Bounded-Memory Runtime Enforcement of Timed Properties

Saumya Shankar, Srinivas Pinisetty, Thierry Jéron

Bounded-Memory Runtime Enforcement of Timed Properties

Saumya Shankar
Indian Institute of Technology Bhubaneswar, India

Srinivas Pinisetty
Indian Institute of Technology Bhubaneswar, India

Thierry Jéron
Univ Rennes, Inria, IRISA, France

Abstract

Runtime Enforcement (RE) is a monitoring technique aimed at correcting possibly incorrect executions w.r.t. a set of formal requirements (properties) of a system. In this paper, we consider enforcement monitoring of real-time properties. Thus, executions are modelled as timed words and specifications as timed automata. Moreover, we consider that the enforcer has the ability to delay events by storing or buffering them into its internal memory (and releasing them when the property is finally satisfied) and suppressing events when no delaying is appropriate. Practically, in an implementation, the internal memory of the enforcer is finite.

In this paper, we propose a new RE paradigm for timed properties, where the memory of the enforcer is bounded/finite, to address practical applications with memory constraints and timed specifications. Bounding the memory presents a number of difficulties, e.g., how to accommodate a timed event into the memory when the memory is full, s.t., regardless of the course of action we choose to handle this situation, the behaviour of the bounded enforcer should not significantly differ from that of the unbounded enforcer. The problem of how to optimally discard events when the buffer is full is significantly more difficult in a timed environment where the progress of time affects the satisfaction or violation of a property. We define the bounded-memory RE problem for timed properties and develop a framework for regular timed properties specified as timed automata. The proposed framework is implemented in Python, and its performance is evaluated. From experiments, we discovered that the enforcer has a reasonable execution time overhead.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Automata over infinite objects; Software and its engineering → Software verification and validation

Keywords and phrases Formal methods, Runtime enforcement, Bounded-memory, Timed automata

Digital Object Identifier 10.4230/LIPIcs.TIME.2023.6

Funding This work has been partially supported by The Ministry of Human Resource Development, Government of India (SPARC P#701), IIT Bhubaneswar Seed Grant (SP093).

1 Introduction

Runtime Enforcement (RE) [8, 11, 2, 6, 19] is a monitoring technique to ensure that a system conforms to a set of formal requirements (properties) at runtime. It does so by employing an enforcement monitor (enforcer), which modifies an untrustworthy (not satisfying the property) sequence of input events into a trustworthy (satisfying the property) sequence of output events. This transformation of an input sequence into an output sequence should be constrained by the so-called *Soundness* (the enforcer should only output correct sequences), *Transparency* (the enforcer should not modify correct input sequences), *Optimality* (actions should be released as soon as possible as output), etc. properties.

We focus on the enforcement of regular timed properties, where the enforcement of a property φ is done on the fly (online). The input sequence of events is made up of actions and delays between them. The general schema is as follows: an enforcer is placed between an event emitter (which emits a sequence of events σ as an input to the enforcer) and an event receiver (which receives the corrected sequence of events o from the enforcer). The enforcer can increase the delays between the events to satisfy the desired timed property. Thus, when a sequence of events is received by the enforcer that is currently not satisfying the property, the enforcer stores or delays those events until a time/events come that result in the satisfaction of the property. Now, for that purpose, the enforcer is equipped with an internal memory (which we refer to as a *buffer* throughout the paper). Moreover, we consider that the enforcer can also suppress input events when it determines that the input events cannot be corrected by delaying, whatever their continuation.

In usual RE mechanisms such as [8, 11], the buffer of the enforcer is regarded as infinite/unbounded. But in the case of a real implementation of the enforcer, this assumption is obviously not realistic: an internal buffer is inevitably bounded [9, 24]. Thus, a situation may arise, where an event needs to be buffered (since it is not currently satisfying the property but can satisfy the property in the future with the arrival of more events), and the buffer is full. Now, to make room for this incoming event, one can arbitrarily remove some events from the buffer, or just discard the received event. However, these naive approaches can result in deviation in behaviour from the unbounded enforcer.

[23] studies RE for untimed properties with a bounded buffer, i.e., it gives a framework where the enforcer tackles the situation when the buffer is finite in an optimal way (minimal dropping of events from the buffer ("cleaning"), minimal deviation from the unbounded enforcer). The framework in [23] (referred to as *Bounded-Memory Runtime Enforcement*) synthesised an enforcer, for a given regular property φ, with the maximum size of buffer k, that takes as input a word σ and outputs a word o that (i) satisfies φ (soundness), (ii) is a prefix or subword of input σ (transparency), (iii) the output is as long as possible (optimality) and equivalent to one produced by an unbounded-memory enforcer (∞-compatible).

For various domains such as safety-critical systems, cyber-physical systems, and communication protocols, the correct functioning of their software systems depends crucially on real-time considerations (timing constraints and deadlines) where the time between events matters. Thus, we have timed properties to specify and verify models of real-time systems. They allow expressing constraints on the time that should elapse between (sequences of) events. Timed properties can be formally expressed using models such as *timed automata* [1]. By enforcing timed properties on these systems, we can ensure that the system behaviour adheres to these timing requirements, thereby guaranteeing its correctness and reliability.

RE allows for dynamic monitoring during execution. It provides a mechanism to detect violations of the timed properties at runtime (during execution of the system) and respond accordingly, by taking corrective actions. This enables proactive management of timing-related issues and helps prevent potential failures. Thus, various formal RE monitor synthesis approaches for timed properties, where properties are expressed as timed automata (or its variant) have been proposed [7, 15, 17, 18, 16].

Motivation. A framework for RE of "timed" properties with memory constraints on the buffer of the enforcer, however, has not been investigated yet, which is required since in a real-time safety-critical system, the properties are timed properties and the buffer is constrained by memory limitations.

Let us understand the usefulness of the bounded-memory enforcer for timed properties by an example. Consider a wireless sensor network in which the sensor nodes move throughout the environment working to gather and process information (e.g., temperature, humidity,

air quality, etc.) about their surroundings at specific intervals. These nodes typically consist of a low power microcontroller capable of limited information processing, sensors to capture specific data from the environment, memory to store collected data, and a radio to transmit data between nodes. Data buffers, used to handle large flows of network data, often consume large amounts of memory and therefore must be carefully allocated. So, if there are memory constraints on the whole sensor system, then the limited memory in sensor systems necessitates rethinking the data buffering model. Thus, whenever these systems are safeguarded by an enforcer which has constraints on the buffer size and we do not want to lose any random event stored in the buffer, then our work outshines others as it works on the principle of suppressing idempotent events (data) which makes insignificant differences in many cases.

Also, often the timing of the data captured by the sensor nodes is critical for maintaining temporal relevance. Consequently, this highlights the requirement for an enforcer that ensures adherence to timed properties.

Contributions. We introduce the first formal framework for bounded-memory runtime enforcement of timed properties modelled as timed automata[1]. We tackle the problem of obtaining an enforcer with memory constraints on the buffer, given a timed property. The enforcer intervenes when an execution is about to violate the property by its following abilities: delaying events in a buffer (increase the absolute dates of events of the observed input while allowing to shorten the delay between some events), releasing them when the property is finally satisfied, and suppressing (a minimal number of) events if there is no other way to avoid a property violation or a buffer overflow. The notions of soundness, monotonicity, and optimality are similar to the ones commonly used in the other RE frameworks [7]. Transparency is, however, simplified into two parts to account for cases where input events are corrected by just delaying or when suppression is also required. Moreover, we propose the notion of optimal suppression, which discards events when there is no possibility of finding a correct delay for an event.

The contributions can be summarized as follows:

1. We define an enforcer as one dedicated to a desired timed property φ, that transforms words.
2. We define the constraints that should be satisfied by the enforcer.
3. We present algorithms describing how the proposed enforcer can be implemented.
4. We implemented the proposed algorithms in Python and evaluated them using example properties. All our results are formalised and proven.

Due to space constraints, the algorithms and performance evaluation are provided at Appendix B, and sketches of proofs are provided at our github repository BMRE_timed.

2 Related Work

Runtime enforcement. *RE* is introduced in [21] by Schneider, where security policies are specified by security automata, a variant of the Büchi automaton. Later, many works extended the work of Schneider, e.g., Ligatti et al. introduced edit automata [10, 11] which

[1] Timed automata provide an explicit modeling of timing constraints and behaviour, including the ability to represent clock variables, timeouts, and synchronization mechanisms. This explicit representation makes timed automata well-suited for systems where precise timing constraints are crucial, such as real-time or embedded systems. Note that a timed temporal logic such as MTL can also be considered as an alternative for specifying properties, which can be transformed into timed automata [12].

not only recognise (truncate) the incorrect sequence of events but also correct those using edit functions. These edit functions perform suppression and insertion of input sequences (along with truncation). He also pioneered mandatory results automata [5], which have to provide a result to the target application before it can see the further action it wants to take.

Runtime enforcement of timed properties. The RE approaches are used to protect safety-critical systems. These are real-time systems, so the properties that need to be enforced are generally timed properties (properties with timing constraints on the sequence of events). Alur et al. in [1] pioneered *timed automata* to simulate the behaviour of real-time systems. Since then, different formal RE monitor synthesis approaches have been proposed for timed properties modelled by timed automata, e.g., [7, 15, 17, 18, 22].

Tools for runtime monitoring of timed properties. Several tools have been proposed to monitor the correctness of real-time systems. For example, the tool RT-MaC [20] verifies timeliness and reliability correctness properties at runtime. The Analog Monitoring Tool (AMT) [13] monitors the temporal properties of continuous signals. LARVA [4] can be used for the runtime verification of real-time properties of Java programs. The tool in [3] (implemented in Larva) presents dynamic communicating automata with timers and events to describe the properties of systems to monitor the temporal and contextual properties of Java programs. Tool TiPEX [14], as proposed in [7], implements the enforcement monitoring algorithms for timed properties.

Runtime enforcement with memory limitations. The approaches mentioned above do not consider any memory restrictions on the enforcer. There are a few works that take into consideration the memory limitations of the enforcer, e.g., the work by Fong in [9] and Talhi et al. in [24]. However, these approaches primarily concentrate on characterising the set of enforceable properties in a memory constrained environment. [23] introduces a framework for enforcement with bounded memory, restricted to untimed properties.

We extend the work in [23] and [7] to develop a framework for the bounded-memory RE of timed properties. To the best of our knowledge, our framework is the first to define how to synthesise an enforcer for a timed property that offers a solution when the memory of the enforcer is full at runtime.

3 Preliminaries and notations

We first recall some basic notions about untimed languages in Sect. 3.1. We then recall timed words and languages in Sect. 3.2 and talk about timed properties as timed automata in Sect. 3.3. At last, we introduce some preliminaries to RE of timed properties in Sect. 3.4.

3.1 Untimed languages

Languages. A (finite) word w is a finite sequence of elements of finite alphabet Σ. The length of w is the number of elements in w and is denoted as $|w|$. The empty word over Σ is denoted by ϵ. The sets of all words and non-empty words are denoted by Σ^* and Σ^+ respectively. A language (property) $\mathcal{L}$ over Σ is any subset of Σ^*.

The concatenation of two words w and w' is denoted by $w \cdot w'$. A word w' is a prefix of the word w, denoted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$, and $w' \prec w$, if additionally $w' \neq w$; conversely w is said to be an extension of w'.

The set $\text{pref}(w)$ stands for the *set of prefixes* of w. Subsequently, $\text{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathcal{L}} \text{pref}(w)$ is the set of prefixes of words in $\mathcal{L}$.

Given two words u and v, we define $w = v^{-1} \cdot u$ as the residual of u by v, s.t. $v \cdot w = u$, if this word exists i.e., if v is a prefix of u. Intuitively, $v^{-1} \cdot u$ is the suffix of u after reading prefix v. By extension, for a language $\mathcal{L} \subseteq \Sigma^*$ and a word $v \in \Sigma^*$, the residual of $\mathcal{L}$ by v is the language $v^{-1} \cdot \mathcal{L} \stackrel{\text{def}}{=} w \in \Sigma^* | v \cdot w \in \mathcal{L}$. It is the set of suffixes of words which, concatenated to v, belong to $\mathcal{L}$. In other words, $v^{-1} \cdot \mathcal{L}$ is the set of suffixes of words in $\mathcal{L}$ after reading the prefix v.

A word $w' = a_1 ... a_n$ is a subword/subsequence of w, denoted $w' \triangleleft w$, if w' can be obtained by deleting some letters from w or, equivalently, $w = w_0 a_1 w_1 ... a_n w_n$ for some $w_0, ..., w_n \in \Sigma^*$. We use the terms subword and subsequence interchangeably.

For a word w and $i \in [1, |w|]$, the i-th letter of w is denoted by $w_{[i]}$. Given a word w and integers i, j, s.t. $1 \leq i \leq j \leq |w|$, the suffix of word w starting from index i is denoted by $w_{[i \cdots]}$ and the subword from index i to j by $w_{[i \cdots j]}$.

Given a n-tuple of symbols $e = (e_1, ..., e_n)$, for $i \in [1, n]$, we define $\Pi_i(e)$ as the projection of e on its i-th element ($\Pi_i(e) = e_i$).

3.2 Timed words and languages

Let Σ be a finite alphabet of actions and $\mathbb{R}_{\geq 0}$ denotes the set of non-negative real numbers. An event is a tuple $(t, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$, where $\text{date}(t, a) \stackrel{\text{def}}{=} t$ is the absolute time instant at which action $\text{act}(t, a) \stackrel{\text{def}}{=} a$ occurs.

A timed word over Σ denoted $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$, is a finite sequence of non-decreasing events ($i \leq j \implies t_i \leq t_j$), ranging over $(\mathbb{R}_{\geq 0} \times \Sigma)^*$. We denote the starting date of σ by $start(\sigma) \stackrel{\text{def}}{=} t_1$, and its ending date by $end(\sigma) \stackrel{\text{def}}{=} t_n$ (with the convention that the starting and ending dates for the empty timed word ϵ are equal to 0).

The set of timed words over Σ is denoted by $tw(\Sigma)$. A timed language is any subset $\mathcal{L} \subseteq tw(\Sigma)$.

Note that, even though the alphabet $(\mathbb{R}_{\geq 0} \times \Sigma)$ is infinite in this case, previous notions and notations defined in the untimed case (related to length, prefix, subword/subsequence, etc.) naturally extend to timed words.

Concatenating timed words, on the other hand, takes greater care because, when concatenating two timed words, one must make sure that the result is a timed word, i.e., dates must not be decreasing. When we observe that the ending date of the first timed word does not exceed the starting date of the second one, this is ensured. Formally, let $\sigma = (t_1, a_1) \cdots (t_n, a_n)$ and $\sigma' = (t'_1, a'_1) \cdots (t'_m, a'_m)$ be two timed words with $end(\sigma) \leq start(\sigma')$[2], their concatenation is $\sigma \cdot \sigma' \stackrel{\text{def}}{=} (t_1, a_1) \cdots (t_n, a_n) \cdot (t'_1, a'_1) \cdots (t'_m, a'_m)$. By convention $\sigma \cdot \epsilon = \epsilon \cdot \sigma = \epsilon$. Concatenation is undefined otherwise.

The untimed projection of σ is $\Pi_\Sigma(\sigma) \stackrel{\text{def}}{=} a_1 \cdot a_2 \cdots a_n \in \Sigma^*$, i.e., dates are ignored.

3.3 Timed properties as timed automata

A timed automaton is a model used to specify the properties of a series of events where the timing between the events matters. In this section, we introduce timed automata as a formalism for specifying timed properties.

[2] Throughout the paper, when we consider concatenation of two timed words σ and σ' as $\sigma \cdot \sigma'$, we assume that $end(\sigma) \leq start(\sigma')$.

Timed automata . A Timed Automaton (TA) [1] is a finite automaton extended with a finite set of real valued clocks. A clock valuation for X, where $X = \{x_1, \cdots, x_k\}$ is a finite set of clocks, is an element of $\mathbb{R}^X_{\geq 0}$, i.e., a function from X to $\mathbb{R}^X_{\geq 0}$. $v + \delta$ is the valuation assigning $v(x) + \delta$ to each clock x of X, where $v \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$ (delay since previous action). Given a set of clocks $X' \subseteq X$, $v[X' \leftarrow 0]$ is the clock valuation v where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of guards. These are clock constraints defined as Boolean combinations of simple constraints of the form $x \bowtie c$ with $x \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and $v \in \mathbb{R}^X_{\geq 0}$, we denote $v \models g$ when g holds according to v.

▶ **Definition 1** (Timed automata). *TA is a tuple $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, s.t. L is a finite set of locations. $l_0 \in L$ is initial location. X is a finite set of clocks. Σ is a finite set of actions. $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the transition relation. $F \subseteq L$ is the set of accepting locations.*

▶ **Example 2** (Timed automata). The automaton $\mathcal{A}_P$ in Figure 1 denotes a timed automaton of a prototype property P, with $L = \{l_0, l_1, l_2, l_3\}$ as the set of locations, l_0 the initial location, and l_2 the accepting location (denoted by double circles). The alphabet of events is $\Sigma = \{h, r\}$. The automaton has one clock x.

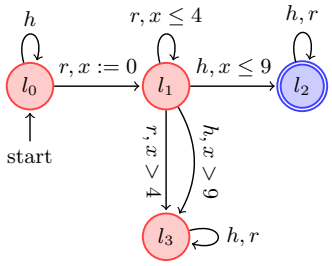

Figure 1 $\mathcal{A}_P$.

The transitions can be understood as follows: from initial location l_0 and on reception of input action h, $\mathcal{A}_P$ remains at the same location. It makes a transition to location l_1 with the clock x being reset (to keep an eye on the reception of next r or h action) on reception of input action r. From location l_1, if action h is received within 9 t.u., then $\mathcal{A}_P$ makes a transition to the accepting location l_2, otherwise goes to violating (non-accepting) location l_3. Moreover, from location l_1, if action r is received within 4 t.u., $\mathcal{A}_P$ remains at the same location l_1, otherwise goes to location l_3. From locations l_2 and l_3, on input actions $\{h, r\}$, $\mathcal{A}_P$ remains at same respective locations.

▶ **Definition 3** (Semantics of TA). *The semantics of a TA is a timed transition system $[\![A]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ where the (infinite) set of states is given by $Q = L \times \mathbb{R}^X_{\geq 0}$. $q_0 = (l_0, v_0)$ is the initial state where v_0 is the valuation that maps each clock in X to 0. The set of accepting states is given by $Q_F = F \times \mathbb{R}^X_{\geq 0}$. The set of transition labels is given by $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$. A label is a pair consisting of a delay and an action. The transition relation $\rightarrow \subseteq Q \times \Gamma \times Q$ is a set of transitions of the form $t_r : (l, v) \xrightarrow{(\delta, a)} (l', v')$ with $v' = (v + \delta)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ s.t. $v + \delta \models g$ for $\delta \in \mathbb{R}_{\geq 0}$.*

Deterministic and complete TA. $\mathcal{A}$ is *deterministic* whenever for any two distinct transitions (l, g_1, a, Y_1, l'_1) and $(l, g_2, a, Y_2, l'_2) \in \Delta$, $g_1 \wedge g_2$ is unsatisfiable. $\mathcal{A}$ is *complete* whenever for any location $l \in L$ and an action $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labelled by a evaluates to *true*.

In this work, we only consider deterministic and complete TA [1, 7]. So, wherever we say a TA, it refers to a deterministic and complete TA.

A run ρ from $q \in Q$ is a sequence of moves in $[\![\mathcal{A}]\!] : \rho = q \xrightarrow{(\delta_1,a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n,a_n)} q_n$, for some $n \in N$, where the trace of a run ρ is the timed word $(t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$, (where the date t_n of action a_n is the sum of all the delays i.e., $\Sigma_{i=1}^{n} \delta_i$). The set of runs from $q_0 \in Q$ is denoted by $Run(\mathcal{A})$. The subset of runs accepted by $\mathcal{A}$, i.e., when $q_n \in F_G$ is denoted by $Run_{F_G}(\mathcal{A})$. $\mathcal{L}(\mathcal{A})$ denotes the language accepted by $\mathcal{A}$ from its initial state q_0, whereas $\mathcal{L}(\mathcal{A}, q)$ denotes the language accepted by $\mathcal{A}$ from state q.

3.4 Preliminaries to runtime enforcement of timed properties

The following preliminaries will be useful for RE.

- obs(σ, t)*:* Given $t \in \mathbb{R}_{\geq 0}$, and $\sigma \in tw(\Sigma)$, the observation of σ at date t (i.e., obs(σ, t)) is the maximal prefix of σ that can be observed at date t. Formally,
$$\text{obs}(\sigma, t) \stackrel{\text{def}}{=} \max_{\preccurlyeq}\{\sigma' \in (\mathbb{R}_{\geq 0} \times \Sigma)^* \mid \sigma' \preccurlyeq \sigma \wedge \text{end}(\sigma') \leq t\}.$$
- Delaying order $=_d$: For $\sigma, \sigma' \in tw(\Sigma)$, σ' delays σ (noted $\sigma' =_d \sigma$) iff they have the same untimed projection but the dates of events in σ' is greater than or equal to the dates of corresponding events in σ. Formally:
$$(\sigma' =_d \sigma) \stackrel{\text{def}}{=} (\Pi_\Sigma(\sigma') = \Pi_\Sigma(\sigma)) \wedge \forall i \in [1, |\sigma|] : date(\sigma'_{[i]}) \geq date(\sigma_{[i]}).$$
Sequence σ' is obtained from σ by keeping all actions, but with a potential increase in dates. For example, $(4, a) \cdot (7, b) \cdot (9, c) =_d (3, a) \cdot (5, b) \cdot (8, c)$. Note that delays between events may be decreased (e.g., between b and c), but absolute dates are increased.
- *Delayed prefix* $\preccurlyeq_d$: For $\sigma, \sigma' \in tw(\Sigma)$, we say that σ' is a delayed prefix of σ (noted $\sigma' \preccurlyeq_d \sigma$) iff the untimed projection of σ' is a prefix of the untimed projection of σ and the dates of events in σ' is greater than or equal to the dates of corresponding events in σ. Formally:
$$(\sigma' \preccurlyeq_d \sigma) \stackrel{\text{def}}{=} (\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)) \wedge \forall i \in [1, |\sigma'|] : date(\sigma'_{[i]}) \geq date(\sigma_{[i]}).$$
For example, $(4, a) \cdot (7, b) \preccurlyeq_d (3, a) \cdot (5, b) \cdot (8, c)$.
- Delaying subsequence order $\lhd_d$: For $\sigma, \sigma' \in tw(\Sigma)$, we say that σ' is a delayed subword/subsequence of σ (noted $\sigma' \lhd_d \sigma$) iff there exists a subsequence σ'' of σ such that σ' delays σ''. Formally: $(\sigma' \lhd_d \sigma) \stackrel{\text{def}}{=} \{\exists \sigma' \in tw(\Sigma) : (\sigma' =_d \sigma'' \wedge \sigma'' \lhd \sigma)\}$.
Sequence σ' is obtained from σ by first suppressing some actions and then increasing the dates of the actions that are kept. For example, $(5, a) \cdot (10, c) \lhd_d (3, a) \cdot (5, b) \cdot (8, c)$ (where event $(5, b)$ has been suppressed, while a and c are shifted in time).
- Lexical order $\preceq_{lex}$: For $\sigma, \sigma' \in tw(\Sigma)$, with same untimed projection (i.e., $\Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$), the order $\preceq_{lex}$ is defined inductively as follows: $\epsilon \preceq_{lex} \epsilon$, and for two events with identical actions (δ, a) and (δ', a), $(\delta, a) \cdot \sigma \preceq_{lex} (\delta', a) \cdot \sigma'$ if $\delta < \delta' \vee (\delta = \delta' \wedge \sigma \preceq_{lex} \sigma')$. This order is useful to select a unique timed word from a group of words with same untimed projection. For example $(4, a) \cdot (5, b) \cdot (8, c) \cdot (11, d) \preceq_{lex} (4, a) \cdot (5, b) \cdot (9, c) \cdot (10, d)$.
- Choosing a unique timed word with minimal duration $min_{\preceq_{lex},end}$:
Given a set of timed words having the same untimed projection, $min_{\preceq_{lex},end}$ chooses a timed word among timed words with minimal ending date w.r.t. the lexical order: first the set of timed words with minimal ending date are considered, and then, from these timed words, the (unique) minimal one is selected w.r.t. the lexical order. Formally, for a set $E \subseteq tw(\Sigma)$ such that $\forall \sigma, \sigma' \in E : \Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$ (i.e., such that all words have the same untimed projection), we have $min_{\preceq_{lex},end}(E) = min_{\preceq_{lex}}(min_{\preceq_{end}}(E))$ where $\sigma \preceq_{end} \sigma'$ if $end(\sigma') \geq end(\sigma)$, for $\sigma, \sigma' \in tw(\Sigma)$.
- Maximal strict prefix: The maximal strict prefix of $\sigma \in tw(\Sigma)$ that belongs to $\varphi \subseteq tw(\Sigma)$ is defined as $max^{\varphi}_{\prec,\epsilon}(\sigma) \stackrel{\text{def}}{=} max_{\preccurlyeq}(\{\sigma' \in (\mathbb{R}_{\geq 0} \times \Sigma)^* | \sigma' \prec \sigma \wedge \sigma' \in \varphi\} \cup \{\epsilon\})$.

The following notions of delayable (that checks whether a word is delayable w.r.t. the given property) are useful in defining constraints and respective enforcer.

- Function $\text{delayable1}_\varphi(\sigma)$: Given property $\varphi \subseteq tw(\Sigma)$, a timed word $\sigma \in tw(\Sigma)$, it returns the set of delayed words σ' of σ, s.t. σ' can be extended to satisfy φ in the future (i.e., $\sigma' \in \text{pref}(\varphi)$). Formally:
  $\text{delayable1}_\varphi(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in tw(\Sigma) : (\sigma' =_d \sigma) \wedge (\sigma' \in \text{pref}(\varphi))\}$.
- Function $\text{delayable2}_\varphi(\sigma_1, \sigma_2)$: Given property $\varphi \subseteq tw(\Sigma)$, and two timed words $\sigma_1, \sigma_2 \in tw(\Sigma)$, it returns a set of delayed words σ_2' of σ_2, where each word in the set returned should start at or after the ending date of σ_2, which is the date t of the last event (t, a) of σ_2, s.t. $\sigma_1 \cdot \sigma_2'$ can be extended to satisfy the property φ in the future. Formally,
  $\text{delayable2}_\varphi(\sigma_1, \sigma_2) \stackrel{\text{def}}{=} \{\sigma_2' \in tw(\Sigma) : (\sigma_2' =_d \sigma_2) \wedge (\sigma_1 \cdot \sigma_2' \in \text{pref}(\varphi)) \wedge (start(\sigma_2') \geq end(\sigma_2))\}$.
- Function $k^{\mathcal{L}}(\sigma_1, \sigma_2)$: Given $\mathcal{L} \subseteq tw(\Sigma)$ and two timed words $\sigma_1, \sigma_2 \in tw(\Sigma)$, function $k^{\mathcal{L}}(\sigma_1, \sigma_2)$ computes the set of timed words w that delay σ_2, start at or after the ending date of σ_2, s.t. when σ_1 is extended with w (i.e., $\sigma_1 \cdot w$), the word should belong to $\mathcal{L}$. Formally,
  $k^{\mathcal{L}}(\sigma_1, \sigma_2) \stackrel{\text{def}}{=} \{w \in \sigma_1^{-1} \cdot \mathcal{L} \mid (w =_d \sigma_2) \ \wedge \ (start(w) \geq end(\sigma_2))\}$.

4 Runtime enforcement in a timed context with unbounded memory

In this section, we define the enforcement monitoring framework with an unbounded buffer. We first define the expected constraints on the input/output behaviour of the enforcer in Sect. 4.1 and then define an enforcer dedicated to a desired timed property $\varphi \in tw(\Sigma)$ in Sect. 4.2 ; this is a reformulation of paper [7]; the paper contains all the proofs.

4.1 Constraints on an enforcer

We first define the constraints that should be satisfied by an enforcer before providing the actual definition of an enforcer in Sect. 4.2. The following constraints can serve as a specification of the expected behaviour of an enforcer for timed properties that has the ability to delay as well as suppress events.

▶ **Definition 4.** *(Constraints on an enforcer). An enforcer for a timed property $\varphi \subseteq tw(\Sigma)$ is a function $E^\varphi : tw(\Sigma) \to tw(\Sigma)$, satisfying the following constraints:*

Soundness	**(Snd)**	$\forall \sigma \in tw(\Sigma) : E^\varphi(\sigma) \models \varphi \vee E^\varphi(\sigma) = \epsilon$
Monotonicity	**(Mo)**	$\forall \sigma, \sigma' \in tw(\Sigma) : \sigma \preccurlyeq \sigma' \implies E^\varphi(\sigma) \preccurlyeq E^\varphi(\sigma')$
Transparency	**(Tr1)**	$\forall \sigma \in tw(\Sigma), \text{delayable1}_\varphi(\sigma) = \emptyset \implies E^\varphi(\sigma) \lhd_d \sigma$
	(Tr2)	$\forall \sigma \in tw(\Sigma), \text{delayable1}_\varphi(\sigma) \neq \emptyset \implies E^\varphi(\sigma) \preccurlyeq_d \sigma$

Soundness expresses that for any word $\sigma \in tw(\Sigma)$, the output produced by the enforcer (i.e., $E^\varphi(\sigma)$) should satisfy the property φ, as soon as it is non-empty. Monotonicity expresses that the output produced for the extension σ' of an input word σ (i.e., $E^\varphi(\sigma')$) extends the output produced for σ (i.e., $E^\varphi(\sigma)$), conveying that the output is a continuously growing timed word, i.e., for a given input timed word, what is output can only be changed by appending new events with greater dates. Transparency is defined using Tr_1 and Tr_2. Here, Tr_1 expresses that if no delayed word of $\sigma \in tw(\Sigma)$ exists that can satisfy the property in the future, then

the output of the enforcer for σ will be a delayed subword of σ (i.e., some events may be suppressed). The constraint Tr_2 expresses that if any delayed word of $\sigma \in tw(\Sigma)$ exists that can satisfy the property in the future, then the output of the enforcer for σ will be a delayed prefix of σ (i.e., none of the events are suppressed).

4.2 Definition of enforcement function

We now define an enforcement function / enforcer dedicated to a desired property φ. At an abstract level, it serves as a delayer with suppression, where suppression only occurs upon the reception of an event that inhibits any satisfaction of φ in the future.

▶ **Definition 5** (Enforcement function/ Enforcer). *The enforcer for a property $\varphi \subseteq tw(\Sigma)$ is the function $E^\varphi : tw(\Sigma) \to tw(\Sigma)$ defined as:*
$\forall \sigma \in tw(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall a \in \Sigma,$

$$E^\varphi(\sigma) = \Pi_1(\text{store}^\varphi(\sigma)), \text{ where}$$

$\text{store}^\varphi : tw(\Sigma) \to tw(\Sigma) \times tw(\Sigma)$ *is defined as:*

- $\text{store}^\varphi(\epsilon) = (\epsilon, \epsilon)$

$$\text{store}^\varphi(\sigma \cdot (t,a)) = \begin{cases} (\sigma_s \cdot min_{\preceq_{lex},end}(k^\varphi(\sigma_s, \sigma_{ca})), \epsilon) & \text{if } k^\varphi(\sigma_s, \sigma_{ca}) \neq \emptyset, \\ (\sigma_s, \sigma_c) & \text{if } k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) = \emptyset, \\ (\sigma_s, \sigma_{ca}) & \text{otherwise.} \end{cases}$$

with:

- $(\sigma_s, \sigma_c) = \text{store}^\varphi(\sigma)$
- $\sigma_{ca} = \sigma_c \cdot (t,a)$

The unbounded enforcer E^φ takes a timed word over Σ as input, and produces a timed word over Σ as output.
For a given input σ, function store^φ computes a pair (σ_s, σ_c) of timed words: i) σ_s is the prefix of maximal length for which the absolute dates have been computed to satisfy property φ; it is a delayed subsequence of the input σ; and it is extracted by the projection function Π_1 to produce output $E^\varphi(\sigma)$, ii) σ_c is a subsequence[3] of the remaining suffix of σ for which the releasing dates of events are still required to be computed. It is a temporary memory.

Enforcer E^φ incrementally computes a timed word according to the input timed word, and, inductively, is defined as follows: when the empty word ϵ is input, it produces (ϵ, ϵ). Otherwise, suppose that for the input σ, the result of $\text{store}^\varphi(\sigma)$ is (σ_s, σ_c) and consider a newly received event (t,a). Now, the new timed word to be corrected is $\sigma_{ca} = \sigma_c \cdot (t,a)$. There are three possible cases, according to the vacuity of the two sets $k^\varphi(\sigma_s, \sigma_{ca})$ and $k^{pref(\varphi)}(\sigma_s, \sigma_{ca})$. For any $\mathcal{L} \subseteq tw(\Sigma)$, definition of $k^{\mathcal{L}}(\sigma_s, \sigma_{ca})$ is given in Sect. 3.4.

$k^\varphi(\sigma_s, \sigma_{ca})$ is the set of timed words w that delay σ_{ca}, such that $\sigma_s \cdot w$ satisfies φ; and $k^{pref(\varphi)}(\sigma_s, \sigma_{ca})$ is the set of timed words w that delay σ_{ca}, such that some additional continuation w' may satisfy φ, i.e., $\sigma_s \cdot w \cdot w' \models \varphi$. Finally the three possible cases are:

- If $k^\varphi(\sigma_s, \sigma_{ca}) \neq \emptyset$ (and thus $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset$), it indicates that, for the timed word $\sigma_{ca} = \sigma_c \cdot (t,a)$, it is possible to choose its appropriate dates to satisfy φ. The minimal timed word in $k^\varphi(\sigma_s, \sigma_{ca})$ w.r.t. the lexicographic order is selected among those with a minimal ending date and appended to σ_s. Since all events memorised in $\sigma_c \cdot (t,a)$ are corrected and appended to σ_s, σ_c is set to ϵ.

[3] and not the complete suffix, since, some events may have been suppressed when no delaying allowed to satisfy φ, whatever is the continuation of σ, if any

- If $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) = \emptyset$ (and thus $k^{\varphi}(\sigma_s, \sigma_{ca}) = \emptyset$), it means that, for the current input $\sigma \cdot (t, a)$, whatever is its continuation, there is no chance to find a suitable delaying for (t, a). Thus, event (t, a) should be suppressed, leaving σ_c and σ_s unchanged.
- Otherwise, i.e., when $k^{\varphi}(\sigma_s, \sigma_{ca}) = \emptyset$ but $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset$, it means that for $\sigma_{ca} = \sigma_c \cdot (t, a)$, it is not yet possible to select its appropriate dates to satisfy φ, however, if the input is continued, if at all, there is still a chance to do it in the future. Thus, σ_c is modified into $\sigma_{ca} = \sigma_c \cdot (t, a)$ in memory, but σ_s is left unchanged.

▶ **Proposition 6 (Snd, Mo, Tr1, Tr2).** *Given some timed property $\varphi \subseteq tw(\Sigma)$, its enforcer E^{φ} as per Definition 5 satisfies Snd, Mo, Tr1, and Tr2 constraints as per Definition 4.*

▶ **Proposition 7** (Optimal Suppression)**.** *Given some timed property $\varphi \subseteq tw(\Sigma)$, its enforcer E^{φ} as per Definition 5 satisfies the following constraint:*

$$\forall \sigma \in tw(\Sigma),\ \ \exists \sigma_s, \sigma_c \in tw(\Sigma) : \text{store}^{\varphi}(\sigma) = (\sigma_s, \sigma_c)\ \wedge\ \forall (t, a) \in (\mathbb{R}_{\geq 0} \times \Sigma),\ t \geq end(\sigma_c) :$$
$$(\text{delayable2}_{\varphi}(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset \implies \forall \sigma_{\text{con}} \in tw(\Sigma) : start(\sigma_{\text{con}}) \geq t,$$
$$E^{\varphi}(\sigma \cdot (t, a) \cdot \sigma_{\text{con}}) = E^{\varphi}(\sigma \cdot \sigma_{\text{con}}))$$

(Opts)

For any input σ, for which the output of the $\text{store}^{\varphi}(\sigma)$ *is (σ_s, σ_c), the optimal suppression constraint expresses that, when σ_c is extended with a timed event (t, a) and if no delayed word of $\sigma_c \cdot (t, a)$ exists s.t. $\sigma_s \cdot \sigma_c \cdot (t, a)$ can be extended to satisfy the property φ in future (i.e.,* $\text{delayable2}_{\varphi}(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$*) then, the action a should be suppressed by the enforcer*[4].

▶ Remark 8. Snd, Mo, Tr1, Tr2 constraints outline an enforcer's expected input-output behaviour throughout the whole input sequence. However, it should be noted that it does not strongly constrain the output. These restrictions are specifically met by an enforcer that never produces any output. But, to be practical, a real enforcer should also offer some guarantees on the output sequence it produces in terms of delay, w.r.t. the input sequence. Such assurances are specified by the optimality property, which is provided in Appendix A.1.

▶ **Example 9** (Unbounded enforcer)**.** We illustrate how Definition 5 is applied to enforce a prototype property P of example 2, recognised by the automaton depicted in Fig. 1 with $\Sigma = \{h, r\}$, and the input timed word $\sigma = (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h) \cdot (5, r) \cdot (8, r) \cdot (9, h)$.

Table 1 shows evolution of the observed input timed word $\text{obs}(\sigma, t)$ and the output of function store^P when the input timed word is $\text{obs}(\sigma, t)$. The first element of the output of function store^P can be extracted by projection function Π_1 to produce output $E^P(\sigma)$. Variable t keeps track of physical time, i.e., it contains current date.

From the table, we can see that, at time $t = 8$, event $(8, r)$ is suppressed, as it is not possible to correct the input sequences (since, reception of this event is leading to a violating location l_3 in the timed automata 1). Till $t < 9$, the observed output is empty (since $\Pi_1(\text{store}^P(\text{obs}(\sigma, t))) = \epsilon$). When $t \geq 9$, the observed output is $(9, h) \cdot (10, h) \cdot (11, h) \cdot (12, h) \cdot (13, r) \cdot (17, h)$ (since $\Pi_1(\text{store}^P(\text{obs}(\sigma, t))) = (9, h) \cdot (10, h) \cdot (11, h) \cdot (12, h) \cdot (13, r) \cdot (17, h)$).

5 Runtime enforcement in a timed context with bounded-memory

In this section, we introduce the enforcement monitoring framework with a bounded buffer. We describe how expected constraints must be adapted by the enforcer in Sect. 5.1 and then define an enforcer dedicated to a desired timed property $\varphi \in tw(\Sigma)$ in Sect. 5.2.

[4] As stated in Tr1, suppression should only be done when necessary.

Table 1 Evolution of the unbounded enforcer for property P.

$t \in [0,1)$	$\text{obs}(\sigma,t) = \epsilon$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,\epsilon)$
$t \in [1,2)$	$\text{obs}(\sigma,t) = (1,h)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h))$
$t \in [2,3)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h)\cdot(2,h))$
$t \in [3,4)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)\cdot(3,h)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h)\cdot(2,h)\cdot(3,h))$
$t \in [4,5)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h))$
$t \in [5,8)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)\cdot(5,r)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)\cdot(5,r))$
$t \in [8,9)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)\cdot(5,r)\cdot(8,r)$ $\text{store}^P(\text{obs}(\sigma,t)) = (\epsilon,(1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)\cdot(5,r))$
$t \in [9,\infty)$	$\text{obs}(\sigma,t) = (1,h)\cdot(2,h)\cdot(3,h)\cdot(4,h)\cdot(5,r)\cdot(8,r)\cdot(9,h)$ $\text{store}^P(\text{obs}(\sigma,t)) = ((9,h)\cdot(10,h)\cdot(11,h)\cdot(12,h)\cdot(13,r)\cdot(17,h),\epsilon)$

A bounded-memory enforcer, denoted by $E^{\varphi,k}$, for a given timed property $\varphi \in tw(\Sigma)$ is equipped with a buffer of size k and should be able to transform an input timed word σ which is possibly incorrect w.r.t. φ into an output timed word that is correct w.r.t. φ.

Enforcer $E^{\varphi,k} : tw(\Sigma) \to tw(\Sigma) \times \{\bot, \top, stop\}$, outputs a tuple consisting of an output word, referred by $E^{\varphi,k}_{\text{out}}(\sigma)$, (element of $tw(\Sigma)$) and mode information, referred by $E^{\varphi,k}_{\text{mode}}(\sigma)$, (which is an element of $\{\top, \bot, stop\}$) permitting to warn the user, where $\top$, $\bot$, and *stop* respectively represent nominal mode indicating none of the events from the buffer were suppressed, degraded mode indicating some of the events from the buffer were suppressed, and *stop* mode indicating the enforcer cannot continue operating[5]. $\text{buff}(E^{\varphi,k}(\sigma))$ refers to the buffer content of enforcer $E^{\varphi,k}$, after reading σ.

5.1 Constraints on an enforcer

We here define the constraints that should be satisfied by an enforcer.

▶ **Definition 10** (Constraints on an enforcer). *A bounded enforcer for a timed property $\varphi \subseteq tw(\Sigma)$, equipped with a buffer of size k, is a function $E^{\varphi,k}$*

$E^{\varphi,k} : tw(\Sigma) \to tw(\Sigma) \times \{\bot, \top, stop\}$

satisfying the following constraints:

Soundness	***(SndB)***	$\forall \sigma \in tw(\Sigma) : E^{\varphi,k}_{out}(\sigma) \models \varphi \vee E^{\varphi,k}_{out}(\sigma) = \epsilon$
Monotonicity	***(Mo1B)***	$\forall \sigma,\sigma' \in tw(\Sigma) : \sigma \preccurlyeq \sigma' \implies E^{\varphi,k}_{out}(\sigma) \preccurlyeq E^{\varphi,k}_{out}(\sigma')$
	(Mo2B)	$\forall \sigma,\sigma' \in tw(\Sigma) : \sigma \preccurlyeq \sigma', (E^{\varphi,k}_{mode}(\sigma) = \bot \implies E^{\varphi,k}_{mode}(\sigma') = \bot)$
Transparency	***(Tr1B)***	$\forall \sigma \in tw(\Sigma), \text{delayable1}_\varphi(\sigma) = \emptyset \vee E^{\varphi,k}_{\text{mode}}(\sigma) = \bot \implies E^{\varphi,k}_{out}(\sigma) \lhd_d \sigma$
	(Tr2B)	$\forall \sigma \in tw(\Sigma), \text{delayable1}_\varphi(\sigma) \neq \emptyset \wedge E^{\varphi,k}_{\text{mode}}(\sigma) = \top \implies E^{\varphi,k}_{out}(\sigma) \preccurlyeq_d \sigma$

The notion of (SndB), (Mo1B), (Tr1B [6]) and (Tr2B [7]) are the same as in the unbounded case in Def. 4, with some notational variations and mode information being included as a consideration. (Mo2B) expresses that when the mode is degraded, it cannot return to nominal.

[5] Note that, intially the mode is considered to be nominal, i.e., $E^{\varphi,k}_{\text{mode}}(\epsilon) = \top$.

[6] Tr1B can alternatively be expressed as: $\forall \sigma \in tw(\Sigma), \forall (t,a) \in (\mathbb{R}_{\geq 0} \times \Sigma),\ E^{\varphi,k}_{\text{mode}}(\sigma) = \bot \vee \text{delayable1}_\varphi(\sigma \cdot a) = \emptyset \implies E^{\varphi,k}_{out}(\sigma \cdot a) \lhd_d \sigma \cdot a$

[7] Tr2B can alternatively be expressed as: $\forall \sigma \in tw(\Sigma), \forall (t,a) \in (\mathbb{R}_{\geq 0} \times \Sigma),\ E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge \text{delayable1}_\varphi(\sigma \cdot a) \neq \emptyset \implies E^{\varphi,k}_{out}(\sigma \cdot a) \preccurlyeq_d \sigma \cdot a$

5.2 Definition of enforcement functions

We now define a bounded-memory enforcer $E^{\varphi,k}$ dedicated to a desired property φ. The enforcer works as a delayer with suppression, where suppression happens upon reception of an event that prevents any satisfaction of φ in the future or when the buffer is full.

Preliminary to the enforcer. During enforcement of an input timed word σ, when an incoming event needs to be buffered and the buffer is full, then the buffer is cleaned (i.e., some events stored in the buffer are suppressed to make room for the incoming events). The cleaning has to be done in such a way that, the treatment of future events by the enforcer is not affected (i.e., the enforcer handles the future events the same way with or without cleaning of the buffer). This is guaranteed if the state reached in the property automaton remains the same, even if some events are removed from the buffer. The notion of property equivalence can be understood as follows:

Equivalence of two timed words. Two timed words $\sigma \in tw(\Sigma)$ and $\sigma' \in tw(\Sigma)$ are φ-equivalent, noted $\sigma \sim_\varphi \sigma'$ if runs ρ and ρ' from q_0 of $\mathcal{A}_\varphi$ (i.e., (l_0, v_0)) upon σ and σ' respectively i.e.,

$$\rho = q_0 \xrightarrow{(\delta_1,a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n,a_n)} q_n \text{ and } \rho' = q_0 \xrightarrow{(\delta'_1,a'_1)} q'_1 \cdots q'_{m-1} \xrightarrow{(\delta'_m,a'_m)} q_m$$

(where, the trace of a run ρ and ρ' are timed words $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ and $\sigma' = (t'_1, a'_1) \cdot (t'_2, a'_2) \cdots (t'_m, a'_m)$ respectively (with $t_n = \Sigma_{i=1}^{n} \delta_i$ and $t'_m = \Sigma_{i=1}^{m} \delta'_i$), of different lengths) end on q_n and q_m respectively s.t. $\mathcal{L}(\mathcal{A}_\varphi, q_n) = \mathcal{L}(\mathcal{A}_\varphi, q_m)$.

It means that if the respective runs from the initial state of the property automaton upon two timed words end on two states, from where the language accepted are identical, then we say that the words are property equivalent.

▶ **Definition 11.** *(Bounded enforcement function / enforcer.) A bounded enforcer for a property $\varphi \subseteq tw(\Sigma)$ is the function $E^{\varphi,k} : tw(\Sigma) \to tw(\Sigma) \times \{\top, \bot, stop\}$, and is defined as:*

$\forall \sigma \in tw(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall a \in \Sigma,$

$$E^{\varphi,k}(\sigma) = (\Pi_1(\text{store}^{\varphi,k}(\sigma)), \Pi_3(\text{store}^{\varphi,k}(\sigma))), \textit{ where:}$$

$\text{store}^{\varphi,k} : tw(\Sigma) \to tw(\Sigma) \times tw(\Sigma) \times \{\top, \bot, stop\}$ *is defined as:*

- $\text{store}^{\varphi,k}(\epsilon) = (\epsilon, \epsilon, \top)$
- $\text{store}^{\varphi,k}(\sigma \cdot (t, a)) =$

$$\begin{cases} (\sigma_s \cdot min_{\preceq_{lex},end}(k^\varphi(\sigma_s, \sigma_{ca})), \epsilon, \{\top, \bot\}) & \textit{if } k^\varphi(\sigma_s, \sigma_{ca}) \neq \emptyset, \\ (\sigma_s, \sigma_c, \bot) & \textit{if } k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) = \emptyset, \\ (\sigma_s, \sigma_{ca}, \{\top, \bot\}) & \textit{if } k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset \ \wedge |\sigma_{ca}| \leq k \\ (\sigma_s, \sigma_c, stop) & \textit{if } k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset \ \wedge \ |\sigma_{ca}| > k \\ & \quad \wedge \ \text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \emptyset \\ (\sigma_s, \text{Clean}^{\varphi,k}(\sigma_s, \sigma_{ca}), \bot) & \textit{if } k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset \ \wedge |\sigma_{ca}| > k \\ & \quad \wedge \ \text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) \neq \emptyset \end{cases}$$

with:

- $(\sigma_s, \sigma_c, \{\top, \bot\}) = \text{store}^{\varphi,k}(\sigma)$,
- $\sigma_{ca} = \sigma_c \cdot (t, a)$
- $E^{\varphi,k}_{\text{out}}(\sigma) = \Pi_1(E^{\varphi,k}(\sigma))$
- $E^{\varphi,k}_{\text{mode}}(\sigma) = \Pi_3(E^{\varphi,k}(\sigma))$
- $\text{buff}(E^{\varphi,k}(\sigma)) = \Pi_2(E^{\varphi,k}(\sigma))$

- $\text{Clean}^{\varphi,k} : tw(\Sigma) \times tw(\Sigma) \to tw(\Sigma)$
  $\text{Clean}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \sigma' \in \text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) : \forall \sigma'' \in \text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}),$
  $\sigma' \neq \sigma'' \ \wedge \ |\sigma'| > |\sigma''| \ \wedge \ (\text{index}(\sigma', \sigma_{ca}) \leq \text{index}(\sigma'', \sigma_{ca}))$
- $\text{index}(\sigma', \sigma_{ca}) = (i \in \mathbb{N} \mid i \in [1, |\sigma_{ca}|] : \sigma_{ca[i]} \neq \sigma'_{[i]})$

- $\text{Get\_SW}^{\varphi,k} : tw(\Sigma) \times tw(\Sigma) \to 2^{tw(\Sigma)}$
  $\text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \{\sigma'' \in tw(\Sigma) \mid \exists \sigma' \in \text{delayable2}^{\varphi}(\sigma_s, \sigma_{ca}) \wedge$
  $\exists i, j, k \in \mathbb{N} \wedge 1 \leq i \leq j < k :$
  $(\sigma'' = \sigma'_{[1\ldots i-1]} \cdot \sigma'_{[j+1\ldots k]}) \wedge (\sigma'_{[1\ldots i-1]} \cdot \sigma'_{[i\ldots j]} \cdot \sigma'_{[j+1\ldots k]} \sim_{\varphi} \sigma'')\}$

The bounded enforcer $E^{\varphi,k}$ takes a timed word over Σ as input, and produces a timed word over Σ and a sequence of modes (elements from the set $\{\top, \bot, \mathit{stop}\}$) as output. For a given input σ over Σ, function $\text{store}^{\varphi,k}$ computes a pair (σ_s, σ_c) of timed words over Σ (of which the first timed word is extracted by the projection function Π_1 to produce the output $E^{\varphi,k}_{\text{out}}(\sigma)$) and a sequence of mode as output (which is extracted by the projection function Π_3 to be the mode $E^{\varphi,k}_{\text{mode}}(\sigma)$). The pair (σ_s, σ_c) is same as in Def. 5.

Function $E^{\varphi,k}$ incrementally computes a timed word according to the input timed word and is defined inductively as follows: When the empty word ϵ is input, it produces $(\epsilon, \epsilon, \top)$. Otherwise, suppose that for input σ, the result of $\text{store}^{\varphi,k}(\sigma)$ is $(\sigma_s, \sigma_c, \{\top, \bot\})$ and consider a new received event (t, a). Now, the new timed word to correct is $\sigma_{ca} = \sigma_c \cdot (t, a)$. There are five possible cases, according to the vacuity of the two sets $k^{\varphi}(\sigma_s, \sigma_{ca})$ and $k^{pref(\varphi)}(\sigma_s, \sigma_{ca})$, whether the buffer is full, and if there exist any property equivalent word in the buffer:

- If $k^{\varphi}(\sigma_s, \sigma_{ca}) \neq \emptyset$. Since this case is similar to the first one in Def. 5, the same course of actions are followed. Mode remains unchanged.
- If $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) = \emptyset$. Since this case is similar to the second one in Def. 5, the same course of actions are followed. In addition, the mode changes to $\bot$.
- If $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset$, and $|\sigma_{ca}| \leq k$, i.e., it means that for $\sigma_{ca} = \sigma_c \cdot (t, a)$, it is not yet possible to select appropriate dates to satisfy φ, however, depending on the continuation of the input, if any, there is still a chance to do it in the future, and there is space in the buffer to accomodate this incoming event. Thus σ_c is modified into $\sigma_{ca} = \sigma_c \cdot (t, a)$ in memory, but σ_s and mode are left unmodified.
- If $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset$, $|\sigma_{ca}| > k$, and $\text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \emptyset$. In this case, the buffer is full. Thus, cleaning of the buffer is required. However, there does not exist a property equivalent word (of length $< k$) of the buffer contents to be able to replace the buffer contents. Thus, "safe" cleaning cannot be done and the enforcer stops operating (conveying this information by changing the mode to *stop*).
- If $k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset$, $|\sigma_{ca}| > k$, and $\text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) \neq \emptyset$. In this case, there exists a property equivalent word (of length $< k$) of the buffer contents to be able to replace the buffer contents. Thus, the function $\text{Clean}^{\varphi,k}$ is called to clean the buffer (received event is also considered for cleaning) in order to accommodate the event. σ_s is left unmodified, σ_c is replaced by the output of function $\text{Clean}^{\varphi,k}$, and the mode changes to $\bot$.

Function $\text{Clean}^{\varphi,k}$ takes two timed words over Σ as input. It produces a timed word as output, which should be a delayed subword of σ_{ca}. Also, the output word should be of maximal length, and the events discarded are the most *obsolete*[8] ones. For this purpose, function $\text{Get\_SW}^{\varphi,k}$ provides all delayed subwords σ'' of σ_{ca}. It does so by first finding all

[8] The earliest received events are considered for deletion in this approach; this is an implementation choice.

delayed words σ' (of length k) of σ_{ca} and then finding all the property equivalent words σ'' (of length $< k$) of every delayed word σ', if any. Function Clean$^{\varphi,k}$ selects the longest subwords among those and then chooses a unique subword, with the most *obsolete* action being discarded. This is done by comparing the indexes of σ_{ca} with the indexes of received subwords from function Get_SW$^{\varphi,k}$ using function index. The contents of the buffer are then substituted by the output of function Clean$^{\varphi,k}$.

▶ **Proposition 12 (SndB, Mo1B, Mo2B, Tr1B, Tr2B).** *Given some timed property $\varphi \subseteq tw(\Sigma)$ and the maximum buffer size k, let $n \in \mathbb{N}$ be the number of locations in $\mathcal{A}_\varphi$. If $k \geq n$, then the enforcer $E^{\varphi,k}$ as per Definition 11 satisfies SndB, Mo1B, Mo2B, Tr1B, and Tr2B constraints as per Definition 10.*

The notion of optimal suppression in the case of a bounded enforcer is same as in case of unbounded enforcer (Opts), with some notational variations; we omit the definition due to space constraints. Also, similar to Sect. A.1, some additional constraints to provide some guarantees on the output sequence produced by the enforcer in terms of length and delay are provided in Appendix A.2.

▶ **Example 13** (Bounded enforcer)**.** We consider example 9, and see (in Table 2) how Def. 11 can be applied with $k = 4$. Other parameters are the same as in Table 1. From the table 2, we can see that, at time $t = 5$, the buffer is already full, thus function Clean^{P,4} invokes function Get_SW^{P,4} to calculate the possible property equivalent subwords of the delayed word of $(1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h) \cdot (5, r)$.

Table 2 Evolution of the enforcer for property P.

$t \in [0, 1)$	$\text{obs}(\sigma, t) = \epsilon$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, \epsilon)$
$t \in [1, 2)$	$\text{obs}(\sigma, t) = (1, h)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (1, h))$
$t \in [2, 3)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (1, h) \cdot (2, h))$
$t \in [3, 4)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h) \cdot (3, h)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (1, h) \cdot (2, h) \cdot (3, h))$
$t \in [4, 5)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h))$
$t \in [5, 8)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h) \cdot (5, r)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (6, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$
$t \in [8, 9)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h) \cdot (5, r) \cdot (8, r)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = (\epsilon, (6, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$
$t \in [9, \infty)$	$\text{obs}(\sigma, t) = (1, h) \cdot (2, h) \cdot (3, h) \cdot (4, h) \cdot (5, r) \cdot (8, r) \cdot (9, h)$ $\text{store}^{P,4}(\text{obs}(\sigma, t)) = ((9, h) \cdot (10, h) \cdot (11, h) \cdot (12, r) \cdot (16, h), \epsilon)$

For example, some of the words returned by function Get_SW^{P,4} will be $((6, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$, $((5, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$, $((7, h) \cdot (8, h) \cdot (9, r))$, etc. Function Clean^{P,4} will first choose all the longest subwords among the set of delayed property equivalent subwords e.g., $((6, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$, $((5, h) \cdot (7, h) \cdot (8, h) \cdot (9, r))$, etc. Then it will choose a unique subword which is formed by deleting the most obsolete event, which is $(6, h) \cdot (7, h) \cdot (8, h) \cdot (9, r)$ in this presented example with event $(5, h)$ being suppressed while *cleaning*. All other cases are similar as in example 9. The final output is $(9, h) \cdot (10, h) \cdot (11, h) \cdot (12, r) \cdot (16, h)$.

▶ Remark 14 (Performance evaluation and analysis). We have provided an online algorithm and an experimentation framework in order to: i.) validate the viability of enforcement monitoring; and ii.) analyse the performance of the enforcer through experiments. Through performance analysis, we found that the enforcer had a reasonable execution time overhead (the average time taken in cleaning (per call) is found to be 0.019 s, which is low and reasonable[9]). The results are provided in Appendix B.

In the future, we plan to assess the framework within a more realistic scenario, taking into account various metrics. For instance, we aim to enhance the evaluation by increasing the complexity of the TAs, such as augmenting the number of locations within the TA. Additionally, we plan to explore the impact of varying buffer sizes, among other factors. This expanded evaluation will enable us to observe the growth patterns of time, memory, and other relevant aspects.

▶ Remark 15 (Stop mode of the enforcer and enforceable properties). Note that for any given property $\varphi \subseteq tw(\Sigma)$, for some input observation $\sigma \in tw(\Sigma)$ if the mode of the enforcer changes to *stop* upon σ, then it is an indication that the enforcer should halt and cannot continue any further. We have, $\forall \sigma \in tw(\Sigma), (E^{\varphi,k}_{\text{mode}}(\sigma) = stop) \implies \forall \sigma_{con}, E^{\varphi,k}_{\text{out}}(\sigma \cdot \sigma_{con}) = E^{\varphi,k}_{\text{out}}(\sigma)$.

This happens when "safe" buffer cleaning is not attainable, (i.e., there does not exist a property equivalent word (of length $< k$) of the buffer contents to be able to replace the buffer contents, when cleaning of the buffer is required), leading to halting of the enforcer and the mode getting changed to *stop*.

Ideally however, we expect that for a given property φ, $E^{\varphi,k}$ should continuously operate, and the mode of the enforcer should not change to *stop*. We call a given property φ, as continuously enfocerable as per Def. 10 and $E^{\varphi,k}$ an enforcer for φ as per Def. 11 if, for any input timed word $\sigma \in tw(\Sigma)$, mode is different from *stop*, i.e., $\forall \sigma \in tw(\Sigma), E^{\varphi,k}_{\text{mode}}(\sigma) \neq stop$.

To achieve this, we introduce (in Appendix C) specific syntactic and semantic conditions on the TA, s.t. if a TA satisfies these conditions, it guarantees that the enforcer never halts.

6 Conclusion and future work

In conclusion, this paper presents a novel framework for enforcing timed properties with memory constraints on the enforcer. The proposed approach intervenes in executions by delaying or suppressing events to prevent property violations or buffer overflows. The paper defines the necessary constraints for an enforcer and proposes a dedicated function that transforms words to enforce a desired property. The presented algorithms provide implementation details for the proposed approach. Overall, this work contributes to the field of runtime enforcement of timed properties and provides a valuable framework for ensuring system correctness with real-time constraints. In the future, we also plan to develop other alternative implementations of the proposed enforcement framework using other TA frameworks such as TChecker[10] (to obtain the zone graphs and for reachability analysis).

References

1 Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.

[9] Applications where this overhead is acceptable, the approach can be implemented.
[10] TChecker is a model-checking tool for real-timed systems. https://www.labri.fr/perso/herbrete/tchecker/

2 Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: Runtime enforcement for reactive systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 533–548, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46681-0_51.

3 Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems*, pages 135–149, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

4 Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 33–37, 2009. doi:10.1109/SEFM.2009.13.

5 Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *Int. J. Inf. Sec.*, 14(1):47–60, February 2015. doi:10.1007/s10207-014-0239-8.

6 Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.*, 14(3):349–382, 2012. doi:10.1007/s10009-011-0196-8.

7 Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Systems & Control Letters*, 123:2–41, 2016. doi:10.1016/j.scico.2016.02.008.

8 Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods Syst. Des.*, 38(3):223–262, 2011. doi:10.1007/s10703-011-0114-4.

9 P. W. L. Fong. Access control by tracking shallow execution history. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 43–55, 2004. doi:10.1109/SECPRI.2004.1301314.

10 Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005. doi:10.1007/s10207-004-0046-8.

11 Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), January 2009. doi:10.1145/1455526.1455532.

12 Oded Maler, Dejan Nickovic, and Amir Pnueli. From mitl to timed automata. In *Proceedings of the 4th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'06, pages 274–289, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11867340_20.

13 Dejan Nickovic and Oded Maler. Amt: A property-based monitoring tool for analog systems. In Jean-François Raskin and P. S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems*, pages 304–319, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

14 Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, and Hervé Marchand. Tipex: A tool chain for timed property enforcement during execution. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 306–320, Cham, 2015. Springer International Publishing.

15 Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena-Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014. doi:10.1007/s10703-014-0215-y.

16 Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard von Hanxleden. Runtime enforcement of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s):178:1–178:25, 2017. doi:10.1145/3126500.

17 Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, and Hervé Marchand. Enforcement of (timed) properties with uncontrollable events. In *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, pages 542–560, 2015. doi:10.1007/978-3-319-25150-9_31.

18 Matthieu Renard, Antoine Rollet, and Yliès Falcone. Runtime enforcement of timed properties using games. *Formal Aspects of Computing*, 32(2):315–360, 2020. URL: `https://link.springer.com/article/10.1007/s00165-020-00515-2`.

19 G. Roc su. On safety properties and their monitoring. *Scientific Annals of Computer Science*, 22(2):327–365, 2012. `doi:10.7561/SACS.2012.2.327`.

20 U. Sammapun, Insup Lee, and O. Sokolsky. Rt-mac: runtime monitoring and checking of quantitative and probabilistic properties. In *11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05)*, pages 147–153, August 2005. `doi:10.1109/RTCSA.2005.84`.

21 Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000. `doi:10.1145/353323.353382`.

22 Saumya Shankar and Srinivas Pinisetty. Serial compositional runtime enforcement of safety timed properties. In *Proceedings of the 16th Innovations in Software Engineering Conference*, ISEC '23, New York, NY, USA, 2023. Association for Computing Machinery. `doi:10.1145/3578527.3578529`.

23 Saumya Shankar, Antoine Rollet, Srinivas Pinisetty, and Yliès Falcone. Bounded-memory runtime enforcement. In Owolabi Legunsen and Grigore Rosu, editors, *Model Checking Software*, pages 114–133, Cham, 2022. Springer International Publishing.

24 Chamseddine Talhi, Nadia Tawbi, and Mourad Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2):158–184, 2008. Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA '06). `doi:10.1016/j.ic.2007.07.009`.

A Appendix: Additional constraints

A.1 Additional constraints on an enforcer with unbounded-memory

We here provide some additional constraints to provide some guarantees on the output sequence produced by the enforcer in terms of delay.

- **Optimality** (minimum delay)

$$\begin{array}{l}\forall \sigma \in tw(\Sigma) : E^{\varphi}(\sigma) = \epsilon \vee \exists m, w \in tw(\Sigma) : E^{\varphi}(\sigma) = m \cdot w (\models \varphi) \text{ with} \\ m = max^{\varphi}_{\prec,\epsilon}(E^{\varphi}(\sigma)) \text{ and} \\ w = min_{\preceq,lex,end}\{w' \in m^{-1} \cdot \varphi \mid \Pi_{\Sigma}(w') = \Pi_{\Sigma}(m^{-1} \cdot E^{\varphi}(\sigma)) \\ \qquad\qquad \wedge \ m \cdot w' \lhd_d \sigma \ \wedge \ start(w') \geq end(\sigma)\}\end{array} \quad \textbf{(Opt)}$$

For any input σ, if the output $E^{\varphi}(\sigma)$ is not empty, then it can be separated into: the maximal strict prefix m of $E^{\varphi}(\sigma)$ satisfying property φ, and a suffix w. The optimality constraint expresses that, among those sequences w' that could have been chosen, w is the minimal one in terms of ending date, and lexical order. The "sequences that could have been chosen" are those such that $m \cdot w'$ satisfies the property, have the same events, are delayed subsequences of the input σ, and have a starting date greater than or equal to $end(\sigma)$.

▶ **Proposition 16** (Optimality). *Given some property $\varphi \subseteq tw(\Sigma)$, its enforcer E^{φ} as per Definition 5 satisfies* **Opt** *property.*

A.2 Additional constraints on an enforcer with bounded-memory

Similar to Sect. A.1, we here provide some additional constraints to provide some guarantees on the output sequence produced by the enforcer in terms of length and delay.

Let us first look at the below definition of ∞-compatibility used to define one of the optimality property: Consider the case when the buffer is full. Thus, as we have discussed, the enforcer would clean the buffer in such a way that, the treatment of the future events by the enforcer is not affected, implying that after cleaning, the behaviour of the bounded enforcer should be the same as that of the unbounded enforcer (bounded enforcer before/without cleaning). Thus, we define the notion of ∞-compatible as follows:

▶ **Definition 17** (∞-compatible). *Enforcer $E^{\varphi,k}$ is compatible with $E^{\varphi,\infty}$ (enforcer with buffer size $k = \infty$, i.e., unbounded enforcer), noted ∞-compatible$(E^{\varphi,k})$, if* $\forall \sigma \in tw(\Sigma) : E^{\varphi,\infty}(\sigma) \cdot \text{buff}(E^{\varphi,\infty}(\sigma)) \sim_\varphi E^{\varphi,k}_{\text{out}}(\sigma) \cdot \text{buff}(E^{\varphi,k}(\sigma))$.

The above definition says that, for φ, a bounded enforcer $E^{\varphi,k}$ is compatible with an unbounded enforcer $E^{\varphi,\infty}$, if for any input word σ, the concatenation of the output and the buffer content of $E^{\varphi,\infty}$, is φ-equivalent to the concatenation of the output and the buffer content of $E^{\varphi,k}$. Finally the optimality properties are:

- **Optimality** (output is of maximum length):
  Consider any bounded enforcer $F^{\varphi,k}$ as per Def. 10. We have
  $\exists \sigma \in tw(\Sigma), \forall (t,a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$:

$$(E^{\varphi,k}_{\text{out}}(\sigma) \cdot \text{buff}(E^{\varphi,k}(\sigma) = F^{\varphi,k}_{\text{out}}(\sigma) \cdot \text{buff}(F^{\varphi,k}(\sigma)) \quad \wedge |E^{\varphi,k}_{\text{out}}(\sigma \cdot (t,a)) \cdot \text{buff}(E^{\varphi,k}(\sigma \cdot (t,a)))| \\ < |F^{\varphi,k}_{\text{out}}(\sigma \cdot (t,a)) \cdot \text{buff}(F^{\varphi,k}(\sigma \cdot (t,a))|) \implies \neg(\infty\text{-compatible}(F^{\varphi,k})) \tag{Opt1B}$$

  Opt1B expresses that an enforcer $E^{\varphi,k}$ (as per Def. 11) is optimal; if, for some input, for any other enforcer $F^{\varphi,k}$, the length of the concatenation of its output and its buffer content is greater than the length of the concatenation of output and the buffer content of $E^{\varphi,k}$, then the output produced by $F^{\varphi,k}$ is not ∞-compatible. Simply, it implies that, there does not exist an enforcer that can clean the buffer in a better way; by discarding less events and being ∞-compatible with the unbounded enforcer.
- **Optimality** (minimum delay): The optimality property ensuring that each subsequence is output as soon as possible, with minimum delay for bounded-memory case is similar to **Opt** in Sect. A.1 for the unbounded-memory case, although with some notational variations.

$$\forall \sigma \in tw(\Sigma) : E^{\varphi,k}_{\text{out}}(\sigma) = \epsilon \vee \exists m, w \in tw(\Sigma) : E^{\varphi,k}_{\text{out}}(\sigma) = m \cdot w (\models \varphi) \text{ with} \\ m = max^{\varphi}_{\preceq,\epsilon}(E^{\varphi,k}(\sigma)) \text{ and} \\ w = min_{\preceq,lex,end}\{w' \in m^{-1} \cdot \varphi | \Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E^{\varphi,k}(\sigma)) \\ \wedge\, m \cdot w' \lhd_d \sigma \quad \wedge \quad start(w') \geq end(\sigma)\} \tag{Opt2B}$$

▶ **Proposition 18** (Optimality). *Given some property φ and the maximum buffer size k, its enforcer $E^{\varphi,k}$ as per Definition 11 satisfies* **Opt1B** *and* **Opt2B** *properties.*

B Appendix: Enforcement algorithm and performance evaluation

In Sect. 5.2, we provided an abstract view of our bounded-memory enforcer, defining it as a function that transforms words. In this section, we provide the overall enforcement algorithm. Also, we implemented the algorithm, to validate the feasibility of enforcement monitoring through experiments and analyse the performance of the enforcer.

B.1 Enforcement algorithm

Let $\mathcal{A}_\varphi = (L, l_0, X, \Sigma, \Delta, F)$ define φ. Consider that it has one clock[11]. The Algorithm 1, takes $\mathcal{A}_\varphi$ and the buffer size $k \in \mathbb{N}$ as input parameters.

In the algorithm, σ_s and σ_c refer to the output and the buffered events as in Def. 11. *currState* holds the current state. Function `await_event` is used to wait for a new input event. Function `check_reachability` computes all the reachable paths from the current state *currState* upon events in $\sigma_c \cdot (\delta, a)$. Function `get_acc_paths` takes as input all the paths returned by function `check_reachability` and returns only those that lead to a location in F. Function `get_od` computes the optimal delays for $\sigma_c \cdot (\delta, a)$. Also, it returns the state reached upon $\sigma_c \cdot (\delta, a)$. Function `append` is used to add the given event to the given word. Function `release` releases the given word as output. Function `check_reach_acc` takes all the paths returned by the function `check_reachability`, finds the last state of each paths and checks if an accepting location is reachable from that last state of the considered path. Function `len` is used to get the length of the word. Function `Get_SW` behaves as the same as function Get_SW$^{\varphi,k}$ in Def. 11. It returns property equivalent subwords (of length $< k$) of delayed $\sigma_c \cdot (\delta, a)$. Function `clean` optimally deletes events from the given word, similar[12] to the function Clean$^{\varphi,k}$ in Def. 11. Function `sum_d` returns the sum of the delays of each of the events of the given word.

The algorithm proceeds as follows: Initially, buffers σ_c and σ_s are empty and *currState* is initialized with the initial state of $\mathcal{A}_\varphi$ (i.e., $[l_0, 0]$). Variable c holds the sum of the delays of the events deleted by function `clean`[13]. It then enters into an infinite loop waiting for an input event. Upon receiving an event (δ, a) (as in line no. 5 of Algorithm 1), where the delay δ is relative to the delay of the previous received event, the enforcer adds extra delay c if returned by function `clean`, and progresses its clock (as shown in line 8). It then computes all the reachable paths from the current state *currState* upon events in $\sigma_c \cdot (\delta, a)$ using function `check_reachability` (as shown in line 9). Function `get_acc_paths` then returns only those paths that lead to a state in F i.e., it returns all accepting paths, (as shown in line 10). If an accepting path exists (i.e., $accPaths \neq \emptyset$ as in line no. 11), then the enforcer computes optimal delays for $\sigma_c \cdot (\delta, a)$, (in σ_{ca}), appends each event from σ_{ca} to σ_s to be released as output and sets the current state accordingly (as in line nos. 12-17). Otherwise, if an accepting path does not exist, it finds out if there is still a chance to reach an accepting location in the future using function `check_reach_acc`. If an accepting location is not reachable in the future, the enforcer drops/suppresses the received event and continues enforcement. Otherwise, if an accepting location is reachable, it appends the received event to the buffer, if the buffer is not full (as in line no. 24). Else, if the buffer is full, it optimally cleans the buffer to accomodate the received event, given that there exists any property equivalent subwords (of length $< k$) delaying $\sigma_c \cdot (\delta, a)$; otherwise the enforcer halts.

Function `clean` first computes the optimal delayed word σ_d of σ_{ca} using function `get_od`. It then enters a loop where in every iteration i ($1 \leq i \leq k+1$), it checks if a subword of length i from index j of σ_d can be read on a cycle s.t., the state remains same with or without

[11] This initial approach can be generalised and extended to any number of clocks.

[12] Function `clean` in this algorithm also returns the sum of the delays of the suppressed events in addition to the cleaned timed word; however Function Clean$^{\varphi,k}$ of Def. 11 does not, since the enforcement mechanism considers the delays to be absloute.

[13] When function `clean` delete the very first or the intermediate events then c returned by it is 0 (because function `clean` internally has taken care of those delays by adding them to the delay of the next event in buffer, since the delays are relative; thus it need not be added to the incoming event (δ, a), thus $c = 0$), otherwise $c \neq 0$ (then, c is added to the delay of the incoming event).

Algorithm 1 Algorithm Enforcer $(\mathcal{A}_\varphi, k)$.

```
1: σ_c, σ_s = [ ]
2: currState ← [l_0, 0]
3: c = 0
4: while true do
5:     (δ, a) ← await_event()
6:     if c ≠ 0 then
7:         δ = δ + c
8:     currState[1] = currState[1] + δ
9:     allPaths = check_reachability(currState, σ_c · (δ, a))
10:    accPaths = get_acc_paths(allPaths)
11:    if accPaths ≠ ∅ then
12:        (σ_ca, state) = get_od(currState, σ_c · (δ, a))
13:        for event ∈ σ_ca do
14:            append(σ_s, σ_ca[event])
15:        σ_c = [ ]
16:        currState = state
17:        release(σ_s)
18:    else
19:        isReachable = check_reach_acc(allPaths)
20:        if isReachable == False then
21:            continue
22:        else
23:            if len(σ_c) < k then
24:                append(σ_c, (δ, a))
25:            else
26:                if Get_SW(σ_s, σ_ca) ≠ ∅ then
27:                    (σ_c, c) = clean(currState, σ_c · (δ, a))
28:                else
29:                    exit
```

```
1: function CLEAN(curr_state, σ_ca)
2:     (σ_d, state_d) = get_od(curr_state, σ_ca)
3:     for i ∈ 1 ··· k + 1 do
4:         j = 1
5:         while i + j ≤ k + 2 do
6:             if j == 1 then
7:                 (DelayedW1, state1) = get_od(curr_state, σ_d[j···j+i−1])
8:                 if curr_state == state1 then
9:                     return σ_d[j+i···k+1], 0
10:            else
11:                (DelayedW1, state1) = get_od(curr_state, (σ_d[1···j−1] · σ_d[j···j+i−1])
12:                (DelayedW2, state2) = get_od(curr_state, σ_d[1···j−1])
13:                if state1 == state2 then
14:                    return σ_d[1···j−1] · σ_d[j+i···k+1], sum_d(σ_d[j···j+i−1])
15:        j + +
```

the subword. If yes, the subword is removed from σ_d and the resultant subword is returned by function `clean` along with the sum of delays of all the events of the subwords. The time complexity of function `clean` in Algorithm 1 is $\mathcal{O}(k^3)$ with k as the buffer size.

B.2 Implementation and performance evaluation

We implemented Algorithm 1 and developed an experimentation framework in order to i.) validate the feasibility of enforcement monitoring and ii.) analyse the performance of the enforcer through experiments.

Our experimental framework. There are tools for obtaining EMs, from the properties expressed using formalisms such as TA. We use the tool called TiPEX [14], as this is the RE monitor generation tool based on the theory of RE of timed properties proposed in [7], which we consider in this work.

We update the EMTA module to account for the bounded case. We add the definition of function clean as proposed in Algorithm 1 which will clean the buffer when required. In the implementation functions checkReachability, getAccPaths, getOptimalDelays, and clean straightaway maps to the functions `check_reachability`, `get_acc_paths`, `get_od`,

and `clean` in Algorithm 1. Function `check_reach_acc` in the algorithm is implemented to check if an action needs to be suppressed. It is implemented as follows: First it calls function checkReachability to give all the paths from the current state upon the given word; then, it finds the last state of each paths and checks if an accepting location is reachable from that last state of the considered path. It returns true in that case. Function clean is implemented in approx 50 LoC and Tipex is of approx 1200 LoC.

Table 3 Effect on T(s) by bounded-memory enforcer by varying |Input|.

Input	T	clean	
		No.	T_c
100	1.513	0	0
200	5.843	2	0.0254
300	13.294	27	0.3564
400	23.306	52	0.8372
500	35.485	77	1.4245
600	51.615	102	2.1624
700	67.942	127	2.94386
800	89.137	152	3.88968
900	120.853	177	5.27106
1000	148.1558	202	6.62964

Performance evaluation and analysis. For measuring the performance, the timed property P in the prototype example 2 is used. The length of input sequence was varied from 100 to 1000 with an increment of 100 each time. To determine the worst case time taken, the input sequences were set so that function clean is invoked frequently by the bounded-memory enforcer. Considering buffer size[14] $k = 50$, we have the following observations from Tab. 3, where `Input` denotes the length of input sequences, `T` indicates the time taken by the bounded-memory enforcer to output the word, `No.` and T_c under `clean` indicates the number of times function `clean` is called and the total time taken by it respectively: i.) the time taken by the bounded-memory enforcer increases non-linearly[15] with the linear increase in trace length. ii.) the average time taken by the function `clean` (per call) is 0.019 s and is low/reasonable.

▶ Remark 19. Upon calculating the processing time for both the unbounded enforcer and the bounded-memory enforcer with the same set of input sequences of identical length, it is observed that the unbounded enforcer takes longer time. This is due to the extra workload of managing a buffer that can expand with incoming input sequences in the unbounded enforcer, as opposed to the bounded enforcer where maintaining a fixed-size buffer incurs less overhead.

C Appendix: Enforceable properties

When we construct an enforcer for any given property $\varphi \subseteq tw(\sigma)$ using the proposed approach, we ideally want the enforcer to never halt (i.e., the mode of the enforcer should never change to stop). While the fourth case $(k^{pref(\varphi)}(\sigma_s, \sigma_{ca}) \neq \emptyset \wedge |\sigma_{ca}| > k \wedge \text{Get\_SW}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \emptyset)$

[14] Buffer allocation should be done carefully. If we use a small buffer, there will be a higher chance of buffer overflow, resulting in the deletion of more events. However, if we use a sufficiently sized buffer, there will be less overflow, which is crucial for retaining critical events and avoiding their loss.

[15] The behaviour is non-linear because of reasons such as, overhead in maintaining the list of uncorrected events (events in σ_c), the more numbers of times function clean is called, etc.

of Def. 11 in Sect. 5.2 may result in the enforcer halting, which is undesirable in practical scenarios, our objective is to ensure that the enforcer consistently operates. To achieve this, we propose (based on our preliminary observations) specific syntactic and semantic conditions on the TA, s.t. if the TA corresponding to any given property $\varphi \subseteq tw(\sigma)$ satisfies these conditions, it guarantees that the enforcer never halts (i.e., the fourth case of Def. 11 never arise; or the function Get_SW$^{\varphi,k}$ always finds a word (of length $< k$), which is property equivalent with the buffer contents).

▶ **Definition 20** (Conditions on TA). *We define the following conditions on TA $\mathcal{A}_\varphi$:*
$\forall \rho \in Run_{F_G}(\mathcal{A}_\varphi)$ *and timed words* $\sigma_s, \sigma, \sigma_{con}$, $\exists \rho_0, \rho_1, \rho_2 : \rho = \rho_0 \cdot \rho_1 \cdot \rho_2$,
$(\rho_0 = (l_0, v_0) \xrightarrow{\sigma_s} (l, v)) \wedge \rho_1 = (l, v) \xrightarrow{\sigma} (l, v')) \wedge (\rho_2 = (l, v') \xrightarrow{\sigma_{con}} (l_n, v_n))$, $\forall t_r \in \rho_1$:

1. $t_r = (l, g, a, \emptyset, l')$ *where,* $g \in \mathcal{G}(X)$ *with* $\bowtie \in \{<, \leq\})$, *or*
2. $t_r = (l, \epsilon, a, \emptyset, l')$

The above definition gives (not necessary but sufficient) syntactic and semantic conditions for a TA. The condition expresses that, for any run accepted by $\mathcal{A}_\varphi$, if it can be broken down (as shown in figure 2) into three consecutive runs ρ_0, ρ_1, and ρ_2 as follows:

ρ_0: from initial state (l_0, v_0) upon σ_s, $\mathcal{A}_\varphi$ makes the last transition to state (l, v),
ρ_1: from (l, v) upon σ, $\mathcal{A}_\varphi$ makes last transition to state (l, v') with no change in location,
ρ_2: from (l, v') upon σ_{con}, $\mathcal{A}_\varphi$ makes the last transition to state (l_n, v_n) (an accepting state),

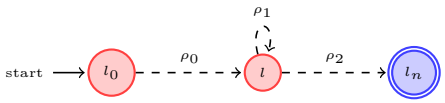

Figure 2 Runs of a TA, showing only the locations reached.

then all the transitions of the run ρ_1 should be s.t., i) the guards are Boolean combinations of simple constraints where the operators are restricted to $\{\leq, <\}$, or, ii) the transitions should not have guards and resets acting on them.

▶ **Example 21.** (TA satisfying conditions of Def. 20). TA in Figure 1 of Example 2 satisfies the syntactic conditions of Def. 20, since the accepting run from initial state (l_0, v_0) can be broken down into runs ρ_0 (from (l_0, v_0) to (l_1, v_1)), ρ_1 (from (l_1, v_1) to (l_1, v_1')), and ρ_2 (from (l_1, v_1') to (l_2, v_2), where (l_2, v_2) is an accepting state) and the transition in ρ_1 (e.g., $t_r = (l_1, \{x \leq 4\}, r, \epsilon, l_1)$) has guard with constraints where the operator is $\leq$ (i.e., $x \leq 4$)

▶ Remark 22. (Condition for enforceability.) If $k \geq n$ (where $n \in \mathbb{N}$ is the number of locations in $\mathcal{A}_\varphi$), and the TA satisfies the conditions given in Definition 20, then $E^{\varphi,k}$ defined in Def. 11 never halts.