Dynamic data structures for timed automata acceptance

Alejandro Grez
Pontificia Universidad Católica de Chile, Chile

Filip Mazowiecki
Max Planck Institute for Software Systems, Germany

Michał Pilipczuk
University of Warsaw, Poland

Gabriele Puppis
University of Udine, Italy

Cristian Riveros
Pontificia Universidad Católica de Chile, Chile

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 through the lenses of parameterized dynamic data structures: we assume that the automaton is fixed and its size is the parameter, while the input word is revealed as in a stream, one symbol at a time following the natural order on positions. The goal is to design a dynamic data structure that can be efficiently updated upon revealing the next symbol, while maintaining the answer to the query on whether the word consisting of symbols revealed so far is accepted by the automaton. We provide complexity bounds for this dynamic acceptance problem for timed automata that process symbols interleaved with time spans. The main contribution is a dynamic data structure that maintains acceptance of a fixed one-clock timed automaton $\mathcal{A}$ with amortized update time $2^{\mathcal{O}(|\mathcal{A}|)}$ per input symbol.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases timed automata, data stream, dynamic data structure

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

1 Introduction

Imagine we would like to monitor whether the behavior of a server is correct. The run of the server can be abstracted by an infinite stream $w = a_1 a_2 a_3 \ldots \in \Sigma^\omega$, where Σ is a finite alphabet of possible events. The events are disclosed one at a time on the input, and at every moment we should tell whether the prefix consisting of the events observed so far is correct.

A simple yet expressive formalism for describing properties of such *data streams* is provided by classical finite automata. For example, suppose we would like to verify the property that a certain resource is being used by at most one process. Assume that the alphabet is $\Sigma = \{o, r\} \cup \Gamma$, where o denotes a request of the resource, r denotes a release of the resource, and Γ contains other immaterial events. The streams satisfying the discussed property can be then characterized as those where every prefix is accepted by the two-state automaton $\mathcal{A}$ of Figure 1. Here, a state indicates whether the resource is currently available or not.

Verifying the correctness of a stream over time can be formalized through the following *dynamic acceptance problem*: for a fixed automaton $\mathcal{A}$, design a *data structure* that upon receiving subsequent events from the stream, monitors whether the prefix read so far is accepted by $\mathcal{A}$. An obvious, though usually suboptimal solution would be to store in the data structure the prefix read so far, and, upon receiving a new symbol, run the automaton on the whole prefix. This would require time linear in the total length of the prefix, which after a while can become very large compared to $|\mathcal{A}|$, the size of the automaton $\mathcal{A}$. So we would like to minimize the update time by smartly organizing and reusing information computed before.

Cast in this way, the dynamic acceptance problem naturally lends itself to a treatment using the notions of parameterized complexity. Namely, we consider the automaton $\mathcal{A}$ fixed and use the parameter $|\mathcal{A}|$ as an auxiliary measure for expressing guarantees on the update time. Ideally, we would like to obtain update time bounded by a computable function of $|\mathcal{A}|$ only. This way, our work inscribes into the area of *parameterized dynamic data structures*, which is a direction that is still relatively unexplored, but starts to attract considerable attention; see e.g. [3, 7, 11] and references therein for an overview of recent advances.

For finite automata, the dynamic acceptance problem can be solved easily with update time $\mathcal{O}(|\mathcal{A}|)$, as follows. After reading a prefix u, the data structure stores the subset of states $S \subseteq Q$ in which the automaton may be after reading u (in general, we allow the automaton to be non-deterministic). Upon receiving the next input symbol, the set S is updated by applying the possible transitions on every state in S. Moreover, telling whether $\mathcal{A}$ accepts the current input prefix boils down to checking whether S contains an accepting state. Both the update and the query described above can be implemented in time linear in $|\mathcal{A}|$.

Unfortunately, real-life scenarios involve many aspects that cannot be captured by a simple formalism such as finite automata. One of these aspects is *time*. Consider the following example of property that needs to be verified: at every moment in time when an event occurs, a backup operation has been performed within the last 24 hours. A natural choice to model this and similar properties is to enhance finite automata with the ability of measuring time, by adding one or more *clocks*. A definition of the resulting automaton model, called *timed automaton*, is presented in Section 2. Intuitively, a possible timed automaton for the

Figure 1 Left: a finite automaton $\mathcal{A}$ recognising $\Gamma^*(o\Gamma^* r\Gamma^*)^*$. Both states of $\mathcal{A}$ are accepting, but invalid streams do not admit any run. Right: a timed automaton $\mathcal{B}$ with single clock $\mathtt{x}$.

considered property would have one clock $\mathtt{x}$ and two states, "before backup" and "after backup", and would behave as follows (see the right hand-side of Figure 1). The idea is that while processing an input prefix u, the automaton non-deterministically guesses a single backup event b and verifies that this event occurred within the last 24 hours. Thus, upon reading an occurrence of event b, the automaton may either ignore this event and carry on, or move from state "before backup" to state "after backup" and reset the clock. The input prefix u is accepted if the automaton reached state "after backup" and, during events since the last reset, the value of the clock has never exceeded 24 hours.

Timed automata are a central topic in the area of verification, and they have a rich and diverse literature, see e.g. [4, 8, 12]. In this work we are interested in the dynamic acceptance problem for timed automata, defined analogously to that for finite automata.

Note that in the setting of timed automata, the same technique that worked for finite automata will not work so easily. The reason is that for a finite automaton $\mathcal{A}$, the set of configurations in which $\mathcal{A}$ may be is a subset of the set of control states, whose size is bounded by the size of $\mathcal{A}$. On the other hand, a configuration of a timed automaton consists of a control state and a tuple of clock values, so the number of possible configurations is a priori unbounded. Concretely, after reading a prefix of length n, there may be as many as $\mathcal{O}(n^k)$ different configurations which the given k-clock timed automaton may possibly reach, due to non-determinism and clock resets. Efficient maintenance of this configuration set in a data structure poses the main conceptual challenge in this paper.

Our contribution. We design a dynamic data structure that, for a fixed timed automaton $\mathcal{A}$ with *one* clock, monitors whether $\mathcal{A}$ accepts the prefix read so far with amortized update time $2^{\mathcal{O}(|\mathcal{A}|)}$. This can be improved to worst-case (i.e. non-amortized) update time when the input stream is *discrete*, that is, when all time spans between consecutive events are equal. Our data structure actually works in a slightly more general setting, where the automaton $\mathcal{A}$ is not entirely fixed, but rather is provided on input upon initialization of the data structure.

We also give a somewhat complementary lower bound: under the 3SUM Conjecture, we prove that there exists a fixed timed automaton $\mathcal{A}$ with two clocks and additive constraints on them such that no data structure for the dynamic acceptance problem for $\mathcal{A}$ may achieve strongly sublinear amortized update time (i.e. time $\mathcal{O}(n^{1-\delta})$ for $\delta > 0$). Here, by additive constraints we mean that in the transition relation of $\mathcal{A}$ we may use affine clock conditions that involve more than one clock, e.g. $\mathtt{x} + \mathtt{y} = c$ where $\mathtt{x}, \mathtt{y}$ are clocks and c is a constant.

If the given timed automaton $\mathcal{A}$ has more than one clock, but only constraints involving a single clock are allowed, it remains open whether there is an efficient data structure for the dynamic acceptance problem or a lower bound similar to the above one.

Related work. The setting in this work is close to *runtime verification* [24], an area that focuses on verification techniques that could be performed at runtime, e.g. using timed automata [30, 10]. However, while we study monitoring a data stream through a suitable data structure in the *dynamic* setting, studies on runtime verification typically focus on *static* problems. An example of such a problem is: given an input prefix u, verify whether there is a sequence of events that extends u to a word accepted by the device (e.g. a finite automaton). The problem studied in [29] is similar to the setting presented here; however, this line of work considers constants (e.g. 24 in Figure 1) as part of the input contributing to the considered parameter, and this considerably simplifies the problem (see Section 2 and 3).

The dynamic acceptance problem that we consider here resembles the setting of *streaming algorithms*; see e.g. [5, 13, 21] for works with a similar motivation. In this context, a typical problem is to compute (possibly approximately) some statistics or an aggregate function over the sequence of data, where the main point is to assume severe restrictions on the space usage. Note that in our setting, we focus on obtaining low time complexity per update and query, rather than optimizing the space complexity. In this respect, our work leans more towards the area of dynamic data structures, in particular dynamic query evaluation [9, 22]. For Boolean properties several papers [25, 26, 6] have considered streaming algorithms for testing membership in regular and context-free languages. Another variant of the problem was considered in [18, 17, 16], 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 [28], which studies the dynamic evaluation problem for monoids over a sliding window, and describes a data structure that can be updated in constant time for a fixed finite monoid. When the monoid is finite, the considered problem is basically the same as monitoring whether the input stream restricted to the sliding window is accepted by a finite automaton. We explain in Appendix A that, in this case, the problem can be reduced to the dynamic acceptance problem for a special form of timed automaton.

2 Preliminaries

Finite automata. A *finite automaton* is a tuple $\mathcal{A} = (\Sigma, Q, I, E, F)$, where Σ 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, ρ is a *successful* run if $q_0 \in I$ and $q_n \in F$. A word w is *accepted* by $\mathcal{A}$ if there is a successful run of $\mathcal{A}$ on w.

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

A *timed automaton* is a tuple $\mathcal{A} = (\Sigma, Q, X, I, E, F)$, where Q, Σ, 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 finite 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, ν), where $q \in Q$ and ν is a clock valuation. Recall that finite automata process words over a finite alphabet Σ; likewise, timed automata process timed words over an alphabet of the form $\Sigma \uplus \mathbb{R}_{>0}$, with Σ finite.

A *run* of a timed automaton $\mathcal{A}$ on a timed word $w = e_1 \dots e_n \in (\Sigma \cup \mathbb{R}_{>0})^*$ is a 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, ν_i) is a configuration and

- if $e_i \in \mathbb{R}_{>0}$, then $q_{i+1} = q_i$ and $\nu_{i+1}(\mathtt{x}) = \nu_i(\mathtt{x}) + e_i$ for all $\mathtt{x} \in X$;
- 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 $\nu_{i+1}(\mathtt{x}) = 0$ or $\nu_{i+1}(\mathtt{x}) = \nu_i(\mathtt{x})$ depending on whether $\mathtt{x} \in Z$ or $\mathtt{x} \in X \setminus Z$.

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. An example of a one clock timed automaton was given in the introduction (see Figure 1).

A run ρ as above is *successful* if $q_0 \in I$, $\nu_0(\mathtt{x}) = 0$ for all $\mathtt{x} \in X$, and $q_n \in F$. A word $w \in (\Sigma \cup \mathbb{R}_{>0})^*$ is *accepted* by $\mathcal{A}$ if there is a successful run of $\mathcal{A}$ 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 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 $\mathtt{x} < c$, $\mathtt{x} > c$, $\mathtt{x} = c$) appearing in the clock condition γ. *Note that the size of a timed automaton does not take into account the magnitude of the clock constants.* These constants are specified with the automaton and stored in suitable floating-point memory cells (see the computation model below).

Computation model. As clock constants and time spans in the input stream are arbitrary real numbers, it is convenient to use the *real RAM model* of computation. This is a standard model with 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. Basic arithmetic operations — addition, subtraction, multiplication, and division — can be performed in unit time, but modulo arithmetics and rounding are not included in the model. In fact, we do not use multiplication or division on real numbers either.

3 The dynamic acceptance problem and main results

The *dynamic acceptance problem* amounts to designing a data structure that can be initialized for a given timed automaton $\mathcal{A}$ with one clock, and afterwards, upon consuming consecutive elements of the data stream, efficiently maintains the information on whether the word read so far is accepted by $\mathcal{A}$. Formally, the data structure should support the following operations:

- `init`$(\mathcal{A})$: Initialize the data structure for a given automaton $\mathcal{A}$. This automaton is fixed for the entire lifespan of the data structure.
- `accepted()`: Query whether the prefix of the stream consumed up to the current moment is accepted by $\mathcal{A}$.
- `read`(e): Consume the next element e from the input stream, be it a letter from Σ or a time span from $\mathbb{R}_{>0}$, and update the data structure accordingly.

The running time of each of these operations needs to be as low as possible. More precisely, we shall say that a data structure *supports dynamic acceptance in time* $f(s,n)$ if the first operation `init`$(\mathcal{A})$ takes at most $f(s,0)$ time, and every subsequent execution of `accepted()` or `read`(e) takes at most $f(s,n)$ time, where $s = |\mathcal{A}|$ and n is the number of stream elements consumed so far. Similarly, a data structure *supports dynamic acceptance in amortized time* $f(s,n)$ if the first operation `init`$(\mathcal{A})$ takes at most $f(s,0)$ time, while every n subsequent operations `accepted()` and `read`(e) take at most $n \cdot f(s,n)$ time in total. Ultimately, we are mostly interested in designing data structures where the complexity guarantee $f(s,n)$ is independent of n, that is, the (amortized) update time is a function of $|\mathcal{A}|$ only.

In Appendix A we provide two examples of applications of the dynamic acceptance problem in the literature on verification. The first one concerns the *sliding window model*, while the second is about *complex event processing*.

Results. We say that a stream w is *discrete* if its elements range over $\Sigma \uplus \{1\}$, that is, if all time spans in the stream coincide with the time unit 1. Our main result is the following:

▶ **Theorem 1.** *Consider the dynamic acceptance problem for timed automata with one clock. There is a data structure that*

- *supports dynamic acceptance in time $2^{\mathcal{O}(|\mathcal{A}|)}$ on discrete streams, and*
- *supports dynamic acceptance in amortized time $2^{\mathcal{O}(|\mathcal{A}|)}$ on arbitrary streams,*

where $\mathcal{A}$ is the automaton provided upon initialization.

We stress that the complexity in Theorem 1 depends only on the size of $\mathcal{A}$. In particular, it does not depend on the bitlength of clock constants (e.g. 24 in Figure 1). Note that thanks to the assumption of the real RAM model, the question of the complexity of arithmetic operations on reals is separated from the running time analysis in the proof of Theorem 1. This feature reflects the real-life scenarios, where the automaton is small, while real numbers involved can be efficiently manipulated by the processor despite having large bitlength.

The proof of Theorem 1 is presented in Section 4. We do not know whether this theorem can be generalized to timed automata with more than one clock while preserving independence of the time complexity of updates from the length of the consumed stream prefix.

However, we establish a negative result for a slightly more powerful model of timed automata, called timed automata with additive constraints (see e.g. [8]). 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 $\left(\sum_{\mathtt{x}\in Z} \mathtt{x}\right) \sim c$, where $Z \subseteq X$ and $\sim \in \{<, >, =\}$. For instance, one can write $\mathtt{x} + \mathtt{y} \leq c$, where c is a clock constant.

Our lower bound relies on the 3SUM conjecture, stated below. 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 $n = |S|$; the 3SUM Conjecture asserts that this 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 values forming the input.*

The 3SUM Conjecture is widely used in computational geometry and fine-grained complexity theory (see an overview in [2, Appendix A]), and it was applied to establish lower bounds for dynamic problems in [1, 3, 23, 27]. Our lower bound, stated below, is similar in nature.

▶ **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 that, when initialized on $\mathcal{A}$, supports dynamic acceptance in time $\mathcal{O}(n^{1-\delta})$ for any $\delta > 0$, where n is the length of the consumed stream prefix.*

The proof of the above theorem is in Appendix B, together with a broader discussion of the 3SUM Conjecture and of the extension of timed automata by additive constraints. Again, we do not know whether a negative result similar to the above one also holds for plain timed automata (without additive constraints).

4 Data structure: proof of Theorem 1

Notation. Let us fix, once and for all, the timed automaton $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ with a single clock $\mathtt{x}$ that is provided upon initialization. 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 at any time. As $\mathcal{A}$ 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 $\mathtt{x}$.

Let $0 = C_0 < C_1 < \ldots < C_k$ be the clock constants used in $\mathcal{A}$, where we assume without loss of generality that $C_0 = 0$. For simplicity we also let $C_{k+1} = \infty$. Note that $k \leq |\mathcal{A}|$.

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 $\texttt{read}(e)$. 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 dynamic acceptance problem essentially boils down to designing an efficient data structure that maintains K_n upon reading subsequent 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 that are in the same order 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 J_0, J_1, $\dots J_{2k+1}$, where $J_{2i} = [C_i, C_i]$, $J_{2i+1} = (C_i, C_{i+1})$, for all $p \in \{0, \dots, k\}$. The following assertion holds: 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*.

Inner data structure: an overview. Every inner data structure is constructed for 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 $\texttt{init}(J)$ stores the interval J and initializes $\mathbb{D}[J]$ by setting $L = \emptyset$.
- Method $\texttt{accepted}()$ returns true or false, depending on whether or not L contains an accepting configuration, that is, a configuration (q,t) such that $q \in F$.
- Method $\texttt{insert}(q,t)$ 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 $\texttt{updateTime}(r)$, 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 $\texttt{updateLetter}(a)$ updates L by applying to all configurations in L all possible transitions over the given letter $a \in \Sigma$. Precisely, the updated set 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. The 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, 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 each $J \in \{J_0, J_1, \dots, J_{2k+1}\}$, the inner data structure $\mathbb{D}[J]$ can be implemented so that methods* $\texttt{init}()$, $\texttt{accepted}()$, $\texttt{insert}(\cdot,\cdot)$, *and* $\texttt{updateLetter}(\cdot)$ *run in time $2^{\mathcal{O}(|\mathcal{A}|)}$, while method* $\texttt{updateTime}(\cdot)$ *runs in time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is 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 monitors the acceptance of $\mathcal{A}$.

4.1 Outer data structure

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

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

We first explain how the outer data structure implements the promised operations: initialization, queries about the acceptance, and updates upon reading the next element of the stream w. Then we discuss the amortized complexity of the updates.

Initialization. Given $\mathcal{A}$, we store $\mathcal{A}$ in the data structure and we read the clock constants $0 = C_0 < C_1 < \ldots < C_k$ from $\mathcal{A}$. Then we initialize $2k+1$ copies $\mathbb{D}[0], \ldots, \mathbb{D}[2k+1]$ of the inner data structure by calling method `init`(J) for each interval J among $J_0, J_1, \ldots, J_{2k+1}$. Finally, 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 $\mathbb{D}[0], \ldots, \mathbb{D}[2k+1]$ 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 element 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 $S_{2k+1}, S_{2k}, \ldots, S_0$ 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 $t \le t'$ 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. All these configurations have clock value 0, hence the length of R_i is at most $|Q|$. It now suffices to insert all the configurations $(q, 0)$ appearing on all the lists R_i to $\mathbb{D}[0]$ 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.

From Lemma 4 it readily follows that each of the following operations takes time $2^{\mathcal{O}(|\mathcal{A}|)}$: initialization, 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 $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \sum_{i=0}^{2k+1} |S_i|$, where $S_0, \ldots, S_{2k+1}$ 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 amortized time complexity of all these updates is bounded by $2^{\mathcal{O}(|\mathcal{A}|)}$.

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 clock value 0 may also become active. 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. Note that there may be at most $|Q|$ different active configurations with the same active clock value, hence the complexity of each update by a time span is bounded by $2^{\mathcal{O}(|\mathcal{A}|)} \cdot |Q|$ times 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 $2k+1$ 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 bounded by $2^{\mathcal{O}(|\mathcal{A}|)} \cdot |Q| \cdot (2k+1) \cdot (n+1) = 2^{\mathcal{O}(|\mathcal{A}|)} \cdot n$. Hence, the amortized time complexity is $2^{\mathcal{O}(|\mathcal{A}|)}$.

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

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

4.2 Inner data structure

We now describe 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 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 $\lambda\colon \mathbb{R}_{\geq 0} \to 2^Q$ 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 $\lambda(t) \neq \varnothing$. Recall that we assume that $\widehat{L} \subseteq J$.

Before we dive into the details, let us discuss the intuition. The basic idea is to store all the configurations in L in a queue, implemented as a doubly-linked list ordered by non-decreasing clock values. To handle clock values efficiently, we do not store them directly. Instead, we maintain a global clock that measures the total time since the initialization of the data structure, and each configuration bears a timestamp that is the value of this global clock at the moment of the last reset. Thus, updating by a time span boils down to increasing the value of the global clock and popping any configurations at the back of the queue whose clock values ceased to fit into the interval J.

Updating by a letter is more problematic, as we need to apply the transition relation of the automaton $\mathcal{A}$ to all the configurations of L simultaneously. In the data structure we store a partition of the active clock values $\widehat{L}$ according to their images under $\lambda(\cdot)$, so that for each block of this partition (whose number is at most $2^{|Q|}$), we can simultaneously update all corresponding configurations in constant time. There is a caveat here: it is possible that for some $t, t' \in \widehat{L}$ we have $\lambda(t) \neq \lambda(t')$ before the update, but $\lambda(t) = \lambda(t')$ after the update. That is, the blocks of the partition may require merging upon updates. We resolve this issue by representing the partition in a *forest*, similarly as the union-find data structure would do. The key point is that the height of this forest can be kept bounded by $2^{|Q|}$.

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

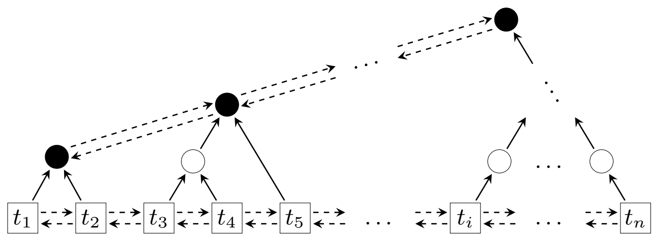

Figure 2 The inner data structure. List elements are depicted as squares while the forest nodes are depicted as circles. The black circles are the roots.

- The *clock*, denoted `y`, is a real that represents the total time elapsed since initialization.
- 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 λ.

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

<u>*The list.*</u> 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 α on `l` is a record consisting of:

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

As usual, the data structure stores `l` by maintaining pointers to the first and last nodes.

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

<u>*The forest.*</u> Forest `f` represents the mapping from elements $t \in \widehat{L}$, encoded in `l`, to respective sets of control states $\lambda(t)$. It is a rooted forest where nodes may have arbitrarily many children, and these children are unordered. Every node γ of `f` is a record containing:

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

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

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

- $\texttt{states}(\beta) \subseteq Q$: a non-empty subset of control states for which β is responsible; and
- $\texttt{rank}(\beta)$: an integer from the set $\{1, 2, 3, \ldots, 2^{|Q|}\}$.

We will maintain two invariants about these values. First, the sets $\texttt{states}(\beta)$ and the ranks $\texttt{rank}(\beta)$ should be pairwise different for distinct roots β of `f`. Note that this means that `f` always has at most $2^{|Q|} - 1$ roots. Second, for every root β, the *tree rooted at* β — which is the tree containing β and all its descendants in `f` — has depth at most $\texttt{rank}(\beta)$. 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 `f` is bounded by $2^{|Q|}$.

Function λ is then represented as follows. For every node α on `l`, let $\texttt{root}(\alpha)$ be the root of the tree of `f` that contains α. Then denoting $t = \texttt{clock}(\alpha)$, we have $\lambda(t) = \texttt{states}(\texttt{root}(\alpha))$. Note that the invariant stated above implies that from every leaf α of `f`, $\texttt{root}(\alpha)$ can be computed from α by following the $\texttt{parent}(\cdot)$ pointer at most $2^{|Q|}$ times. Hence, given $t \in \widehat{L}$ and a node α on `l` satisfying $t = \texttt{clock}(\alpha)$, we can compute $\lambda(t)$ in time $\mathcal{O}(2^{|Q|}) \le 2^{\mathcal{O}(|\mathcal{A}|)}$.

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

I2. For each node α on `l`, the value $\texttt{clock}(\alpha) = \texttt{y} - \texttt{timestamp}(\alpha)$ belongs to J.

I3. The nodes on `l` are sorted by increasing clock values, or equally by decreasing timestamps. That is, $\texttt{timestamp}(\alpha) > \texttt{timestamp}(\texttt{next}(\alpha))$ for every non-last node α on `l`.

I4. Every root of `f` has at least one child, and the leaves of `f` are exactly all the nodes on `l`.

I5. The roots of `f` carry pairwise different, non-empty sets of control states, and they have pairwise different ranks. Moreover, all the ranks belong to the set $\{1, 2, \dots, 2^{|Q|}\}$.

I6. For every root β of `f`, the depth of the tree rooted at β is at most $\texttt{rank}(\beta)$.

Implementation. Now we show how to implement the methods $\texttt{init}(J)$, $\texttt{accepted}()$, $\texttt{insert}(q,t)$, $\texttt{updateTime}(r)$, and $\texttt{updateLetter}(a)$ in the data structure. Recall that all these methods should work in time $2^{\mathcal{O}(|\mathcal{A}|)}$, with the exception of $\texttt{updateTime}(r)$ which is allowed to work in time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is the size of 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 `l` and from the forest `f`, as this procedure will be used several times. Suppose we are given a node α on the list `l` and we would like to remove it, which corresponds to removing from L all configurations (q,t) where $t = \texttt{clock}(\alpha)$ and $q \in \lambda(t)$. We can remove α from `l` in the usual way. Then we remove α from `f` as follows. First, we decrement the counter of children in the parent of α. If this counter stays positive then there is nothing more to do. Otherwise, we need to remove the parent of α as well, and accordingly decrement the counter of children in the grandparent of α. This can again trigger removal of the grandparent and so on. If eventually we need to remove a root of `f`, we also remove it from the list `r` in the usual way. Note that since by Invariants (I5) and (I6), the depth of `f` is bounded by $2^{|Q|}$, the whole procedure can be performed in time $\mathcal{O}(2^{|Q|}) \le 2^{\mathcal{O}(|\mathcal{A}|)}$. It is clear that all the invariants are maintained.

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

Acceptance query. The $\texttt{accepted}()$ method is implemented as follows. We iterate through the list `r` to check whether there exists a root β of `f` such that $\texttt{states}(\texttt{f})$ contains any accepting state, say q. If this is the case, then by Invariant (I4) there is a node α on `l` satisfying $\texttt{root}(\alpha) = \beta$, hence (q,t) is an accepting configuration that belongs to L, where $t = \texttt{clock}(\alpha)$. 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 `r` has length at most $2^{|Q|} - 1$, the above procedure works in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

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

Let α be the first node on the list `l`. Let $t' = \texttt{clock}(\alpha)$. By the promise, we have $t \le t'$. We consider cases: either $t < t'$ or $t = t'$. The former case also captures the situation when `l` is empty. When $t < t'$ or `l` is empty, the new configuration (q, t) gives rise to a new active clock value t. Therefore, we create a new list node α_0 and insert it at the front of the list `l`. We set the timestamp as $\texttt{timestamp}(\alpha_0) = \texttt{y} - t$, 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 α_0 to the forest `f`. We iterate through the list `r` in search for a root β that satisfies $\texttt{states}(\beta) = \{q\}$. In case there is one, we simply set $\texttt{parent}(\alpha_0) = \beta$ and increment $\#\texttt{children}(\beta)$. Otherwise, we construct a new root β_0 with $\texttt{states}(\beta_0) = \{q\}$ and $\#\texttt{children}(\beta_0) = 1$, insert it at the front of the list `r`, and set $\texttt{parent}(\alpha_0) = \beta_0$. To determine the rank of β_0, we find the smallest integer $k \in \{1, \ldots, 2^{|Q|}\}$ that is *not* used as the rank of any other root of `f`. Observe that, by Invariant (I5), the forest `f` has at most $2^{|Q|} - 1$ roots, so there is always such a number k, and it can be found in time $2^{\mathcal{O}(|\mathcal{A}|)}$ by inspecting the list `r`. We then set $\texttt{rank}(\beta_0) = k$. It is clear that this operation can be performed in time $2^{\mathcal{O}(|\mathcal{A}|)}$, and that Invariants (I4), (I5), and (I6) are maintained. For the last one, observe that the new leaf α_0 is attached directly under a root of `f`, so no tree in `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 $\texttt{root}(\alpha)$ in time $2^{\mathcal{O}(|\mathcal{A}|)}$ and read $X = \texttt{states}(\texttt{root}(\alpha))$. 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 α from `l` and from `f`, using the operation described earlier, and we insert a new node α' at the front of `l`, with the same timestamp equal to that of α. Thus, $\texttt{clock}(\alpha') = t$. We next insert the new node α' to the forest `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 time $2^{\mathcal{O}(|\mathcal{A}|)}$, and the same argumentation shows that all the invariants are maintained.

Update by a time span. Next, we implement the method $\texttt{updateTime}(r)$, for $r \in \mathbb{R}_{>0}$. First, we increment `y` by r. Thus, for every node α in the list `l`, the value $\texttt{clock}(\alpha)$ is incremented by r. However, the Invariant (I2) may have ceased to hold, as 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 the output of the method.

We extract these configurations as follows. Construct an initially empty list of configuration `lret`, on which we shall build the output. Iterate through the list `l`, starting from its back. For each consecutive node α, compute $t = \texttt{clock}(\alpha)$. If $t \in J$, then break the iteration and return `lret`, as there are no more configurations to remove. Otherwise, find $\texttt{root}(\alpha)$ in time $2^{\mathcal{O}(|\mathcal{A}|)}$, read $\lambda(t) = \texttt{states}(\texttt{root}(\alpha))$, and add at the front of `lret` all configurations (q, t) for $q \in \lambda(t)$, in any order. Then remove α from the list `l` and from the forest `f`, and proceed to the previous node in `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. Note that the output list `lret` is ordered by non-decreasing clock values, as required. As for the time complexity, the procedure presented above takes time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell'$, where ℓ' is the number of nodes that we remove from `l`. As for every node α the set $\texttt{states}(\texttt{root}(\alpha))$ is non-empty and of size at most $|Q|$, with every removed node we add to `lret` between 1 and $|Q|$ new configurations. Hence, we can also bound the complexity by $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is the number of configurations that appear in the output list `lret`.

Update by a letter. We proceed to the method $\texttt{updateLetter}(a)$, 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 $(p, a, q, \gamma, \varnothing)$ in E for some $p \in X$ and clock condition γ 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 is $\{(q, 0) : q \in \Psi(X)\}$ where X is the set of all states appearing in the configurations of L. Note that, by Invariant (I4), X can be computed in time $2^{\mathcal{O}(|\mathcal{A}|)}$ by iterating through the list $\texttt{r}$ and computing the union of sets $\texttt{states}(\beta)$ for roots β appearing on it. Thus, the output can be computed in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

Second, we need to update the values of function λ by applying all possible non-resetting transitions over a. This can be done by iterating through the list $\texttt{r}$ and, for each root β appearing on it, substituting $\texttt{states}(\beta)$ with $\Phi(\texttt{states}(\beta))$. Note that since we assumed that for every state q, some transition over a is always available at q, it follows that Φ maps non-empty sets of states to non-empty sets of states. Hence, after this substitution the roots of $\texttt{f}$ 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 β of $\texttt{f}$, inspect the list $\texttt{r}$ and find the root β' that has the largest rank among those satisfying $\texttt{states}(\beta) = \texttt{states}(\beta')$. If $\beta = \beta'$, then do nothing. Otherwise, turn β into a non-root node of $\texttt{f}$, remove it from the list $\texttt{r}$, set $\texttt{parent}(\beta) = \beta'$, and increment $\#\texttt{children}(\beta')$ by one. Note that after applying this modification, the function λ 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 $\texttt{f}$. Observe that each modification of $\texttt{f}$ 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 $\texttt{f}$ is always bounded by $2^{|Q|} - 1$, all the operations described above can be performed in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

5 Concluding remarks and future work

In this work we studied the dynamic acceptance problem for timed automata processing data streams. We designed a suitable data structure for one-clock timed automata, where the amortized update time depends only on the size of the automaton. We leave as an open question whether this result can be generalised to the case of multiple clocks.

More generally speaking, it seems that our work identifies dynamic variants of classic automata problems as a potential area of interest for the paradigm of parameterized dynamic data structures. More precisely, if the automaton model in question allows for the device to potentially be in an unbounded number of configurations, then the dynamic maintenance of this set of configurations is a computationally challenging problem, as show-cased in this paper. There are multiple models of devices where similar questions can be asked. Examples include counter automata, register automata, weighted automata, or pushdown automata.

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. *ACM Trans. Algorithms*, 16(4):45:1–45:46, 2020. URL: https://doi.org/10.1145/3395037, doi:10.1145/3395037.
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, 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 *22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2002*, 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. Max Bannach, Zacharias Heinrich, Rüdiger Reischuk, and Till Tantau. Dynamic kernels for hitting sets and set packing. *Electron. Colloquium Comput. Complex.*, 26:146, 2019. URL: https://eccc.weizmann.ac.il/report/2019/146.
8. 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, doi:10.1016/S0020-0190(00)00075-2.
9. Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *36th ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 303–318, 2017.
10. Patricia Bouyer, Samy Jaziri, and Nicolas Markey. Efficient timed diagnosis using automata with timed domains. In *18th International Conference on Runtime Verification, RV 2018*, pages 205–221, 2018. URL: https://doi.org/10.1007/978-3-030-03769-7_12, doi:10.1007/978-3-030-03769-7_12.
11. Jiehua Chen, Wojciech Czerwiński, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michał Pilipczuk, Manuel Sorge, Bartłomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021. URL: https://doi.org/10.1137/1.9781611976465.50, doi:10.1137/1.9781611976465.50.
12. Lorenzo Clemente and Sławomir Lasota. Timed pushdown automata revisited. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015*, pages 738–749, 2015. URL: https://doi.org/10.1109/LICS.2015.73, doi:10.1109/LICS.2015.73.
13. Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.
14. 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, doi:10.1016/0925-7721(95)00022-2.
15. 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.
16. 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.

17 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*, pages 31:1–31:14, 2018.

18 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*, pages 18:1–18:14, 2016.

19 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event processing. In *22nd International Conference on Database Theory, ICDT 2019*, volume 127 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL: `https://doi.org/10.4230/LIPIcs.ICDT.2019.5`, `doi:10.4230/LIPIcs.ICDT.2019.5`.

20 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`.

21 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.

22 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.

23 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In *27th 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`.

24 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009. URL: `https://doi.org/10.1016/j.jlap.2008.08.004`, `doi:10.1016/j.jlap.2008.08.004`.

25 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.

26 Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014.

27 Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *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`.

28 Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *11th ACM International Conference on Distributed and Event-based Systems*, pages 66–77, 2017.

29 Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electron. Notes Theor. Comput. Sci.*, 113:145–162, 2005. URL: `https://doi.org/10.1016/j.entcs.2004.01.029`, `doi:10.1016/j.entcs.2004.01.029`.

30 Stavros Tripakis. Fault diagnosis for timed automata. In *7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2002*, pages 205–224, 2002. URL: `https://doi.org/10.1007/3-540-45739-9_14`, `doi:10.1007/3-540-45739-9\_14`.

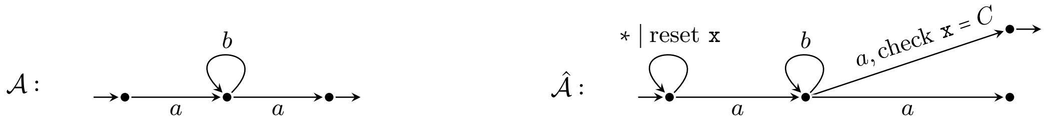

Figure 3 Reducing the sliding window membership problem to the dynamic acceptance problem.

A Examples

▶ **Example 5.** We discuss the relationship between our monitoring problem for timed automata and the monitoring problem for finite monoids over a sliding window, considered in [28]. The common point of these two problems is when the monoid to be monitored is finite. In this case, the monoid is basically equivalent to a finite automaton, in the sense that every monoid element represents a regular language, which can then be described by a finite automaton. Therefore, monitoring a finite monoid over a sliding window is reducible to the automaton membership problem in the *sliding window model* (see, for instance [16]). We describe this problem below.

Let $\mathcal{A} = (\Sigma, Q, I, E, F)$ be a finite automaton and C a positive integer defining the width of the sliding window. The membership problem of $\mathcal{A}$ with a sliding window of width C consists of processing an arbitrary input $w = a_1 a_2 a_3 \dots$ over Σ from left to right, while maintaining the answer to the following query: *is the sequence of the last C consumed letters accepted by $\mathcal{A}$?* The goal is design a data structure whose update time depends only on the automaton $\mathcal{A}$, and is independent of the size of the window C.

We now explain how the above problem can be reduced to our dynamic acceptance problem for timed automata. In this setting, we consider only streams that are discrete. In fact, we will enforce a slightly more restricted form of streams: we assume that every input stream belongs to the language $(\{1\} \cdot \Sigma)^\omega$, namely, that the letters from Σ are interleaved by the time unit 1. We map the input word $w = a_1 a_2 a_3 \dots$ to a corresponding discrete stream $\widehat{w} = 1 a_1 1 a_2 1 a_3 \dots$, and modify the finite automaton $\mathcal{A}$ 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 of $\widehat{\mathcal{A}}$, and a clock $\mathtt{x}$. We then replace every transition (q, a, q') of $\mathcal{A}$ with the transition $(q, a, \mathtt{true}, q', \varnothing)$. 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 $(q, a, \mathtt{x} = C, \widehat{q}, \varnothing)$ in $\widehat{\mathcal{A}}$. Finally, we add looping transitions on the initial states that reset the clock, that is, transitions of the form $(q, a, \mathtt{true}, q, \{\mathtt{x}\})$, with $q \in I$ and $a \in \Sigma$. Figure 3 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 membership problem for $\mathcal{A}$ in the C-width sliding window model is reduced to the dynamic acceptance problem for $\widehat{\mathcal{A}}$ over the stream $\widehat{w}$. By Theorem 1, we know that there is a data structure that supports dynamic acceptance for $\widehat{\mathcal{A}}$ with update time $2^{\mathcal{O}(|\widehat{\mathcal{A}}|)} = 2^{\mathcal{O}(|\mathcal{A}|)}$. This means that we can process one letter at a time from a word w, while answering in time $2^{\mathcal{O}(|\mathcal{A}|)}$ whether $\mathcal{A}$ accepts the sequence of the last C consumed letters. Note that the complexity here is independent of C.

▶ **Example 6.** Here we consider a scenario from *complex event processing* (CEP), with a specification language called CEL and defined by the following grammar [19]:

$$\varphi \;\coloneqq\; a \;\mid\; \varphi ; \varphi \;\mid\; \varphi \text{ WITHIN } t$$

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

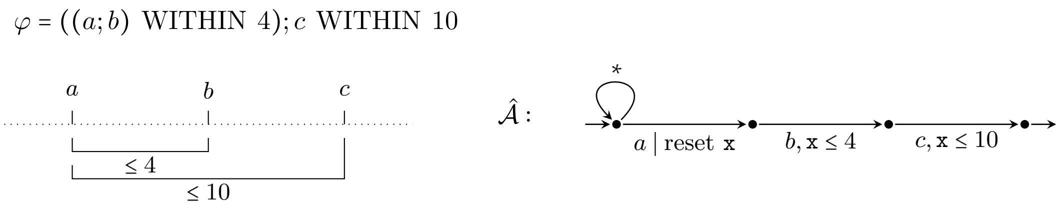

Figure 4 Translation of a CEL expression into an equivalent single-clock timed automaton.

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

Given a word $w = a_1 a_2 \dots$ and an expression φ, 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 φ. One can reduce this latter problem to our monitoring problem for timed automata, by using a discrete timed word $\widehat{w} = 1a_1 1a_2 1 \dots$ as before and by translating the expression φ 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 characterize 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 $\varphi = ((a; b)$ WITHIN $4); c$ 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 4 shows a single-clock timed automaton that is equivalent to φ, in the sense that it accepts a timed word of the form $1a_1 1a_2 1 \dots 1a_n$ if and only if $a_1 a_2 \dots a_n \vDash \varphi$. In this case one can validate any input stream against a fixed expression φ in time that is constant per input letter, by simply reducing to our dynamic acceptance problem for single-clock timed automata and discrete timed words.

B Lower bound for two-clock timed automata with additive constraints

In this section, we prove a complexity lower bound for a variant of the dynamic acceptance problem. Ideally, we would like to prove that there is a timed automaton $\mathcal{A}$ with two clocks such that no data structure can support dynamic acceptance for $\mathcal{A}$ in time depending only on $|\mathcal{A}|$. This would imply that our result (Theorem 1) for the dynamic acceptance problem for single-clock timed automata 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 $\left(\sum_{\mathtt{x} \in Z} \mathtt{x}\right) \sim c$. To give some background, let us discuss in more detail the power of this extension. Allowing additive constraints is a nontrivial extension of timed automata and in particular it makes the emptiness problem undecidable [8, Theorem 2]. However, undecidability holds when at least four clocks are available. Moreover, it is shown that for timed automata with additive constraints with two clocks the emptiness problem is decidable; and the proof is a straightforward modification of the standard region construction [8, Proposition 1].

Our lower bound is based on the 3SUM Conjecture, which we restate below for convenience.

▶ **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 values forming the input.*

The 3SUM Conjecture was introduced by Gajentaan and Overmars [14, 15] in a stronger form, which postulated the non-existence of *sub-quadratic* algorithms, that is, achieving running time $o(n^2)$. This formulation was refuted by Grønlund and Pettie [20], 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 randomization is allowed. However, the existence of a strongly sub-quadratic algorithm is conjectured to be hard.

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, 23, 27]. 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. That is, we provide a lower bound for the dynamic acceptance problem for 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 that, when initialized on $\mathcal{A}$, supports dynamic acceptance in time $\mathcal{O}(n^{1-\delta})$ for any $\delta > 0$, where n is the length of the consumed stream prefix.*

Our approach is similar in spirit to the other lower bounds on dynamic problems, which we mentioned above [1, 3, 23, 27]. 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 amortized 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.

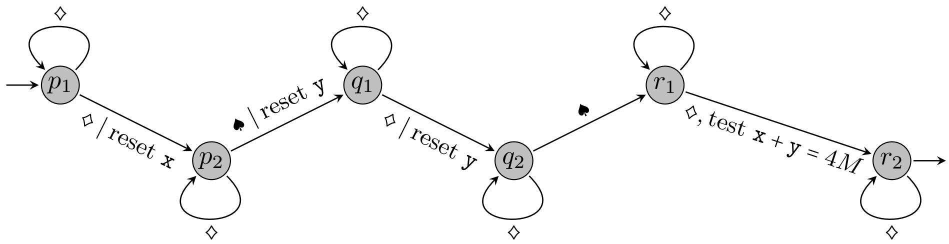

Figure 5 Timed automaton for reducing 3SUM.

▶ **Lemma 7.** *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 $w \in \left(\Sigma \uplus \mathbb{R}_{>0}\right)^*$ as input, where Σ is a two-letter alphabet, decides whether $\mathcal{A}$ accepts w in time $\mathcal{O}(n^{2-\delta})$ for any $\delta > 0$ and for $n = |w|$.*

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 $w \in \left(\Sigma \uplus \mathbb{R}_{>0}\right)^*$ 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 analyze 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 has length $\mathcal{O}(n)$ and 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 5. It uses two clocks, named $\mathtt{x}$ and $\mathtt{y}$. 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 analyze 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 ρ 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 ρ 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 $\mathtt{x} = 2(s_i - s_{i-1}) + \dots + 2(s_2 - s_1) + 2s_1 \ (= 2s_i)$ and $\mathtt{y} = 0$. We conclude the following.

▷ Claim 8. The set of possible clock valuations at the moment of reaching the state q_1 for the first time is $\{(\mathtt{x} = 2a,\ \mathtt{y} = 0) : a \in S\}$.

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 $\mathtt{x}$ increases exactly by the sum of the time spans within u, i.e. by $2M$. On consuming the second factor u, the clock $\mathtt{y}$ 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, $\mathtt{y}$ is incremented by exactly $2s_j$ units. We conclude the following.

▷ Claim 9. The set of possible clock valuations at the moment of reaching the state r_1 for the first time is $\{(\mathtt{x} = 2a + 2M,\ \mathtt{y} = 2b) : a, b \in S\}$.

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 $\{(\mathtt{x} = 2a + 2M + M - c,\ \mathtt{y} = 2b + M - c) : a, b \in S\}$, for some choice of $c \in S$. Moreover, the latter valuations satisfy the condition $\mathtt{x} + \mathtt{y} = 4M$ if and only if $a + b = c$.

Based on the above arguments, we infer that a successful run like ρ 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 7. 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(·)` operations corresponding to w, followed by the query `accepted()`.