inner data structure: an overview.** Every inner data structure is constructed from an interval  $J \in \{J_0, \dots, J_{2k+1}\}$ . We will denote it by  $\mathbb{D}[J]$ , or simply by  $\mathbb{D}[i]$  when  $J = J_i$ . Each structure  $\mathbb{D}[J]$  stores a set of configurations  $L$  satisfying the following invariant: all clock values of configurations in  $L$  belong to  $J$ . In the final design we will maintain the invariant that the set  $L$  stored by  $\mathbb{D}[i]$  at step  $n$  is equal to  $K_n[i]$ , but for the design of  $\mathbb{D}[J]$  it is easier to treat  $L$  as an arbitrary set of configurations with clock values in  $J$ .

The inner data structure should support the following methods:

- Method **init**( $J$ ) stores the interval  $J$  and initialises  $\mathbb{D}[J]$  by setting  $L = \emptyset$ .
- Method **accepted**() returns true or false, depending on whether  $L$  contains an accepting configuration, that is, a configuration  $(q, t)$  such that  $q \in F$ .
- Method **insert**( $q, t$ ) adds a configuration  $(q, t)$  to  $L$ . This method will be always applied with a promise that  $t \in J$  and  $t \leq t'$  for all configurations  $(q', t')$  already present in  $L$ .
- Method **updateTime**( $r$ ), where  $r \in \mathbb{R}_{>0}$ , increments the clock values of all configurations in  $L$  by  $r$ . All configurations whose clock values ceased to belong to  $J$  are removed from  $L$ , and they are returned by the method on output. This output is organised as a doubly linked list of configurations, sorted by non-decreasing clock values.

- Method `updateLetter(a)` updates  $L$  by applying to all configurations in  $L$  all possible transitions over the given letter  $a \in \Sigma$ . Precisely, the updated set  $L$  comprises all configurations  $(q, t)$  that can be obtained from configurations belonging to  $L$  before the update using transitions over  $a$  that do not reset the clock. All configurations  $(q, 0)$  which can be obtained from  $L$  using transitions over  $a$  that do reset the clock are not included in the updated set  $L$ , but are instead returned by the method as a doubly linked list. In Section 4.2 we will provide an efficient implementation of the inner data structure, which is encapsulated in the following lemma.

► **Lemma 4.** *For every  $J = J_i$ ,  $i \in \{0, 1, \dots, 2k + 1\}$ , the inner data structure  $\mathbb{D}[J]$  can be implemented so that methods `init()`, `accepted()`, `insert(·, ·)`, and `updateLetter(·)` run in constant time, while method `updateTime(·)` runs in time linear in the size of its output.*

We postpone the proof of Lemma 4 and we show now how to use it to prove Theorem 1. That is, we design an *outer data structure* that solves the monitoring problem for  $\mathcal{A}$ .

## 4.1 Outer data structure

The outer data structure consists of a list of data structures  $\mathbb{D}[0], \dots, \mathbb{D}[2k + 1]$ , where each  $\mathbb{D}[i]$  is a copy of the inner data structure constructed for the interval  $J_i$ . We will maintain the following invariant:

- I1.** After step  $n$ , for each  $i \in \{0, 1, \dots, 2k + 1\}$  the data structure  $\mathbb{D}[i]$  stores  $K_n[i]$ .

We first explain how the outer data structure implements the promised operations: initialisation, queries about the acceptance, and updates upon reading the next symbol from the stream  $w$ . Then we discuss the amortised complexity of the updates.

Initialisation. Given the tuple  $\bar{c} = (C_0, C_1, \dots, C_k)$  of clock constants of  $\mathcal{A}$ , with  $C_0 = 0$ , we initialise  $2k + 1$  copies  $\mathbb{D}[0], \dots, \mathbb{D}[2k + 1]$  of the inner data structure by calling method `init(J)` for each interval  $J$  among  $J_0, J_1, \dots, J_{2k+1}$ . Then, for each initial state  $q$ , we apply method `insert(q, 0)` on  $\mathbb{D}[0]$ . As  $K_0 = \{(q, 0) : q \in I\}$ , after this we have that Invariant (I1) holds for  $n = 0$ .

Query. We query all the data structures  $\mathbb{D}[0], \dots, \mathbb{D}[2k + 1]$  for the existence of accepting configurations using the `accepted()` method, and return the disjunction of the answers. The correctness follows directly from Invariant (I1).

Update by a time span. Suppose the next symbol read from the stream is a time span  $r \in \mathbb{R}_{>0}$ . We update the outer data structure as follows. First, we apply method `updateTime(r)` to each data structure  $\mathbb{D}[i]$ . This operation increments the clock values of all configurations stored in  $\mathbb{D}[i]$  by  $r$ , but may output a set of configurations whose clock values ceased to fit in the interval  $J_i$ . Recall that this set is organised as a doubly linked list of configurations, sorted by non-decreasing clock values; call this list  $S_i$ . Now, we need to insert each configuration  $(q, t)$  that appears on those lists into the appropriate data structure  $\mathbb{D}[j]$ , where  $j$  is such that  $t \in J_j$ . However, we have to be careful about the order of insertions: we process the lists  $S_{2k+1}, S_{2k}, \dots, S_0$  in this precise order, and each list  $S_i$  is processed from the end, that is, following the non-increasing order of clock values. When processing a configuration  $(q, t)$  from the list  $S_i$ , we find the index  $j > i$  such that  $t \in J_j$  and apply the method `insert(q, t)` on the structure  $\mathbb{D}[j]$ . In this way the condition required by the `insert` method — that  $t \leq t'$  for every configuration  $(q', t')$  currently stored in  $\mathbb{D}[j]$  — is satisfied. It is also easy to see that Invariant (I1) is preserved after the update.

Update by a letter. Suppose the next symbol read from the stream is a letter  $a \in \Sigma$ . We update the outer data structure as follows. First, we apply method `updateLetter(a)` to



each data structure  $\mathbb{D}[i]$ . This operation applies all possible transitions on letter  $a$  to all configurations stored in  $\mathbb{D}[i]$ , and outputs a list of configurations  $R_i$  where the clock got reset. Note that all these configurations have clock value 0, hence the length of the list  $R_i$  is at most  $|Q|$ , which is a constant. It now suffices to insert all the configurations  $(q, 0)$  appearing on all the lists  $R_i$  to the data structure  $\mathbb{D}[0]$  using method `insert`( $q, 0$ ). We may do this in any order, as the condition required by the `insert` method is trivially satisfied. Again, Invariant (I1) is clearly preserved after the update.

This concludes the implementation of the outer data structure. While the correctness is clear from the description, we are left with arguing that the time complexity is as promised.

Since  $|Q|$  and  $k$  are considered constants, it can be easily argued using Lemma 4 that under the implementation presented above, each of the following operations takes constant time: initialisation, a query about the acceptance, and an update by a letter. As for an update by a time span  $r \in \mathbb{R}_{>0}$ , by Lemma 4 the complexity of such an update is  $\mathcal{O}(\sum_{i=0}^{2k+1} |S_i|)$ , where  $S_0, \dots, S_{2k+1}$  are the sets returned by the applications of method `updateTime`( $r$ ) to data structures  $\mathbb{D}[0], \dots, \mathbb{D}[2k+1]$ , respectively. We need to argue that the amortised time complexity of all these updates is constant.

Consider the following definition: a clock value  $t \in \mathbb{R}_{\geq 0}$  is *active* at step  $n$  if  $K_n$  contains a configuration with clock value  $t$ . Observe that upon an update by a time span  $r \in \mathbb{R}_{>0}$ , the set of active clock values simply gets shifted by  $r$ , while upon an update by a letter  $a \in \Sigma$  it stays the same, except that possibly clock value 0 becomes active in addition. Since at step 0 the only active clock value is 0, we conclude that for every  $n \in \mathbb{N}$ , at most  $n+1$  active clock values may have appeared until step  $n$ . Now observe that since  $|Q|$  is considered a constant and there may be at most  $|Q|$  different active configurations with the same active clock value, the complexity of each update by a time span is proportional to the number of active clock values that change the interval  $J_i$  to which they belong, where we imagine that each active clock value is shifted by the time span. As every active clock value can change its interval at most  $2k+1 = \mathcal{O}(1)$  times, and the total number of active values that appear until step  $n$  is at most  $n+1$ , we conclude that the total time spent on updates by time spans throughout the first  $n$  steps is  $\mathcal{O}(n)$ . This means that the amortised time complexity is  $\mathcal{O}(1)$ .

Finally, note that in the case of discrete streams each set  $S_i$  consists of configurations with the same clock value, hence  $|S_i| \leq |Q| = \mathcal{O}(1)$  for all  $i \in \{0, \dots, 2k+1\}$ . Consequently, in this case the complexity of an update by a time span is also constant, without any amortisation.

This finishes the proof of Theorem 1, assuming Lemma 4. We prove the latter next.

## 4.2 Inner data structure

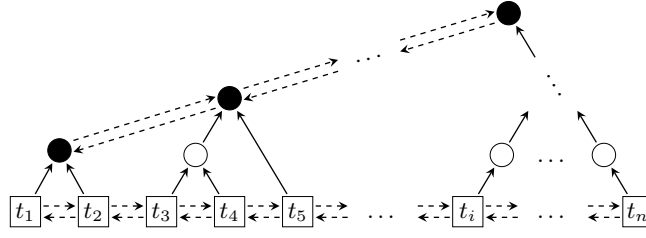
We now describe in detail the inner data structure  $\mathbb{D}[J]$  and prove Lemma 4. Let us fix an interval  $J \in \{J_0, \dots, J_{2k+1}\}$ . We will denote by  $L$  the set of configurations currently stored by the inner data structure  $\mathbb{D}[J]$ . It is convenient to represent  $L$  by a function  $\lambda: \mathbb{R}_{\geq 0} \rightarrow 2^Q$  defined by  $\lambda(t) = \{q \in Q : (q, t) \in L\}$ . We let  $\widehat{L}$  be the set of all clock values that are *active* in  $L$ , that is,  $\widehat{L}$  comprises all  $t \in \mathbb{R}_{\geq 0}$  such that  $\lambda(t) \neq \emptyset$ . Recall that we assume that  $\widehat{L} \subseteq J$ .

**Description of the structure.** In short, the data structure  $\mathbb{D}[J]$  consists of three elements:

- The *clock*, denoted  $y$ , is a non-negative real that stores the total time elapsed so far.
- The *list*, denoted  $l$ , stores the set of active clock values  $\widehat{L}$ .
- The *forest*, denoted  $f$ , is built on top of the elements of  $l$  and describes the function  $\lambda$ .

We describe the list and the forest in more details (the reader can refer to Figure 2).

The list. The list  $l$  encodes the clock values present in  $\widehat{L}$ , sorted in the increasing order and organised into a doubly linked list. Each node  $\alpha$  on  $l$  is a record consisting of:



■ **Figure 2** The inner data structure.

- $\text{next}(\alpha)$ : a pointer to the next node on the list;
- $\text{prev}(\alpha)$ : a pointer to the previous node on the list; and
- $\text{timestamp}(\alpha) \in \mathbb{R}$ : the *timestamp* of the node.

As usual, the data structure stores  $\mathbf{l}$  by maintaining pointers to the first and last node.

The clock value represented by a node  $\alpha$  on  $\mathbf{l}$  (represented by a square in Figure 2) is equal to  $\text{clock}(\alpha) = \mathbf{y} - \text{timestamp}(\alpha)$ ; this will always be a non-negative real. Thus, intuitively, the timestamp is essentially the total elapsed time recorded since the last reset of the clock. Note that this implementation allows for a simultaneous increment of  $\text{clock}(\alpha)$  for all nodes  $\alpha$  on  $\mathbf{l}$  in constant time: it suffices to simply increment  $\mathbf{y}$ .

*The forest.* The forest  $\mathbf{f}$  represents the assignment that maps elements  $t \in \widehat{L}$ , encoded in  $\mathbf{l}$ , to respective sets of control states  $\lambda(t)$ . It has a standard form of a rooted forest, where every node may have arbitrarily many children, and these children are unordered. Every node  $\gamma$  of  $\mathbf{f}$  (represented by a circle in Figure 2) is a record containing:

- $\text{parent}(\gamma)$ : a pointer to the parent of  $\gamma$ ; and
- $\#\text{children}(\gamma)$ : an integer equal to the number of children of  $\gamma$ .

The leaves of the forest will always coincide with the nodes on the list  $\mathbf{l}$ . Thus, we may simply augment the records stored for the nodes on  $\mathbf{l}$  by adding the  $\text{parent}(\cdot)$  pointer, and treat them as nodes of the forest  $\mathbf{f}$  at the same time. The counter  $\#\text{children}(\cdot)$  would always be equal to 0 for those nodes, so we may omit it.

The *roots* of the forest (represented by the black circles in Figure 2) are the nodes  $\beta$  with no parent, i.e.  $\text{parent}(\beta) = \perp$ . We will maintain the invariant that no root is a leaf in  $\mathbf{f}$ , that is, every root has at least one child. In the data structure we will maintain a doubly linked list containing all the roots of  $\mathbf{f}$ . This list will be denoted  $\mathbf{r}$ , and again it will be stored by pointers to its first and last element. Thus, the records of the roots of  $\mathbf{f}$  are augmented by  $\text{next}(\cdot)$  and  $\text{prev}(\cdot)$  pointers describing the structure of  $\mathbf{r}$ , with the usual meaning. In addition to this, every root  $\beta$  of  $\mathbf{f}$  carries two additional values:

- $\text{states}(\beta) \subseteq Q$ : a non-empty subset of control states for which  $\beta$  is responsible; and
- $\text{rank}(\beta)$ : an integer from the set  $\{1, 2, \dots, 2^{|Q|}\}$ .

We will maintain two invariants about these values. First, the sets  $\text{states}(\beta)$  and the ranks  $\text{rank}(\beta)$  should be pairwise different for distinct roots  $\beta$  of  $\mathbf{f}$ . Note that this means that  $\mathbf{f}$  always has at most  $2^{|Q|} - 1 = \mathcal{O}(1)$  roots. Second, for every root  $\beta$ , the *tree rooted at  $\beta$*  — which is the tree consisting of  $\beta$  and all its descendants in  $\mathbf{f}$  — has depth at most  $\text{rank}(\beta)$ . Here, the *depth* of a forest is the maximum number of edges on a path from a leaf to a root, minus 1. Note that this implies that the depth of the forest  $\mathbf{f}$  is bounded by  $2^{|Q|} = \mathcal{O}(1)$ .

The function  $\lambda$  is then represented as follows. For every node  $\alpha$  on  $\mathbf{l}$ , let  $\text{root}(\alpha)$  be the root of the tree of  $\mathbf{f}$  that contains  $\alpha$ . Then denoting  $t = \text{clock}(\alpha)$ , we have  $\lambda(t) = \text{states}(\text{root}(\alpha))$ . Note that the invariant stated above imply that from every leaf  $\alpha$  of  $\mathbf{f}$ ,  $\text{root}(\alpha)$  can be computed from  $\alpha$  by following the  $\text{parent}(\cdot)$  pointer at most  $2^{|Q|} = \mathcal{O}(1)$

times. Hence, given  $t \in \widehat{L}$  together with a node  $\alpha$  on  $\mathbf{l}$  satisfying  $t = \text{clock}(\alpha)$ , we can compute  $\lambda(t)$  in  $\mathcal{O}(1)$  time.

**Invariants.** For convenience, we gather all the invariants maintained by the inner data structure which we mentioned before:

- 12. For each node  $\alpha$  on  $\mathbf{l}$ , the value  $\text{clock}(\alpha) = y - \text{timestamp}(\alpha)$  belongs to  $J$ .
- 13. The nodes on  $\mathbf{l}$  are sorted by increasing clock values, or equally by decreasing timestamps. That is,  $\text{timestamp}(\alpha) > \text{timestamp}(\text{next}(\alpha))$  for every non-last node  $\alpha$  on  $\mathbf{l}$ .
- 14. Every root of  $\mathbf{f}$  has at least one child, and the leaves of  $\mathbf{f}$  are exactly all the nodes on  $\mathbf{l}$ .
- 15. The roots of  $\mathbf{f}$  carry pairwise different, non-empty sets of control states, and they have pairwise different ranks. Moreover, all the ranks belong to the set  $\{1, 2, \dots, 2^{|Q|}\}$ .
- 16. For every root  $\beta$  of  $\mathbf{f}$ , the depth of the tree rooted at  $\beta$  is at most  $\text{rank}(\beta)$ .

**Implementation.** We now show how to implement the methods  $\text{init}(J)$ ,  $\text{accepted}()$ ,  $\text{insert}(q, t)$ ,  $\text{updateTime}(r)$ , and  $\text{updateLetter}(a)$  in the data structure. Recall that all these methods should work in constant time, with the exception of  $\text{updateTime}(r)$  which is allowed to work in time linear in its output. The description of each method is supplied by a running time analysis and an argumentation of the correctness, which includes a discussion on why the invariants stated above are maintained.

Removing nodes. Before we proceed to the description of the required methods, we briefly discuss an auxiliary procedure of removing a node from the list  $\mathbf{l}$  and from the forest  $\mathbf{f}$ , as this procedure will be used several times. Suppose we are given a node  $\alpha$  on the list  $\mathbf{l}$  and we would like to remove it, which corresponds to removing from  $L$  all configurations  $(q, t)$  where  $t = \text{clock}(\alpha)$  and  $q \in \lambda(t)$ . We can remove  $\alpha$  from  $\mathbf{l}$  in the usual way. Then we remove  $\alpha$  from  $\mathbf{f}$  as follows. First, we decrement the counter of children in the parent of  $\alpha$ . If this counter stays positive then there is nothing more to do. Otherwise, we need to remove the parent of  $\alpha$  as well, and accordingly decrement the counter of children in the grandparent of  $\alpha$ . This can again trigger removal of the grandparent and so on. If eventually we need to remove a root of  $\mathbf{f}$ , we also remove it from the list  $\mathbf{r}$  in the usual way. Note that since by Invariants (I5) and (I6), the depth of  $\mathbf{f}$  is bounded by a constant, the total number of removals is bounded by a constant as well, and the whole procedure can be performed in constant time. It is straightforward to verify that all the invariants are maintained.

Initialization. The  $\text{init}(J)$  method stores the interval  $J$ , that defines the range of clock values that could be represented in the data structure. It also sets  $y = 0$  and initialises  $\mathbf{l}$  and  $\mathbf{r}$  as empty lists. The correctness and the running time are clear.

Acceptance query. The  $\text{accepted}()$  method is implemented as follows. We iterate through the list  $\mathbf{r}$  to check whether there exists a root  $\beta$  of  $\mathbf{f}$  such that  $\text{states}(\mathbf{f})$  contains any accepting state, say  $q$ . If this is the case, then by Invariant (I4) there is a node  $\alpha$  on  $\mathbf{l}$  satisfying  $\text{root}(\alpha) = \beta$ , hence  $(q, t)$  is an accepting configuration that belongs to  $L$ , where  $t = \text{clock}(\alpha)$ . So we may return a positive answer from the query. Otherwise, all configurations in  $L$  have non-accepting states, and we may return a negative answer. Note that since by Invariant (I5) the list  $\mathbf{r}$  has bounded length, the above procedure works in constant time.

Insertion. We now implement the method  $\text{insert}(q, t)$ , where  $(q, t)$  is a configuration. Recall that when this method is executed, we have a promise that  $t \in J$  and  $t \leq t'$  for all configurations  $(q', t')$  that are currently present in  $\mathbb{D}[J]$ .

Let  $\alpha$  be the first node on the list  $\mathbf{l}$  and let  $t' = \text{clock}(\alpha)$ . By the promise, we have  $t \leq t'$ . We distinguish two cases: either  $t < t'$  or  $t = t'$ . The former case also encompasses the corner situation when  $\mathbf{l}$  is empty.

When  $t < t'$  or  $\mathbf{l}$  is empty, the new configuration  $(q, t)$  gives rise to a new active clock value  $t$ . Therefore, we create a new list node  $\alpha_0$  and insert it at the front of the list  $\mathbf{l}$ . We set the timestamp as  $\text{timestamp}(\alpha_0) = y - t$ , so that the node correctly represents the clock value  $t$ . It is clear that Invariants (I2) and (I3) are thus satisfied.

Next, we need to insert the new node  $\alpha_0$  to the forest  $\mathbf{f}$ . We iterate through the list  $\mathbf{r}$  in search for a root  $\beta$  that satisfies  $\text{states}(\beta) = \{q\}$ . In case there is one, we simply set  $\text{parent}(\alpha_0) = \beta$  and increment  $\#\text{children}(\beta)$ . Otherwise, we construct a new root  $\beta_0$  with  $\text{states}(\beta_0) = \{q\}$  and  $\#\text{children}(\beta_0) = 1$ , insert it at the front of the list  $\mathbf{r}$ , and set  $\text{parent}(\alpha_0) = \beta_0$ . To determine the rank of  $\beta_0$ , we find the smallest integer  $k \in \{1, \dots, 2^{|Q|}\}$  that is *not* used as the rank of any other root of  $\mathbf{f}$ . Observe that, by Invariant (I5), the forest  $\mathbf{f}$  has at most  $2^{|Q|} - 1$  roots, so there is always such a number  $k$ , and it can be found in constant time by inspecting the list  $\mathbf{r}$ . We then set  $\text{rank}(\beta_0) = k$ . It is clear from the description that this operation can be performed in constant time, and that Invariants (I4), (I5), and (I6) are maintained. For the last one, observe that the new leaf  $\alpha_0$  is attached directly under a root of  $\mathbf{f}$ , so no tree in  $\mathbf{f}$  existing before the insertion could have increased its depth.

We are left with the case when  $t = t'$ . We first compute the set  $X$  equal to  $\lambda(t)$  before the insertion: it suffices to find  $\text{root}(\alpha)$  in constant time and read  $X = \text{states}(\text{root}(\alpha))$ . If  $q \in X$  then the configuration  $(q, t)$  is already present in  $L$ , so there is nothing to do. Otherwise, we need to update the data structure so that  $\lambda(t)$  is equal to  $X \cup \{q\}$  instead of  $X$ . Consequently, we remove the node  $\alpha$  from  $\mathbf{l}$  and from  $\mathbf{f}$ , using the operation described earlier, and we insert a new node  $\alpha'$  at the front of  $\mathbf{l}$ , with the same timestamp as that of  $\alpha$ . Thus,  $\text{clock}(\alpha') = t$ . We next insert the new node  $\alpha'$  to the forest  $\mathbf{f}$  using the same procedure as described in the previous paragraph, but applied to the state set  $X \cup \{q\}$  instead of  $\{q\}$ . Again, it is clear that these operations can be performed in constant time, and the same argumentation shows that all the invariants are maintained.

Update by a time span. Next, we implement the method  $\text{updateTime}(r)$ , where  $r \in \mathbb{R}_{>0}$ .

First, we increment  $y$  by  $r$ . Thus, for every node  $\alpha$  in the list  $\mathbf{l}$  the value  $\text{clock}(\alpha)$  got incremented by  $r$ . However, the Invariant (I2) may have ceased to hold, for some active clock values could have been shifted outside of the interval  $J$ . The configurations with these clock values should be removed from the data structure and their list should be returned as the output of the method.

We extract these configurations as follows. Construct an initially empty list of configuration  $\mathbf{lret}$ , on which we shall build the output. Iterate through the list  $\mathbf{l}$ , starting from its back. For each consecutive node  $\alpha$ , compute  $t = \text{clock}(\alpha)$ . If  $t \in J$ , then break the iteration and return  $\mathbf{lret}$ , as there are no more configurations to remove. Otherwise, find  $\text{root}(\alpha)$  in constant time, read  $\lambda(t) = \text{states}(\text{root}(\alpha))$ , and add at the front of  $\mathbf{lret}$  all configurations  $(q, t)$  for  $q \in \lambda(t)$ , in any order. Then remove  $\alpha$  from the list  $\mathbf{l}$  and from the forest  $\mathbf{f}$ , and proceed to the previous node in  $\mathbf{l}$  (if there is none, finish the iteration).

By Invariant (I3), it is clear that in this way we remove from  $\mathbb{D}[J]$  exactly all the configurations whose clock values got shifted outside of  $J$ , hence Invariants (I2) and (I3) are maintained. As the forest structure was influenced only by removals, Invariants (I4), (I5), and (I6) are maintained as well. Also note that the configurations on the output list  $\mathbf{lret}$  are ordered by non-decreasing clock values, as was required.

As for the time complexity, recall that by Invariant (I5), for every removed node  $\alpha$  the set  $\text{states}(\text{root}(\alpha))$  is non-empty and has size at most  $|Q|$ . Hence, with every removed node  $\alpha$  we add to  $\mathbf{lret}$  between 1 and  $|Q| = \mathcal{O}(1)$  new configurations. As the time complexity of the procedure is bounded linearly in the number of nodes that we remove from  $\mathbf{l}$ , it is also bounded linearly in the number of configurations that appear in the output list  $\mathbf{lret}$ .

*Update by a letter.* We proceed to the method `updateLetter(a)`, where  $a \in \Sigma$ . As argued before, every clock condition appearing in  $\mathcal{A}$  is either true for all clock values in  $J$ , or false for all clock values in  $J$ . For every subset of states  $X \subseteq Q$ , let  $\Phi(X)$  be the set of all states  $q$  such that there is a transition  $(p, a, q, \gamma, \emptyset)$  in  $E$  for some  $p \in X$  and clock condition  $\gamma$  that is true in  $J$ . In other words,  $\Phi(X)$  comprises states reachable from the states of  $X$  by non-resetting transitions over  $a$  that are available for clock values in  $J$ . We define  $\Psi(X)$  in a similar way, but for resetting transitions over  $a$  that are available for clock values in  $J$ .

First, we compute the output of the method, which should be simply  $\{(q, 0) : q \in \Psi(X)\}$ , where  $X$  is the set of all states appearing in the configurations of  $L$ . Observe that, by Invariant (I4),  $X$  can be computed in constant time by iterating through the list  $\mathbf{r}$  and computing the union of sets `states( $\beta$ )` for roots  $\beta$  appearing on it. Thus, the output of the method can be computed in constant time.

Second, we need to update the values of function  $\lambda$  by applying all possible non-resetting transitions over  $a$ . This can be done by iterating through the list  $\mathbf{r}$  and, for each root  $\beta$  appearing on it, substituting `states( $\beta$ )` with  $\Phi(\text{states}(\beta))$ . Note that since we assumed that for every state  $q$ , some transition over  $a$  is always available at  $q$ , it follows that  $\Phi$  maps non-empty sets of states to non-empty sets of states. Hence, after this substitution the roots of  $\mathbf{f}$  will still be assigned non-empty sets of states. However, Invariant (I5) may cease to hold, as some roots may now be assigned the same set of states.

We fix this as follows. For every root  $\beta$  of  $\mathbf{f}$ , inspect the list  $\mathbf{r}$  and find the root  $\beta'$  that has the largest rank among those satisfying `states( $\beta$ )` = `states( $\beta'$ )`. If  $\beta = \beta'$ , then do nothing. Otherwise, turn  $\beta$  into a non-root node of  $\mathbf{f}$ , remove it from the list  $\mathbf{r}$ , set `parent( $\beta$ )` =  $\beta'$ , and increment `#children( $\beta'$ )` by one. Note that after applying this modification, the function  $\lambda$  stored in the data structure stays the same, while Invariant (I5) becomes satisfied.

As for the other invariants, the satisfaction of Invariants (I2), (I3), and (I4) after the update is clear. However, we need to be careful about Invariant (I6), as we might have substantially modified the structure of the forest  $\mathbf{f}$ . Observe that each modification of  $\mathbf{f}$  that we applied boils down to attaching a tree with a root of some rank  $i$  as a child of a tree with a root of some rank  $j > i$ . By Invariant (I6), the former tree has depth at most  $i$ , which is bounded from above by  $j - 1$ . Thus, after the attachment, the depth of the latter tree cannot become larger than  $j$ . We conclude that Invariant (I6) is maintained as well.

Finally, note that since the number of roots of  $\mathbf{f}$  is always bounded by a constant, all the operations described above can be performed in constant time.

We have implemented all the required methods within the claimed running time bounds. This concludes the proofs of Lemma 4 and of Theorem 1.

## 5 Conclusion

We have considered a monitoring problem for streams processed by a timed automaton, which consists of readily answering the membership query “*Does the time automaton accept the current prefix of the stream?*”. We have designed a suitable data structure that solves the monitoring problem for a one-clock timed automaton in amortised constant time assuming that the timed automaton is fixed.

We leave as an open question whether our complexity result can be strengthened, for instance, by devising amortised constant time monitoring algorithms for timed automata with multiple clocks. Concerning the latter question, we proved that, assuming the 3SUM Conjecture, this is not possible for timed automata enhanced with additive clock constraints.

---

References

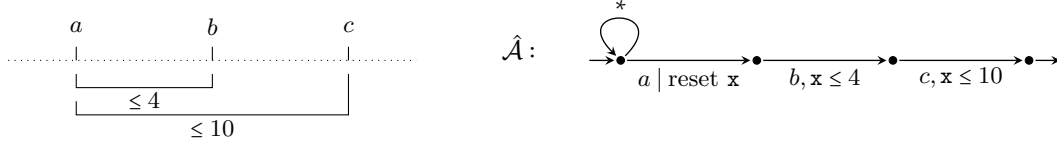
---

- 1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 434–443. IEEE Computer Society, 2014. URL: <https://doi.org/10.1109/FOCS.2014.53>, doi:10.1109/FOCS.2014.53.
- 2 Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. Matching triangles and basing hardness on an extremely popular conjecture. *SIAM J. Comput.*, 47(3):1098–1122, 2018. URL: <https://doi.org/10.1137/15M1050987>, doi:10.1137/15M1050987.
- 3 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 41:1–41:16. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2017. URL: <https://doi.org/10.4230/LIPIcs.ICALP.2017.41>, doi:10.4230/LIPIcs.ICALP.2017.41.
- 4 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. URL: [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8), doi:10.1016/0304-3975(94)90010-8.
- 5 Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, 2002.
- 6 Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. *Theoretical Computer Science*, 494:13–23, 2013.
- 7 Béatrice Bérard and Catherine Dufourd. Timed automata and additive clock constraints. *Inf. Process. Lett.*, 75(1-2):1–7, 2000. URL: [https://doi.org/10.1016/S0020-0190\(00\)00075-2](https://doi.org/10.1016/S0020-0190(00)00075-2), doi:10.1016/S0020-0190(00)00075-2.
- 8 Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318, 2017.
- 9 Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):1–62, 2012.
- 10 Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.
- 11 Anka Gajentaan and Mark H. Overmars. On a class of  $O(n^2)$  problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995. URL: [https://doi.org/10.1016/0925-7721\(95\)00022-2](https://doi.org/10.1016/0925-7721(95)00022-2), doi:10.1016/0925-7721(95)00022-2.
- 12 Anka Gajentaan and Mark H. Overmars. On a class of  $O(n^2)$  problems in computational geometry. *Comput. Geom.*, 45(4):140–152, 2012. URL: <https://doi.org/10.1016/j.comgeo.2011.11.006>, doi:10.1016/j.comgeo.2011.11.006.
- 13 Moses Ganardi. *Language recognition in the sliding window model*. PhD thesis, Universität Siegen, 2019. URL: <https://dspace.ub.uni-siegen.de/handle/ubsi/1523>, doi:http://dx.doi.org/10.25819/ubsi/464.
- 14 Moses Ganardi, Danny HucKe, Daniel König, Markus Lohrey, and Konstantinos Mamouras. Automata theory on sliding windows. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, pages 31:1–31:14, 2018.
- 15 Moses Ganardi, Danny HucKe, and Markus Lohrey. Querying regular languages over sliding windows. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, pages 18:1–18:14, 2016.
- 16 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *22nd International Conference on Database Theory (ICDT 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- 17 Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *J. ACM*, 65(4):22:1–22:25, 2018. URL: <https://doi.org/10.1145/3185378>, doi:10.1145/3185378.



- 18 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.
- 19 Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Efficient query processing for dynamically changing datasets. *ACM SIGMOD Record*, 48(1):33–40, 2019.
- 20 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1272–1287. SIAM, 2016. URL: <https://doi.org/10.1137/1.9781611974331.ch89>, doi:10.1137/1.9781611974331.ch89.
- 21 Philip M Lewis, Richard Edwin Stearns, and Juris Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*, pages 191–202. IEEE, 1965.
- 22 Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014.
- 23 Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 603–610. ACM, 2010. URL: <https://doi.org/10.1145/1806689.1806772>, doi:10.1145/1806689.1806772.
- 24 Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 66–77, 2017.

$$\varphi = ((a;b) \text{ WITHIN } 4); c \text{ WITHIN } 10$$



■ **Figure 3** Translation of a CEL expression into an equivalent single-clock timed automaton.

## A Other application examples for the monitoring problem

► **Example 5.** Here we consider a scenario from *complex event processing* (CEP), with a specification language called CEL and defined by the following grammar [16]:

$$\varphi := a \mid \varphi; \varphi \mid \varphi \text{ WITHIN } t$$

where  $a \in \Sigma$  and  $t \in \mathbb{N}$ . A word  $w = a_1 a_2 \dots a_n \in \Sigma^*$  *matches* an expression  $\varphi$  from the above grammar, denoted  $w \models \varphi$ , if one of the following cases holds:

- $\varphi = a_n$ ,
- $\varphi = \varphi_1; \varphi_2$ ,  $w = w_1 \cdot w_2$ ,  $w_1 \models \varphi_1$  and  $w_2 \models \varphi_2$ ,
- $\varphi = \varphi' \text{ WITHIN } t$  and  $a_m \dots a_n \models \varphi'$ , where  $m = \max\{1, n - t\}$ .

Given a word  $w = a_1 a_2 \dots$  and an expression  $\varphi$ , we would like to read  $w$  sequentially, as in a stream, and decide, at each position  $n = 1, 2, \dots$ , whether the prefix  $w_n = a_1 \dots a_n$  matches a fixed expression  $\varphi$ . One can reduce this latter problem to our monitoring problem for timed automata, by using a discrete timed word  $\hat{w} = 1a_1 1a_2 1 \dots$  as before and by translating the expression  $\varphi$  into an appropriate timed automaton. We omit the straightforward details of the translation of a CEL expression to an equivalent timed automaton, and we only remark that every occurrence of the **WITHIN** operator in an expression corresponds to a condition on a specific clock in the equivalent timed automaton. This means that, in general, the translation may require a timed automaton with multiple clocks. However, there are simple cases (which we do not characterise here) where, even in the presence of nested **WITHIN** operators, one can construct an equivalent timed automaton with a single clock. For example, consider the expression  $\varphi = ((a;b) \text{ WITHIN } 4); c \text{ WITHIN } 10$ , which describes a sequence containing three (possibly not contiguous) events  $a, b, c$ , with  $a$  and  $b$  at distance at most 4 and  $a$  and  $c$  at distance at most 10. Figure 3 shows a single-clock timed automaton that is equivalent to  $\varphi$ , in the sense that it accepts a timed word of the form  $1a_1 1a_2 1 \dots 1a_n$  if and only if  $a_1 a_2 \dots a_n \models \varphi$ . In this case one can validate any input stream against the expression  $\varphi$  in time that is constant per input letter, by simply reducing to our monitoring problem for single-clock timed automata and discrete timed words.

► **Example 6.** Consider a list  $k_1, \dots, k_n \in \mathbb{N}$  of numbers whose greatest common divisor is 1. The Frobenius coin problem deals the following question: what is the maximum integer number  $h$  that cannot be expressed as a sum of multiples of  $k_i$ 's? We consider a dynamic variant of this problem, where we are given  $k_1, \dots, k_n \in \mathbb{N}$  on input, and then for consecutive numbers  $h = 1, 2, 3, \dots$  we should answer whether  $h$  can be expressed as a sum of multiples of  $k_i$ 's. The time complexity should be constant per each choice of  $h$ , assuming that the numbers  $h = 1, 2, 3, \dots$  arrive in this precise ordering.

The idea is to construct a timed automaton  $\mathcal{A}$  that consumes longer and longer prefixes  $(1a)^h$  of a potentially infinite stream, while checking whether the total elapsed time  $h$  can be

written as  $\sum_{i=1}^n k_i \cdot h_i$  for some  $h_1, \dots, h_n \in \mathbb{N}$ , in which case the prefix is accepted. Formally, we let  $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ , where  $\Sigma = \{a\}$ ,  $Q = \{p, q\}$ ,  $X = \{x\}$ ,  $I = \{p\}$ ,  $F = \{q\}$ , and  $E$  contains the following transitions:  $(p, a, \text{true}, p, \emptyset)$ ,  $(p, a, \varphi, q, \{x\})$ , and  $(q, a, \text{true}, p, \emptyset)$ , with  $\varphi = (x = k_1) \vee \dots \vee (x = k_n)$ . Intuitively, the first transition lets some arbitrary amount of time to pass, until the second transition is enabled, which happens when the clock value is any of the constants  $k_1, \dots, k_n$ . Every time the second transition is triggered, the clock is reset and either the word terminates and is accepted, or a subsequent transition brings the control back to the initial state. It is easy to see that an input word of the form  $(1a)^h$  is accepted by this timed automaton if and only if  $h = \sum_{i=1}^n k_i \cdot h_i$  for some  $h_1, \dots, h_n \in \mathbb{N}$ .

## B 3SUM Conjecture and lower bound for the monitoring problem

In this section, we prove a complexity lower bound for a variant of our monitoring problem. Ideally, we would like to prove that there is a timed automaton with two clocks for which monitoring in amortised constant time is not possible. This would imply that our result (Theorem 1) for monitoring a single-clock timed automaton cannot be generalised to the multiple-clock setting. We are not able to establish optimality in this sense. We can however prove a result along the same line, by considering timed automata extended with additive constraints, that is, having clock conditions of the form  $(\sum_{x \in Z} x) \sim c$ . Our lower bound is based on the 3SUM Conjecture, which we restate below for convenience.

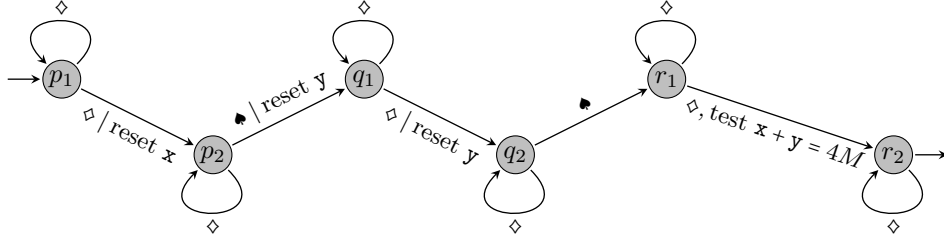
► **Conjecture 7 (3SUM Conjecture).** *In the real RAM model, the 3SUM problem cannot be solved in strongly sub-quadratic time, that is, in time  $\mathcal{O}(n^{2-\delta})$  for any  $\delta > 0$ , where  $n$  is the number of numbers on the input.*

Recall that in the 3SUM problem we are given a set  $S$  of positive real numbers and the question is to determine whether there exist  $a, b, c \in S$  satisfying  $a + b = c$ . We remark that the original phrasing of the conjecture allows non-positive numbers on input and asks for  $a, b, c \in S$  such that  $a + b + c = 0$ . It is easy to reduce this standard formulation to our setting, for example by replacing  $S$  with  $S' = \{3M + x : x \in S\} \cup \{6M - x : x \in S\}$ , where  $M$  is any real satisfying  $M > |a|$  for all  $a \in S$ .

The 3SUM Conjecture has received significant attention in the recent years, as it was realised that it can be used as a base for tight complexity lower bounds for a variety of discrete graph problems, including questions about efficient dynamic data structures [1, 3, 20, 23]. In this setting, it is common to assume the integer formulation of the conjecture: there exists  $d \in \mathbb{N}$  such that the 3SUM problem where the input numbers are integers from the range  $[-n^d, n^d]$  cannot be solved in strongly sub-quadratic time, assuming the word RAM model with words of bit length  $\mathcal{O}(\log n)$ . It is straightforward to verify that the construction we are going to present in this section can be turned into an analogous lower bound assuming the integer formulation of the 3SUM Conjecture. For this, we would need to amend the formulation of the monitoring problem by assuming that the input stream is expected to have total length at most  $N$ , the clock constants and the time spans in the stream are integers of bit length at most  $M$ , and the data structure solving the monitoring problem should work in the word RAM model with words of bit length  $\mathcal{O}(M + \log N)$ .

We now prove Theorem 3, restated below for convenience, which provides a lower bound for monitoring two-clock timed automata with additive constraints under the 3SUM Conjecture.

► **Theorem 3.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton  $\mathcal{A}$  with additive constraints such that there is no data structure supporting monitoring  $\mathcal{A}$  in amortised strongly sub-linear time, and hence also not in amortised constant time.*



■ **Figure 4** Timed automaton for reducing 3SUM.

Our approach is similar in spirit to the other lower bounds on dynamic problems, which we mentioned above [1, 3, 20, 23]. We first prove 3SUM-hardness of deciding acceptance by a timed automaton with additive constraints in the static setting. We then show that any data structure that supports monitoring in amortised strongly sub-linear time would violate the 3SUM-hardness of the former static acceptance problem, thus proving Theorem 3.

The postulated hardness of the static problem is captured by the following lemma.

► **Lemma 8.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton  $\mathcal{A}$  with additive constraints for which there is no algorithm that, given a timed word  $w \in (\Sigma \uplus \mathbb{R}_{>0})^*$  as input, where  $\Sigma$  is a two-letter alphabet, decides whether  $\mathcal{A}$  accepts  $w$  in time  $\mathcal{O}(n^{2-\delta})$  for any  $\delta > 0$ .*

**Proof.** We construct a two-clock timed automaton  $\mathcal{A}$  with additive constraints and an algorithm that given a set  $S$  of  $n$  positive reals, outputs a word  $w \in (\Sigma \uplus \mathbb{R}_{>0})^*$  such that  $w$  is accepted by  $\mathcal{A}$  if and only if there are  $a, b, c \in S$  satisfying  $a + b = c$ . We find it more convenient to first present the construction of  $w$  from  $S$ . Then we present the automaton  $\mathcal{A}$  and analyse its runs on  $w$ .

Let  $M = 1 + \max_{s \in S} |s|$ . By sorting  $S$  we may assume that  $S = \{s_1, s_2, \dots, s_n\}$ , where  $0 < s_1 < \dots < s_n < M$ . We set  $\Sigma = \{\diamond, \spadesuit\}$ . The word is defined as  $w = u \spadesuit u \spadesuit v$ , where

$$\begin{aligned} u &= 2(M - s_n) \diamond 2(s_n - s_{n-1}) \diamond 2(s_{n-1} - s_{n-2}) \diamond \dots \diamond 2(s_2 - s_1) \diamond 2(s_1 - 0); \\ v &= (M - s_n) \diamond (s_n - s_{n-1}) \diamond (s_{n-1} - s_{n-2}) \diamond \dots \diamond (s_2 - s_1) \diamond. \end{aligned}$$

Note that  $w$  can be constructed from  $S$  in time  $\mathcal{O}(n \log n)$ . Intuitively, the factors  $u$ ,  $u$ , and  $v$  above are responsible for the choice of  $a$ ,  $b$ , and  $c$ , respectively. We now describe a timed automaton  $\mathcal{A}$  that accepts  $w$  if and only if  $a + b = c$ .

The automaton  $\mathcal{A}$  is depicted in Figure 4. It uses two clocks, named  $x$  and  $y$ . Note that all the transitions have trivial (always true) clock conditions, apart from the transition from  $r_1$  to  $r_2$ , where we check that the sum of clock values is equal to  $4M$ . The only initial state is  $p_1$ , the only accepting state is  $r_2$ .

We now analyse the runs of  $\mathcal{A}$  on  $w$ , with the goal of showing that  $\mathcal{A}$  accepts  $w$  if and only if there are  $a, b, c \in S$  such that  $a + b = c$ . Consider any successful run  $\rho$  of  $\mathcal{A}$  on  $w$ . Observe that the moment of reading the first symbol  $\spadesuit$  in  $w$  must coincide with firing the transition from  $p_2$  to  $q_1$ . At this moment, the automaton has consumed the first factor  $u$  of  $w$ , and there was a moment where it moved from state  $p_1$  to state  $p_2$  upon reading one of the  $\diamond$  symbols from  $u$ . Supposing that the transition in  $\rho$  from  $p_1$  to  $p_2$  happens at the  $i$ -th symbol  $\diamond$  of  $u$ , the clock valuation at the moment of reaching  $q_1$  for the first time must satisfy  $x = 2(s_i - s_{i-1}) + \dots + 2(s_2 - s_1) + 2s_1 (= 2s_i)$  and  $y = 0$ . We conclude the following.

► **Claim 9.** The set of possible clock valuations at the moment of reaching the state  $q_1$  for the first time is  $\{(x = 2a, y = 0) : a \in S\}$ .

Next, observe that the moment of reading the second occurrence of  $\spadesuit$  in  $w$  must coincide with firing the transition from  $q_2$  to  $r_1$ . Between the first and the second symbol  $\spadesuit$  the automaton consumes the second factor  $u$ , and during this the clock  $\mathbf{x}$  increases exactly by the sum of the time spans within  $u$ , i.e. by  $2M$ . On consuming the second factor  $u$ , the clock  $\mathbf{y}$  is reset once, and precisely when firing the transition from  $q_1$  to  $q_2$ , which happens on reading one of the occurrences of  $\diamond$  in  $u$ . Again, if this happens when reading the  $j$ -th occurrence of  $\diamond$ , then, after the reset,  $\mathbf{y}$  is incremented by exactly  $2s_j$  units. We conclude the following.

▷ **Claim 10.** The set of possible clock valuations at the moment of reaching the state  $r_1$  for the first time is  $\{(\mathbf{x} = 2a + 2M, \mathbf{y} = 2b) : a, b \in S\}$ .

Finally, after consuming the last factor  $v$ , the automaton can move to the accepting state  $r_2$  if and only if at some point, upon reading an occurrence of  $\diamond$ , the condition  $x + y = 4M$  holds. Observe that the sum of the first  $k$  numbers encoded in  $v$  is equal to  $M - s_{n-k+1}$ . Hence, after parsing those numbers, the set of possible clock valuations is  $\{(\mathbf{x} = 2a + 2M + M - c, \mathbf{y} = 2b + M - c) : a, b \in S\}$ , for some choice of  $c \in S$ . Moreover, the latter valuations satisfy the condition  $\mathbf{x} + \mathbf{y} = 4M$  if and only if  $a + b = c$ .

Based on the above arguments, we infer that a successful run like  $\rho$  exists on input  $w$  if and only if there are  $a, b, c \in S$  such that  $a + b = c$ . To conclude the proof, we observe that if an algorithm could decide whether  $\mathcal{A}$  accepts  $w$  in time  $\mathcal{O}(n^{2-\delta})$  for any  $\delta > 0$ , then by combining this algorithm with the presented construction, one could solve 3SUM in time  $\mathcal{O}(n^{2-\delta})$ . This would contradict the 3SUM Conjecture. ◀

Theorem 3 now follows almost directly from the previous lemma. Consider the timed automaton  $\mathcal{A}$  provided by Lemma 8. If a data structure as in the statement of the theorem existed, then using this data structure one could decide in strongly sub-quadratic time whether any input timed word  $w$  is accepted by  $\mathcal{A}$ , by simply applying the sequence of `read( $\cdot$ )` operations corresponding to  $w$ , followed by the query `accepted()`.