



UNIVERSITÀ
DEGLI STUDI
DI UDINE

Università degli studi di Udine

Dynamic data structures for timed automata acceptance

Original

Availability:

This version is available <http://hdl.handle.net/11390/1217472> since 2024-07-26T10:38:15Z

Publisher:

Schloss Dagstuhl- Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing

Published

DOI:10.4230/LIPIcs.IPEC.2021.20

Terms of use:

The institutional repository of the University of Udine (<http://air.uniud.it>) is provided by ARIC services. The aim is to enable open access to all the world.

Publisher copyright

(Article begins on next page)

1 Dynamic data structures for timed automata 2 acceptance

3 **Alejandro Grez** ✉

4 Pontificia Universidad Católica de Chile, Chile

5 **Filip Mazowiecki** ✉

6 Max Planck Institute for Software Systems, Germany

7 **Michał Pilipczuk** ✉

8 University of Warsaw, Poland

9 **Gabriele Puppis** ✉

10 University of Udine, Italy

11 **Cristian Riveros** ✉

12 Pontificia Universidad Católica de Chile, Chile

13 — Abstract —

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

24 **2012 ACM Subject Classification** Theory of computation → Models of computation

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

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



© Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis and Cristian Riveros;
licensed under Creative Commons License CC-BY 4.0

???

Editors: ???; Article No. ??; pp. ??:1–??:20



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

27 **1 Introduction**

28 Imagine we would like to monitor whether the behavior of a server is correct. The run of
 29 the server can be abstracted by an infinite stream $w = a_1a_2a_3 \dots \in \Sigma^\omega$, where Σ is a finite
 30 alphabet of possible events. The events are disclosed one at a time on the input, and at every
 31 moment we should tell whether the prefix consisting of the events observed so far is correct.

32 A simple yet expressive formalism for describing properties of such *data streams* is provided
 33 by classical finite automata. For example, suppose we would like to verify the property
 34 that a certain resource is being used by at most one process. Assume that the alphabet is
 35 $\Sigma = \{o, r\} \cup \Gamma$, where o denotes a request of the resource, r denotes a release of the resource,
 36 and Γ contains other immaterial events. The streams satisfying the discussed property can
 37 be then characterized as those where every prefix is accepted by the two-state automaton \mathcal{A}
 38 of Figure 1. Here, a state indicates whether the resource is currently available or not.

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

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

54 For finite automata, the dynamic acceptance problem can be solved easily with update
 55 time $\mathcal{O}(|\mathcal{A}|)$, as follows. After reading a prefix u , the data structure stores the subset of states
 56 $S \subseteq Q$ in which the automaton may be after reading u (in general, we allow the automaton
 57 to be non-deterministic). Upon receiving the next input symbol, the set S is updated by
 58 applying the possible transitions on every state in S . Moreover, telling whether \mathcal{A} accepts
 59 the current input prefix boils down to checking whether S contains an accepting state. Both
 60 the update and the query described above can be implemented in time linear in $|\mathcal{A}|$.

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

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

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

79 Note that in the setting of timed automata, the same technique that worked for finite
 80 automata will not work so easily. The reason is that for a finite automaton \mathcal{A} , the set
 81 of configurations in which \mathcal{A} may be is a subset of the set of control states, whose size is
 82 bounded by the size of \mathcal{A} . On the other hand, a configuration of a timed automaton consists
 83 of a control state and a tuple of clock values, so the number of possible configurations is a
 84 priori unbounded. Concretely, after reading a prefix of length n , there may be as many as
 85 $\mathcal{O}(n^k)$ different configurations which the given k -clock timed automaton may possibly reach,
 86 due to non-determinism and clock resets. Efficient maintenance of this configuration set in a
 87 data structure poses the main conceptual challenge in this paper.

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

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

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

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

112 The dynamic acceptance problem that we consider here resembles the setting of *streaming*
 113 *algorithms*; see e.g. [5, 13, 21] for works with a similar motivation. In this context, a typical

114 problem is to compute (possibly approximately) some statistics or an aggregate function
 115 over the sequence of data, where the main point is to assume severe restrictions on the space
 116 usage. Note that in our setting, we focus on obtaining low time complexity per update and
 117 query, rather than optimizing the space complexity. In this respect, our work leans more
 118 towards the area of dynamic data structures, in particular dynamic query evaluation [9, 22].
 119 For Boolean properties several papers [25, 26, 6] have considered streaming algorithms for
 120 testing membership in regular and context-free languages. Another variant of the problem
 121 was considered in [18, 17, 16], where the regular property is verified on the last N letters of
 122 the stream, instead of the entire prefix up to the current position.

123 The closest to our setting is the work [28], which studies the dynamic evaluation problem
 124 for monoids over a sliding window, and describes a data structure that can be updated in
 125 constant time for a fixed finite monoid. When the monoid is finite, the considered problem is
 126 basically the same as monitoring whether the input stream restricted to the sliding window
 127 is accepted by a finite automaton. We explain in Appendix A that, in this case, the problem
 128 can be reduced to the dynamic acceptance problem for a special form of timed automaton.

129 2 Preliminaries

130 **Finite automata.** A *finite automaton* is a tuple $\mathcal{A} = (\Sigma, Q, I, E, F)$, where Σ is a finite
 131 alphabet, Q is a finite set of states, $E \subseteq Q \times \Sigma \times Q$ is a transition relation, and $I, F \subseteq Q$
 132 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
 133 $\rho = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ where $(q_{i-1}, a_i, q_i) \in E$ for all $i = 1, \dots, n$. Moreover, ρ is a
 134 *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
 135 \mathcal{A} on w .

136 **Timed automata.** Let X be a finite set of clocks, usually denoted x, y, \dots . A *clock valuation*
 137 is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ from clocks to non-negative reals. *Clock conditions* are formulas
 138 defined by the grammar: $C_X := \mathbf{true} \mid \mathbf{x} < c \mid \mathbf{x} > c \mid \mathbf{x} = c \mid C_X \wedge C_X \mid C_X \vee C_X$, where $\mathbf{x} \in X$
 139 and $c \in \mathbb{R}_{\geq 0}$. By a slight abuse of notation, we also denote by C_X the set of clock conditions
 140 over X . Given a clock condition γ and a valuation ν , we say that ν *satisfies* γ and write
 141 $\nu \models \gamma$, if the arithmetic expression obtained from γ by substituting each clock \mathbf{x} with its
 142 value $\nu(\mathbf{x})$ evaluates to true.

143 A *timed automaton* is a tuple $\mathcal{A} = (\Sigma, Q, X, I, E, F)$, where Q, Σ, I, F are defined exactly
 144 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
 145 transition relation. We say that $c \in \mathbb{R}_{\geq 0}$ is a *clock constant* of \mathcal{A} if c appears in some clock
 146 condition of a transition from E . A *configuration* of \mathcal{A} is a pair (q, ν) , where $q \in Q$ and ν is a
 147 clock valuation. Recall that finite automata process words over a finite alphabet Σ ; likewise,
 148 timed automata process timed words over an alphabet of the form $\Sigma \uplus \mathbb{R}_{>0}$, with Σ finite.

149 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
 150 $\rho = (q_0, \nu_0) \xrightarrow{e_1} (q_1, \nu_1) \xrightarrow{e_2} \dots \xrightarrow{e_n} (q_n, \nu_n)$, where each (q_i, ν_i) is a configuration and
 151 \blacksquare if $e_i \in \mathbb{R}_{>0}$, then $q_{i+1} = q_i$ and $\nu_{i+1}(\mathbf{x}) = \nu_i(\mathbf{x}) + e_i$ for all $\mathbf{x} \in X$;
 152 \blacksquare 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
 153 $\nu_{i+1}(\mathbf{x}) = 0$ or $\nu_{i+1}(\mathbf{x}) = \nu_i(\mathbf{x})$ depending on whether $\mathbf{x} \in Z$ or $\mathbf{x} \in X \setminus Z$.

154 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
 155 are reset when firing the transition. Note that the values of the other clocks stay unchanged.
 156 An example of a one clock timed automaton was given in the introduction (see Figure 1).

157 A run ρ as above is *successful* if $q_0 \in I$, $\nu_0(\mathbf{x}) = 0$ for all $\mathbf{x} \in X$, and $q_n \in F$. A word
 158 $w \in (\Sigma \cup \mathbb{R}_{>0})^*$ is *accepted* by \mathcal{A} if there is a successful run of \mathcal{A} on w .

159 **Size of an automaton.** The size of a finite automaton $\mathcal{A} = (\Sigma, Q, I, E, F)$ is defined as
 160 $|\mathcal{A}| = |Q| + |E|$. This is asymptotically equivalent to essentially every possible definition of
 161 size of a finite automaton that can be found in the literature. The size of a timed automaton
 162 $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ is instead defined as $|\mathcal{A}| = |Q| + |X| + \sum_{(p,a,\gamma,q,Z) \in E} |\gamma|$, where $|\gamma|$ is the
 163 number of atomic expressions (i.e. expressions of the form $x < c$, $x > c$, $x = c$) appearing in
 164 the clock condition γ . *Note that the size of a timed automaton does not take into account*
 165 *the magnitude of the clock constants.* These constants are specified with the automaton and
 166 stored in suitable floating-point memory cells (see the computation model below).

167 **Computation model.** As clock constants and time spans in the input stream are arbitrary
 168 real numbers, it is convenient to use the *real RAM model* of computation. This is a standard
 169 model with integer memory cells that can store integers and floating-point memory cells that
 170 can store real numbers. There are no bounds on the bit length or precision of the stored
 171 numbers. Basic arithmetic operations — addition, subtraction, multiplication, and division —
 172 can be performed in unit time, but modulo arithmetics and rounding are not included in the
 173 model. In fact, we do not use multiplication or division on real numbers either.

174 **3 The dynamic acceptance problem and main results**

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

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

185 The running time of each of these operations needs to be as low as possible. More precisely,
 186 we shall say that a data structure *supports dynamic acceptance in time* $f(s, n)$ if the first
 187 operation **init**(\mathcal{A}) takes at most $f(s, 0)$ time, and every subsequent execution of **accepted**(\cdot)
 188 or **read**(e) takes at most $f(s, n)$ time, where $s = |\mathcal{A}|$ and n is the number of stream elements
 189 consumed so far. Similarly, a data structure *supports dynamic acceptance in amortized time*
 190 $f(s, n)$ if the first operation **init**(\mathcal{A}) takes at most $f(s, 0)$ time, while every n subsequent
 191 operations **accepted**(\cdot) and **read**(e) take at most $n \cdot f(s, n)$ time in total. Ultimately, we
 192 are mostly interested in designing data structures where the complexity guarantee $f(s, n)$ is
 193 independent of n , that is, the (amortized) update time is a function of $|\mathcal{A}|$ only.

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

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

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

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

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

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

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

213 However, we establish a negative result for a slightly more powerful model of timed
 214 automata, called timed automata with additive constraints (see e.g. [8]). Formally, a *timed*
 215 *automaton with additive constraints* is defined exactly as a timed automaton — that is, as a
 216 tuple $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ consisting of an input alphabet, a set of states, a set of clocks,
 217 etc. — but clock conditions are now allowed to satisfy an extended grammar obtained by
 218 adding new rules of the form $(\sum_{x \in Z} x) \sim c$, where $Z \subseteq X$ and $\sim \in \{<, >, =\}$. For instance, one
 219 can write $x + y \leq c$, where c is a clock constant.

220 Our lower bound relies on the 3SUM conjecture, stated below. Recall that in the 3SUM
 221 problem we are given a set S of positive real numbers and the question is to determine
 222 whether there exist $a, b, c \in S$ satisfying $a + b = c$. It is easy to solve the problem in time $\mathcal{O}(n^2)$,
 223 where $n = |S|$; the 3SUM Conjecture asserts that this cannot be significantly improved.

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

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

230 ► **Theorem 3.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton*
 231 *\mathcal{A} with additive constraints such that there is no data structure that, when initialized on \mathcal{A} ,*
 232 *supports dynamic acceptance in time $\mathcal{O}(n^{1-\delta})$ for any $\delta > 0$, where n is the length of the*
 233 *consumed stream prefix.*

234 The proof of the above theorem is in Appendix B, together with a broader discussion of the
 235 3SUM Conjecture and of the extension of timed automata by additive constraints. Again,
 236 we do not know whether a negative result similar to the above one also holds for plain timed
 237 automata (without additive constraints).

238 4 Data structure: proof of Theorem 1

239 **Notation.** Let us fix, once and for all, the timed automaton $\mathcal{A} = (\Sigma, Q, X, I, E, F)$ with a
 240 single clock x that is provided upon initialization. By adding a non-accepting sink state, if
 241 necessary, we may assume that for every $q \in Q$ and $a \in \Sigma$, some transition over letter a can be
 242 always applied at q at any time. As \mathcal{A} uses only one clock, every configuration of \mathcal{A} can be
 243 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 .

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

246 Consider now an arbitrary stream $w \in (\Sigma \cup \mathbb{R}_{>0})^\omega$. For every $n \in \mathbb{N}$, let $w_n = w[1 \dots n]$
 247 be the n -element prefix of w . Recall that w_n can be thought of as the stream prefix that is
 248 disclosed after n operations $\text{read}(e)$. We say that a configuration (q, t) is *active* at step n if
 249 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
 250 (q, t) . We let K_n be the set of all configurations (q, t) that are active at step n .

251 **Partitioning the problem.** It is clear that the dynamic acceptance problem essentially boils
 252 down to designing an efficient data structure that maintains K_n upon reading subsequent
 253 elements from the stream. This data structure should offer a query on whether K_n contains
 254 an accepting configuration. The main observation is that configurations with clock values
 255 that are in the same order with respect to the clock constants C_1, \dots, C_k satisfy exactly
 256 the same clock conditions in E . Precisely, let us consider the partition of $\mathbb{R}_{\geq 0}$ into intervals
 257 $J_0, J_1, \dots, J_{2k+1}$, where $J_{2i} = [C_i, C_i]$, $J_{2i+1} = (C_i, C_{i+1})$, for all $p \in \{0, \dots, k\}$. The following
 258 assertion holds: for any two configurations (q, t) , (q', t') , with $t, t' \in J_i$ for some $0 \leq i \leq 2k + 1$,
 259 exactly the same transitions are available in (q, t) as in (q', t') .

260 For $n \in \mathbb{N}$ and $i \in \{0, \dots, 2k + 1\}$, let

$$261 \quad K_n[i] = \{ (q, t) \in K_n : t \in J_i \}.$$

262 The idea is to maintain each set $K_n[i]$ in a separate data structure. Each of these data
 263 structures follows the same design, which we call the *inner data structure*.

264 **Inner data structure: an overview.** Every inner data structure is constructed for an interval
 265 $J \in \{J_0, \dots, J_{2k+1}\}$. We will denote it by $\mathbb{D}[J]$, or simply by $\mathbb{D}[i]$ when $J = J_i$. Each structure
 266 $\mathbb{D}[J]$ stores a set of configurations L satisfying the following invariant: all clock values of
 267 configurations in L belong to J . In the final design we will maintain the invariant that the
 268 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
 269 treat L as an arbitrary set of configurations with clock values in J .

270 The inner data structure should support the following methods:

- 271 ■ Method $\text{init}(J)$ stores the interval J and initializes $\mathbb{D}[J]$ by setting $L = \emptyset$.
- 272 ■ Method $\text{accepted}()$ returns true or false, depending on whether or not L contains an
 273 accepting configuration, that is, a configuration (q, t) such that $q \in F$.
- 274 ■ Method $\text{insert}(q, t)$ adds a configuration (q, t) to L . This method will be always applied
 275 with a promise that $t \in J$ and $t \leq t'$ for all configurations (q', t') already present in L .
- 276 ■ Method $\text{updateTime}(r)$, where $r \in \mathbb{R}_{>0}$, increments the clock values of all configurations
 277 in L by r . All configurations whose clock values ceased to belong to J are removed from
 278 L , and they are returned by the method on output. This output is organised as a doubly
 279 linked list of configurations, sorted by non-decreasing clock values.
- 280 ■ Method $\text{updateLetter}(a)$ updates L by applying to all configurations in L all possible
 281 transitions over the given letter $a \in \Sigma$. Precisely, the updated set comprises all configura-
 282 tions (q, t) that can be obtained from configurations belonging to L before the update
 283 using transitions over a that do not reset the clock. The configurations $(q, 0)$ which can
 284 be obtained from L using transitions over a that do reset the clock are not included in
 285 the updated set, but are instead returned by the method as a doubly linked list.

286 In Section 4.2 we will provide an efficient implementation of the inner data structure, which
 287 is encapsulated in the following lemma.

288 ► **Lemma 4.** For each $J \in \{J_0, J_1, \dots, J_{2k+1}\}$, the inner data structure $\mathbb{D}[J]$ can be imple-
 289 mented so that methods $\text{init}()$, $\text{accepted}()$, $\text{insert}(\cdot, \cdot)$, and $\text{updateLetter}(\cdot)$ run in time
 290 $2^{\mathcal{O}(|\mathcal{A}|)}$, while method $\text{updateTime}(\cdot)$ runs in time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is the size of its output.

291 We postpone the proof of Lemma 4 and we show now how to use it to prove Theorem 1.
 292 That is, we design an *outer data structure* that monitors the acceptance of \mathcal{A} .

293 4.1 Outer data structure

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

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

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

300 Initialization. Given \mathcal{A} , we store \mathcal{A} in the data structure and we read the clock constants
 301 $0 = C_0 < C_1 < \dots < C_k$ from \mathcal{A} . Then we initialize $2k+1$ copies $\mathbb{D}[0], \dots, \mathbb{D}[2k+1]$ of the inner
 302 data structure by calling method `init(J)` for each interval J among $J_0, J_1, \dots, J_{2k+1}$. Finally,
 303 for each initial state q , we apply method `insert($q, 0$)` on $\mathbb{D}[0]$. As $K_0 = \{(q, 0) : q \in I\}$, after
 304 this we have that Invariant (I1) holds for $n = 0$.

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

308 Update by a time span. Suppose the next element from the stream is a time span $r \in \mathbb{R}_{>0}$. We
 309 update the outer data structure as follows. First, we apply method `updateTime(r)` to each
 310 data structure $\mathbb{D}[i]$. This operation increments the clock values of all configurations stored
 311 in $\mathbb{D}[i]$ by r , but may output a set of configurations whose clock values ceased to fit in the
 312 interval J_i . Recall that this set is organised as a doubly linked list of configurations, sorted
 313 by non-decreasing clock values; call this list S_i . Now, we need to insert each configuration
 314 (q, t) that appears on those lists into the appropriate data structure $\mathbb{D}[j]$, where j is such
 315 that $t \in J_j$. However, we have to be careful about the order of insertions: we process the lists
 316 $S_{2k+1}, S_{2k}, \dots, S_0$ in this precise order, and each list S_i is processed from the end, that is,
 317 following the non-increasing order of clock values. When processing a configuration (q, t)
 318 from the list S_i , we find the index $j > i$ such that $t \in J_j$ and apply the method `insert(q, t)`
 319 on the structure $\mathbb{D}[j]$. In this way the condition required by the `insert` method — that
 320 $t \leq t'$ for every configuration (q', t') currently stored in $\mathbb{D}[j]$ — is satisfied. It is also easy to
 321 see that Invariant (I1) is preserved after the update.

322 Update by a letter. Suppose the next symbol read from the stream is a letter $a \in \Sigma$. We
 323 update the outer data structure as follows. First, we apply method `updateLetter(a)` to
 324 each data structure $\mathbb{D}[i]$. This operation applies all possible transitions on letter a to all
 325 configurations stored in $\mathbb{D}[i]$, and outputs a list of configurations R_i where the clock got
 326 reset. All these configurations have clock value 0, hence the length of R_i is at most $|Q|$. It
 327 now suffices to insert all the configurations $(q, 0)$ appearing on all the lists R_i to $\mathbb{D}[0]$ using
 328 method `insert($q, 0$)`. We may do this in any order, as the condition required by the `insert`
 329 method is trivially satisfied. Again, Invariant (I1) is clearly preserved after the update.

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

332 From Lemma 4 it readily follows that each of the following operations takes time $2^{\mathcal{O}(|\mathcal{A}|)}$:
 333 initialization, a query about the acceptance, and an update by a letter. As for an update
 334 by a time span $r \in \mathbb{R}_{>0}$, by Lemma 4 the complexity of such an update is $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \sum_{i=0}^{2k+1} |S_i|$,
 335 where S_0, \dots, S_{2k+1} are the sets returned by the applications of method `updateTime(r)` to

336 data structures $\mathbb{D}[0], \dots, \mathbb{D}[2k+1]$, respectively. We need to argue that the amortized time
 337 complexity of all these updates is bounded by $2^{\mathcal{O}(|\mathcal{A}|)}$.

338 Consider the following definition: a clock value $t \in \mathbb{R}_{\geq 0}$ is *active* at step n if K_n contains
 339 a configuration with clock value t . Observe that upon an update by a time span $r \in \mathbb{R}_{> 0}$, the
 340 set of active clock values simply gets shifted by r , while upon an update by a letter $a \in \Sigma$
 341 it stays the same, except that clock value 0 may also become active. Since at step 0 the
 342 only active clock value is 0, we conclude that for every $n \in \mathbb{N}$, at most $n+1$ active clock
 343 values may have appeared until step n . Note that there may be at most $|Q|$ different active
 344 configurations with the same active clock value, hence the complexity of each update by a
 345 time span is bounded by $2^{\mathcal{O}(|\mathcal{A}|)} \cdot |Q|$ times the number of active clock values that change the
 346 interval J_i to which they belong, where we imagine that each active clock value is shifted by
 347 the time span. As every active clock value can change its interval at most $2k+1$ times, and
 348 the total number of active values that appear until step n is at most $n+1$, we conclude that
 349 the total time spent on updates by time spans throughout the first n steps is bounded by
 350 $2^{\mathcal{O}(|\mathcal{A}|)} \cdot |Q| \cdot (2k+1) \cdot (n+1) = 2^{\mathcal{O}(|\mathcal{A}|)} \cdot n$. Hence, the amortized time complexity is $2^{\mathcal{O}(|\mathcal{A}|)}$.

351 Finally, note that in the case of discrete streams each set S_i consists of configurations
 352 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
 353 complexity of an update by a time span is bounded by $2^{\mathcal{O}(|\mathcal{A}|)}$, without any amortization.

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

355 4.2 Inner data structure

356 We now describe the inner data structure $\mathbb{D}[J]$ and prove Lemma 4. Let us fix an interval
 357 $J \in \{J_0, \dots, J_{2k+1}\}$. We denote by L the set of configurations currently stored by the inner
 358 data structure $\mathbb{D}[J]$. It is convenient to represent L by a function $\lambda: \mathbb{R}_{\geq 0} \rightarrow 2^Q$ defined by

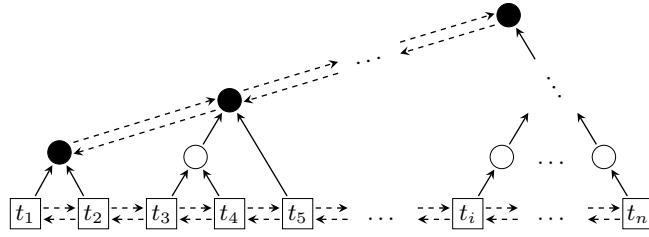
$$359 \quad \lambda(t) = \{q \in Q : (q, t) \in L\}.$$

360 We let \widehat{L} be the set of all clock values that are *active* in L , that is, \widehat{L} comprises all $t \in \mathbb{R}_{\geq 0}$
 361 such that $\lambda(t) \neq \emptyset$. Recall that we assume that $\widehat{L} \subseteq J$.

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

370 Updating by a letter is more problematic, as we need to apply the transition relation of
 371 the automaton \mathcal{A} to all the configurations of L simultaneously. In the data structure we
 372 store a partition of the active clock values \widehat{L} according to their images under $\lambda(\cdot)$, so that for
 373 each block of this partition (whose number is at most $2^{|Q|}$), we can simultaneously update
 374 all corresponding configurations in constant time. There is a caveat here: it is possible that
 375 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.
 376 That is, the blocks of the partition may require merging upon updates. We resolve this issue
 377 by representing the partition in a *forest*, similarly as the union-find data structure would do.
 378 The key point is that the height of this forest can be kept bounded by $2^{|Q|}$.

379 **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.

- 380 ■ The *clock*, denoted y , is a real that represents the total time elapsed since initialization.
- 381 ■ The *list*, denoted \mathbf{l} , stores the set of active clock values \widehat{L} .
- 382 ■ The *forest*, denoted \mathbf{f} , is built on top of the elements of \mathbf{l} and describes the function λ .
- 383 We describe the list and the forest in more details (the reader can refer to Figure 2).

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

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

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

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

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

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

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

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

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

412 We will maintain two invariants about these values. First, the sets $\mathbf{states}(\beta)$ and the ranks
 413 $\mathbf{rank}(\beta)$ should be pairwise different for distinct roots β of \mathbf{f} . Note that this means that \mathbf{f}
 414 always has at most $2^{|Q|} - 1$ roots. Second, for every root β , the *tree rooted at β* — which is
 415 the tree containing β and all its descendants in \mathbf{f} — has depth at most $\mathbf{rank}(\beta)$. Here, the
 416 *depth* of a forest is the maximum number of edges on a path from a leaf to a root, minus 1.
 417 Note that this implies that the depth of the forest \mathbf{f} is bounded by $2^{|Q|}$.

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

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

- 425 **I2.** For each node α on \mathbf{l} , the value $\text{clock}(\alpha) = y - \text{timestamp}(\alpha)$ belongs to J .
 426 **I3.** The nodes on \mathbf{l} are sorted by increasing clock values, or equally by decreasing timestamps.
 427 That is, $\text{timestamp}(\alpha) > \text{timestamp}(\text{next}(\alpha))$ for every non-last node α on \mathbf{l} .
 428 **I4.** Every root of \mathbf{f} has at least one child, and the leaves of \mathbf{f} are exactly all the nodes on \mathbf{l} .
 429 **I5.** The roots of \mathbf{f} carry pairwise different, non-empty sets of control states, and they have
 430 pairwise different ranks. Moreover, all the ranks belong to the set $\{1, 2, \dots, 2^{|\mathcal{Q}|}\}$.
 431 **I6.** For every root β of \mathbf{f} , the depth of the tree rooted at β is at most $\text{rank}(\beta)$.

432 **Implementation.** Now we show how to implement the methods $\text{init}(J)$, $\text{accepted}()$,
 433 $\text{insert}(q, t)$, $\text{updateTime}(r)$, and $\text{updateLetter}(a)$ in the data structure. Recall that all
 434 these methods should work in time $2^{\mathcal{O}(|\mathcal{A}|)}$, with the exception of $\text{updateTime}(r)$ which is
 435 allowed to work in time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is the size of its output. The description of each
 436 method is supplied by a running time analysis and an argumentation of the correctness,
 437 which includes a discussion on why the invariants stated above are maintained.

438 Removing nodes. Before we proceed to the description of the required methods, we briefly
 439 discuss an auxiliary procedure of removing a node from the list \mathbf{l} and from the forest \mathbf{f} , as
 440 this procedure will be used several times. Suppose we are given a node α on the list \mathbf{l} and
 441 we would like to remove it, which corresponds to removing from L all configurations (q, t)
 442 where $t = \text{clock}(\alpha)$ and $q \in \lambda(t)$. We can remove α from \mathbf{l} in the usual way. Then we remove
 443 α from \mathbf{f} as follows. First, we decrement the counter of children in the parent of α . If this
 444 counter stays positive then there is nothing more to do. Otherwise, we need to remove the
 445 parent of α as well, and accordingly decrement the counter of children in the grandparent
 446 of α . This can again trigger removal of the grandparent and so on. If eventually we need
 447 to remove a root of \mathbf{f} , we also remove it from the list \mathbf{r} in the usual way. Note that since
 448 by Invariants (I5) and (I6), the depth of \mathbf{f} is bounded by $2^{|\mathcal{Q}|}$, the whole procedure can be
 449 performed in time $\mathcal{O}(2^{|\mathcal{Q}|}) \leq 2^{\mathcal{O}(|\mathcal{A}|)}$. It is clear that all the invariants are maintained.

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

453 Acceptance query. The $\text{accepted}()$ method is implemented as follows. We iterate through
 454 the list \mathbf{r} to check whether there exists a root β of \mathbf{f} such that $\text{states}(\mathbf{f})$ contains any
 455 accepting state, say q . If this is the case, then by Invariant (I4) there is a node α on \mathbf{l} satisfying
 456 $\text{root}(\alpha) = \beta$, hence (q, t) is an accepting configuration that belongs to L , where $t = \text{clock}(\alpha)$.
 457 So we may return a positive answer from the query. Otherwise, all configurations in L have
 458 non-accepting states, and we may return a negative answer. Note that since by Invariant (I5)
 459 the list \mathbf{r} has length at most $2^{|\mathcal{Q}|} - 1$, the above procedure works in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

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

463 Let α be the first node on the list \mathbf{l} . Let $t' = \text{clock}(\alpha)$. By the promise, we have $t \leq t'$.
 464 We consider cases: either $t < t'$ or $t = t'$. The former case also captures the situation when \mathbf{l}
 465 is empty. When $t < t'$ or \mathbf{l} is empty, the new configuration (q, t) gives rise to a new active
 466 clock value t . Therefore, we create a new list node α_0 and insert it at the front of the list \mathbf{l} .
 467 We set the timestamp as $\text{timestamp}(\alpha_0) = y - t$, so that the node correctly represents the
 468 clock value t . It is clear that Invariants (I2) and (I3) are thus satisfied.

469 Next, we need to insert the new node α_0 to the forest \mathbf{f} . We iterate through the list
 470 \mathbf{r} in search for a root β that satisfies $\text{states}(\beta) = \{q\}$. In case there is one, we simply
 471 set $\text{parent}(\alpha_0) = \beta$ and increment $\#\text{children}(\beta)$. Otherwise, we construct a new root β_0
 472 with $\text{states}(\beta_0) = \{q\}$ and $\#\text{children}(\beta_0) = 1$, insert it at the front of the list \mathbf{r} , and set
 473 $\text{parent}(\alpha_0) = \beta_0$. To determine the rank of β_0 , we find the smallest integer $k \in \{1, \dots, 2^{|\mathcal{Q}|}\}$
 474 that is *not* used as the rank of any other root of \mathbf{f} . Observe that, by Invariant (I5), the forest
 475 \mathbf{f} has at most $2^{|\mathcal{Q}|} - 1$ roots, so there is always such a number k , and it can be found in time
 476 $2^{\mathcal{O}(|\mathcal{A}|)}$ by inspecting the list \mathbf{r} . We then set $\text{rank}(\beta_0) = k$. It is clear that this operation can
 477 be performed in time $2^{\mathcal{O}(|\mathcal{A}|)}$, and that Invariants (I4), (I5), and (I6) are maintained. For the
 478 last one, observe that the new leaf α_0 is attached directly under a root of \mathbf{f} , so no tree in \mathbf{f}
 479 existing before the insertion could have increased its depth.

480 We are left with the case when $t = t'$. We first compute the set X equal to $\lambda(t)$ before
 481 the insertion: it suffices to find $\text{root}(\alpha)$ in time $2^{\mathcal{O}(|\mathcal{A}|)}$ and read $X = \text{states}(\text{root}(\alpha))$.
 482 If $q \in X$ then the configuration (q, t) is already present in L , so there is nothing to do.
 483 Otherwise, we need to update the data structure so that $\lambda(t)$ is equal to $X \cup \{q\}$ instead of
 484 X . Consequently, we remove the node α from \mathbf{l} and from \mathbf{f} , using the operation described
 485 earlier, and we insert a new node α' at the front of \mathbf{l} , with the same timestamp equal to
 486 that of α . Thus, $\text{clock}(\alpha') = t$. We next insert the new node α' to the forest \mathbf{f} using the
 487 same procedure as described in the previous paragraph, but applied to the state set $X \cup \{q\}$
 488 instead of $\{q\}$. Again, it is clear that these operations can be performed in time $2^{\mathcal{O}(|\mathcal{A}|)}$, and
 489 the same argumentation shows that all the invariants are maintained.

490 *Update by a time span.* Next, we implement the method $\text{updateTime}(r)$, for $r \in \mathbb{R}_{>0}$. First,
 491 we increment y by r . Thus, for every node α in the list \mathbf{l} , the value $\text{clock}(\alpha)$ is incremented
 492 by r . However, the Invariant (I2) may have ceased to hold, as some active clock values could
 493 have been shifted outside of the interval J . The configurations with these clock values should
 494 be removed from the data structure and their list should be the output of the method.

495 We extract these configurations as follows. Construct an initially empty list of configura-
 496 tion \mathbf{lret} , on which we shall build the output. Iterate through the list \mathbf{l} , starting from its
 497 back. For each consecutive node α , compute $t = \text{clock}(\alpha)$. If $t \in J$, then break the iteration
 498 and return \mathbf{lret} , as there are no more configurations to remove. Otherwise, find $\text{root}(\alpha)$ in
 499 time $2^{\mathcal{O}(|\mathcal{A}|)}$, read $\lambda(t) = \text{states}(\text{root}(\alpha))$, and add at the front of \mathbf{lret} all configurations
 500 (q, t) for $q \in \lambda(t)$, in any order. Then remove α from the list \mathbf{l} and from the forest \mathbf{f} , and
 501 proceed to the previous node in \mathbf{l} (if there is none, finish the iteration).

502 By Invariant (I3), it is clear that in this way we remove from $\mathbb{D}[J]$ exactly all the
 503 configurations whose clock values got shifted outside of J , hence Invariants (I2) and (I3) are
 504 maintained. As the forest structure was influenced only by removals, Invariants (I4), (I5),
 505 and (I6) are maintained as well. Note that the output list \mathbf{lret} is ordered by non-decreasing
 506 clock values, as required. As for the time complexity, the procedure presented above takes
 507 time $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell'$, where ℓ' is the number of nodes that we remove from \mathbf{l} . As for every node α
 508 the set $\text{states}(\text{root}(\alpha))$ is non-empty and of size at most $|\mathcal{Q}|$, with every removed node we
 509 add to \mathbf{lret} between 1 and $|\mathcal{Q}|$ new configurations. Hence, we can also bound the complexity
 510 by $2^{\mathcal{O}(|\mathcal{A}|)} \cdot \ell$, where ℓ is the number of configurations that appear in the output list \mathbf{lret} .

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

518 First, we compute the output of the method, which is $\{(q, 0) : q \in \Psi(X)\}$ where X is the
 519 set of all states appearing in the configurations of L . Note that, by Invariant (I4), X can be
 520 computed in time $2^{\mathcal{O}(|\mathcal{A}|)}$ by iterating through the list \mathbf{r} and computing the union of sets
 521 `states(β)` for roots β appearing on it. Thus, the output can be computed in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

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

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

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

541 Finally, note that since the number of roots of \mathbf{f} is always bounded by $2^{|\mathcal{Q}|} - 1$, all the
 542 operations described above can be performed in time $2^{\mathcal{O}(|\mathcal{A}|)}$.

543 **5 Concluding remarks and future work**

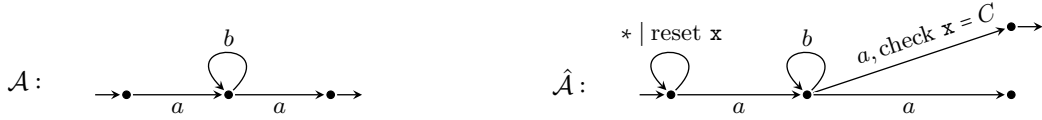
544 In this work we studied the dynamic acceptance problem for timed automata processing
 545 data streams. We designed a suitable data structure for one-clock timed automata, where
 546 the amortized update time depends only on the size of the automaton. We leave as an open
 547 question whether this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

548 More generally speaking, it seems that our work identifies dynamic variants of classic
 549 automata problems as a potential area of interest for the paradigm of parameterized dynamic
 550 data structures. More precisely, if the automaton model in question allows for the device to
 551 potentially be in an unbounded number of configurations, then the dynamic maintenance of
 552 this set of configurations is a computationally challenging problem, as show-cased in this
 553 paper. There are multiple models of devices where similar questions can be asked. Examples
 554 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](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://ecc.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](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](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.uni-siegen.de/handle/ubsi/1523>, doi:<http://dx.doi.org/10.25819/ubsi/464>.

- 607 17 Moses Ganardi, Danny Hucce, Daniel König, Markus Lohrey, and Konstantinos Mamouras.
608 Automata theory on sliding windows. In *35th Symposium on Theoretical Aspects of Computer*
609 *Science, STACS 2018*, pages 31:1–31:14, 2018.
- 610 18 Moses Ganardi, Danny Hucce, and Markus Lohrey. Querying regular languages over sliding
611 windows. In *36th IARCS Annual Conference on Foundations of Software Technology and*
612 *Theoretical Computer Science, FSTTCS 2016*, pages 18:1–18:14, 2016.
- 613 19 Alejandro Grez, Cristian Riveros, and Martín Ugarte. A formal framework for complex event
614 processing. In *22nd International Conference on Database Theory, ICDT 2019*, volume 127
615 of *LIPICs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL:
616 <https://doi.org/10.4230/LIPICs.ICDT.2019.5>, doi:10.4230/LIPICs.ICDT.2019.5.
- 617 20 Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. *J. ACM*,
618 65(4):22:1–22:25, 2018. URL: <https://doi.org/10.1145/3185378>, doi:10.1145/3185378.
- 619 21 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on
620 data streams. *External memory algorithms*, 50:107–118, 1998.
- 621 22 Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner.
622 Efficient query processing for dynamically changing datasets. *ACM SIGMOD Record*, 48(1):33–
623 40, 2019.
- 624 23 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture.
625 In *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1272–
626 1287. SIAM, 2016. URL: <https://doi.org/10.1137/1.9781611974331.ch89>, doi:10.1137/
627 1.9781611974331.ch89.
- 628 24 Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr.*
629 *Program.*, 78(5):293–303, 2009. URL: <https://doi.org/10.1016/j.jlap.2008.08.004>, doi:
630 10.1016/j.jlap.2008.08.004.
- 631 25 Philip M Lewis, Richard Edwin Stearns, and Juris Hartmanis. Memory bounds for recognition
632 of context-free and context-sensitive languages. In *6th Annual Symposium on Switching Circuit*
633 *Theory and Logical Design, SWCT 1965*, pages 191–202. IEEE, 1965.
- 634 26 Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expres-
635 sions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014.
- 636 27 Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *42nd ACM*
637 *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.
- 638 28 Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggrega-
639 tion in worst-case constant time. In *11th ACM International Conference on Distributed and*
640 *Event-based Systems*, pages 66–77, 2017.
- 642 29 Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic
643 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.
- 644 30 Stavros Tripakis. Fault diagnosis for timed automata. In *7th International Symposium on*
645 *Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2002*, pages 205–224,
646 2002. URL: https://doi.org/10.1007/3-540-45739-9_14, doi:10.1007/3-540-45739-9_14.
647
648



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

649 **A** Examples

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

658 Let $\mathcal{A} = (\Sigma, Q, I, E, F)$ be a finite automaton and C a positive integer defining the width
 659 of the sliding window. The membership problem of \mathcal{A} with a sliding window of width C
 660 consists of processing an arbitrary input $w = a_1a_2a_3 \dots$ over Σ from left to right, while
 661 maintaining the answer to the following query: *is the sequence of the last C consumed letters*
 662 *accepted by \mathcal{A} ?* The goal is design a data structure whose update time depends only on the
 663 automaton \mathcal{A} , and is independent of the size of the window C .

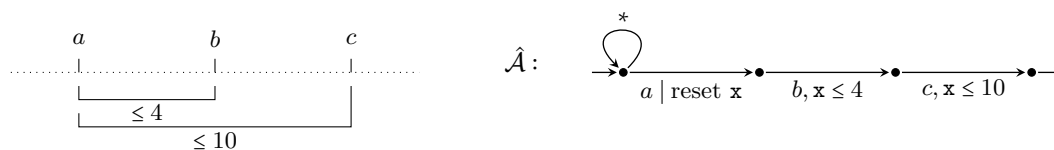
664 We now explain how the above problem can be reduced to our dynamic acceptance
 665 problem for timed automata. In this setting, we consider only streams that are discrete. In
 666 fact, we will enforce a slightly more restricted form of streams: we assume that every input
 667 stream belongs to the language $(\{1\} \cdot \Sigma)^\omega$, namely, that the letters from Σ are interleaved
 668 by the time unit 1. We map the input word $w = a_1a_2a_3 \dots$ to a corresponding discrete
 669 stream $\widehat{w} = 1a_11a_21a_3 \dots$, and modify the finite automaton \mathcal{A} to obtain a corresponding
 670 timed automaton $\widehat{\mathcal{A}}$, as follows. We introduce a new state \widehat{q} , which will be the only final
 671 state of $\widehat{\mathcal{A}}$, and a clock x . We then replace every transition (q, a, q') of \mathcal{A} with the transition
 672 $(q, a, \text{true}, q', \emptyset)$. Note that these transitions have a vacuous clock condition, hence they are
 673 applicable in $\widehat{\mathcal{A}}$ whenever the original transitions of \mathcal{A} are so. In addition, when the former
 674 transition (q, a, q') reaches a final state $q' \in F$, we also have a transition $(q, a, x = C, \widehat{q}, \emptyset)$
 675 in $\widehat{\mathcal{A}}$. Finally, we add looping transitions on the initial states that reset the clock, that is,
 676 transitions of the form $(q, a, \text{true}, q, \{x\})$, with $q \in I$ and $a \in \Sigma$. Figure 3 shows the timed
 677 automaton $\widehat{\mathcal{A}}$ corresponding to an automaton \mathcal{A} recognising ab^*a .

678 From the above construction it is clear that $\widehat{\mathcal{A}}$ accepts a prefix $1a_1 \dots 1a_n$ of \widehat{w} if and
 679 only if \mathcal{A} accepts the C -letter factor $a_{n-C+1} \dots a_n$ of w . Thus, the membership problem for
 680 \mathcal{A} in the C -width sliding window model is reduced to the dynamic acceptance problem for $\widehat{\mathcal{A}}$
 681 over the stream \widehat{w} . By Theorem 1, we know that there is a data structure that supports
 682 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
 683 one letter at a time from a word w , while answering in time $2^{\mathcal{O}(|\mathcal{A}|)}$ whether \mathcal{A} accepts the
 684 sequence of the last C consumed letters. Note that the complexity here is independent of C .

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

687
$$\varphi := a \mid \varphi; \varphi \mid \varphi \text{ WITHIN } t$$

688 where $a \in \Sigma$ and $t \in \mathbb{N}$. A word $w = a_1a_2 \dots a_n \in \Sigma^*$ matches an expression φ from the above
 689 grammar, denoted $w \models \varphi$, if one of the following cases holds:

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


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

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

693 Given a word $w = a_1 a_2 \dots$ and an expression φ , we would like to read w sequentially, as in
 694 a stream, and decide, at each position $n = 1, 2, \dots$, whether the prefix $w_n = a_1 \dots a_n$ matches
 695 a fixed expression φ . One can reduce this latter problem to our monitoring problem for timed
 696 automata, by using a discrete timed word $\hat{w} = 1a_1 1a_2 1 \dots$ as before and by translating the
 697 expression φ into an appropriate timed automaton. We omit the straightforward details of
 698 the translation of a CEL expression to an equivalent timed automaton, and we only remark
 699 that every occurrence of the `WITHIN` operator in an expression corresponds to a condition
 700 on a specific clock in the equivalent timed automaton. This means that, in general, the
 701 translation may require a timed automaton with multiple clocks. However, there are simple
 702 cases (which we do not characterize here) where, even in the presence of nested `WITHIN`
 703 operators, one can construct an equivalent timed automaton with a single clock. For example,
 704 consider the expression $\varphi = ((a; b) \text{ WITHIN } 4); c \text{ WITHIN } 10$, which describes a sequence
 705 containing three (possibly not contiguous) events a, b, c , with a and b at distance at most 4
 706 and a and c at distance at most 10. Figure 4 shows a single-clock timed automaton that
 707 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
 708 and only if $a_1 a_2 \dots a_n \models \varphi$. In this case one can validate any input stream against a fixed
 709 expression φ in time that is constant per input letter, by simply reducing to our dynamic
 710 acceptance problem for single-clock timed automata and discrete timed words.

711 **B Lower bound for two-clock timed automata with additive** 712 **constraints**

713 In this section, we prove a complexity lower bound for a variant of the dynamic acceptance
 714 problem. Ideally, we would like to prove that there is a timed automaton \mathcal{A} with two clocks
 715 such that no data structure can support dynamic acceptance for \mathcal{A} in time depending only
 716 on $|\mathcal{A}|$. This would imply that our result (Theorem 1) for the dynamic acceptance problem
 717 for single-clock timed automata cannot be generalised to the multiple-clock setting. We are
 718 not able to establish optimality in this sense.

719 We can however prove a result along the same line, by considering timed automata
 720 extended with additive constraints, that is, having clock conditions of the form $(\sum_{x \in Z} x) \sim c$.
 721 To give some background, let us discuss in more detail the power of this extension. Allowing
 722 additive constraints is a nontrivial extension of timed automata and in particular it makes
 723 the emptiness problem undecidable [8, Theorem 2]. However, undecidability holds when
 724 at least four clocks are available. Moreover, it is shown that for timed automata with

725 additive constraints with two clocks the emptiness problem is decidable; and the proof is a
 726 straightforward modification of the standard region construction [8, Proposition 1].

727 Our lower bound is based on the 3SUM Conjecture, which we restate below for convenience.
 728

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

732 The 3SUM Conjecture was introduced by Gajentaan and Overmars [14, 15] in a stronger
 733 form, which postulated the non-existence of *sub-quadratic* algorithms, that is, achieving
 734 running time $o(n^2)$. This formulation was refuted by Grønlund and Pettie [20], who gave an
 735 algorithm for 3SUM with running time $\mathcal{O}(n^2/(\log n/\log \log n)^{2/3})$ in the real RAM model,
 736 which can be improved to $\mathcal{O}(n^2(\log \log n)^2/\log n)$ when randomization is allowed. However,
 737 the existence of a strongly sub-quadratic algorithm is conjectured to be hard.

738 Recall that in the 3SUM problem we are given a set S of positive real numbers and the
 739 question is to determine whether there exist $a, b, c \in S$ satisfying $a + b = c$. We remark that
 740 the original phrasing of the conjecture allows non-positive numbers on input and asks for
 741 $a, b, c \in S$ such that $a + b + c = 0$. It is easy to reduce this standard formulation to our setting,
 742 for example by replacing S with $S' = \{3M + x : x \in S\} \cup \{6M - x : x \in S\}$, where M is any
 743 real satisfying $M > |a|$ for all $a \in S$.

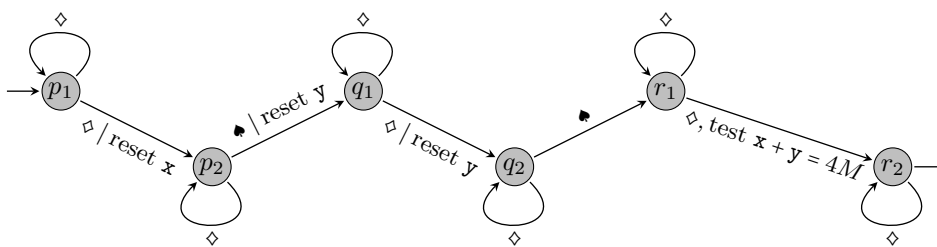
744 The 3SUM Conjecture has received significant attention in the recent years, as it was
 745 realised that it can be used as a base for tight complexity lower bounds for a variety of discrete
 746 graph problems, including questions about efficient dynamic data structures [1, 3, 23, 27]. In
 747 this setting, it is common to assume the integer formulation of the conjecture: there exists
 748 $d \in \mathbb{N}$ such that the 3SUM problem where the input numbers are integers from the range
 749 $[-n^d, n^d]$ cannot be solved in strongly sub-quadratic time, assuming the word RAM model
 750 with words of bit length $\mathcal{O}(\log n)$. It is straightforward to verify that the construction we
 751 are going to present in this section can be turned into an analogous lower bound assuming
 752 the integer formulation of the 3SUM Conjecture. For this, we would need to amend the
 753 formulation of the monitoring problem by assuming that the input stream is expected to have
 754 total length at most N , the clock constants and the time spans in the stream are integers of
 755 bit length at most M , and the data structure solving the monitoring problem should work in
 756 the word RAM model with words of bit length $\mathcal{O}(M + \log N)$.

757 We now prove Theorem 3, restated below for convenience. That is, we provide a lower
 758 bound for the dynamic acceptance problem for two-clock timed automata with additive
 759 constraints under the 3SUM Conjecture.

760 ► **Theorem 3.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton
 761 \mathcal{A} with additive constraints such that there is no data structure that, when initialized on \mathcal{A} ,
 762 supports dynamic acceptance in time $\mathcal{O}(n^{1-\delta})$ for any $\delta > 0$, where n is the length of the
 763 consumed stream prefix.*

764 Our approach is similar in spirit to the other lower bounds on dynamic problems, which
 765 we mentioned above [1, 3, 23, 27]. We first prove 3SUM-hardness of deciding acceptance by
 766 a timed automaton with additive constraints in the static setting. We then show that any
 767 data structure that supports monitoring in amortized strongly sub-linear time would violate
 768 the 3SUM-hardness of the former static acceptance problem, thus proving Theorem 3.

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



■ **Figure 5** Timed automaton for reducing 3SUM.

770 ► **Lemma 7.** *If the 3SUM Conjecture holds, then there is a two-clock timed automaton \mathcal{A} with*
 771 *additive constraints for which there is no algorithm that, given a timed word $w \in (\Sigma \uplus \mathbb{R}_{>0})^*$*
 772 *as input, where Σ is a two-letter alphabet, decides whether \mathcal{A} accepts w in time $\mathcal{O}(n^{2-\delta})$ for*
 773 *any $\delta > 0$ and for $n = |w|$.*

774 **Proof.** We construct a two-clock timed automaton \mathcal{A} with additive constraints and an
 775 algorithm that given a set S of n positive reals, outputs a word $w \in (\Sigma \uplus \mathbb{R}_{>0})^*$ such that
 776 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
 777 convenient to first present the construction of w from S . Then we present the automaton \mathcal{A}
 778 and analyze its runs on w .

779 Let $M = 1 + \max_{s \in S} |s|$. By sorting S we may assume that $S = \{s_1, s_2, \dots, s_n\}$, where
 780 $0 < s_1 < \dots < s_n < M$. We set $\Sigma = \{\diamond, \spadesuit\}$. The word is defined as

$$781 \quad w = u \spadesuit u \spadesuit v,$$

782 where

$$783 \quad u = 2(M - s_n) \diamond 2(s_n - s_{n-1}) \diamond 2(s_{n-1} - s_{n-2}) \diamond \dots \diamond 2(s_2 - s_1) \diamond 2(s_1 - 0);$$

$$784 \quad v = (M - s_n) \diamond (s_n - s_{n-1}) \diamond (s_{n-1} - s_{n-2}) \diamond \dots \diamond (s_2 - s_1) \diamond.$$

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

788 The automaton \mathcal{A} is depicted in Figure 5. It uses two clocks, named x and y . Note that
 789 all the transitions have trivial (always true) clock conditions, apart from the transition from
 790 r_1 to r_2 , where we check that the sum of clock values is equal to $4M$. The only initial state
 791 is p_1 , the only accepting state is r_2 .

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

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

802 Next, observe that the moment of reading the second occurrence of \spadesuit in w must coincide
 803 with firing the transition from q_2 to r_1 . Between the first and the second symbol \spadesuit the

804 automaton consumes the second factor u , and during this the clock x increases exactly by the
 805 sum of the time spans within u , i.e. by $2M$. On consuming the second factor u , the clock y is
 806 reset once, and precisely when firing the transition from q_1 to q_2 , which happens on reading
 807 one of the occurrences of \diamond in u . Again, if this happens when reading the j -th occurrence of
 808 \diamond , then, after the reset, y is incremented by exactly $2s_j$ units. We conclude the following.

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

811 Finally, after consuming the last factor v , the automaton can move to the accepting
 812 state r_2 if and only if at some point, upon reading an occurrence of \diamond , the condition
 813 $x + y = 4M$ holds. Observe that the sum of the first k numbers encoded in v is equal to
 814 $M - s_{n-k+1}$. Hence, after parsing those numbers, the set of possible clock valuations is
 815 $\{(x = 2a + 2M + M - c, y = 2b + M - c) : a, b \in S\}$, for some choice of $c \in S$. Moreover, the
 816 latter valuations satisfy the condition $x + y = 4M$ if and only if $a + b = c$.

817 Based on the above arguments, we infer that a successful run like ρ exists on input w if
 818 and only if there are $a, b, c \in S$ such that $a + b = c$. To conclude the proof, we observe that
 819 if an algorithm could decide whether \mathcal{A} accepts w in time $\mathcal{O}(n^{2-\delta})$ for any $\delta > 0$, then by
 820 combining this algorithm with the presented construction, one could solve 3SUM in time
 821 $\mathcal{O}(n^{2-\delta})$. This would contradict the 3SUM Conjecture. \blacktriangleleft

822 Theorem 3 now follows almost directly from the previous lemma. Consider the timed
 823 automaton \mathcal{A} provided by Lemma 7. If a data structure as in the statement of the theorem
 824 existed, then using this data structure one could decide in strongly sub-quadratic time
 825 whether any input timed word w is accepted by \mathcal{A} , by simply applying the sequence of
 826 $\text{read}(\cdot)$ operations corresponding to w , followed by the query $\text{accepted}()$.