Runtime Enforcement of Regular Timed Properties

Srinivas Pinisetty  
INRIA Rennes - Bretagne Atlantique, France  
Srinivas.Pinisetty@inria.fr

Yliès Falcone  
Laboratoire d’Informatique de Grenoble  
Université Grenoble I, France  
Ylies.Falcone@ujf-grenoble.fr

Thierry Jéron  
INRIA Rennes - Bretagne Atlantique, France  
Thierry.Jeron@inria.fr

Hervé Marchand  
INRIA Rennes - Bretagne Atlantique, France  
Herve.Marchand@inria.fr

ABSTRACT

Runtime enforcement is a verification/validation technique aiming at correcting (possibly incorrect) executions of a system of interest. In this paper, we consider enforcement monitoring for systems with timed specifications (modeled as timed automata). We consider runtime enforcement of any regular timed property specified by a timed automaton. To ease their design and their correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input-output behavior, constraints that should be satisfied by such functions, enforcement monitors that implement an enforcement function as a transition system, and enforcement algorithms that describe the implementation of enforcement monitors. The feasibility of enforcement monitoring for timed properties is validated by prototyping the synthesis of enforcement monitors.

1. INTRODUCTION

Runtime enforcement [10, 6, 7, 5, 9] is a verification/validation technique aiming at correcting (possibly incorrect) executions of a system of interest. In traditional (untimed) approaches, the enforcement mechanism is a monitor that is modeled as a decision procedure that inputs, corrects, and outputs a sequence of events. How a monitor transforms the input sequence is done according to a high-level specification, formalized as a property, that indicates correct and incorrect sequences. Moreover, a monitor should only output correct sequences (the monitor is sound) and should minimally alter the input sequence (the monitor is transparent). An important endeavor in the previous enforcement monitoring approaches is to determine the set of enforceable properties which, from an abstract point of view, consists in determining the set of properties for which an enforcement monitor can be synthesized. Several sets of enforceable properties were delineated according to the enforcement primitives conferred to enforcement monitors, e.g., safety properties for security automata [10, 6], renewal properties for edit-automata [7], response properties for generalized enforcement monitors [5]. In addition to enforcement primitives, enforceability limitations arose because properties were expressed over infinite sequences, thus imposing an enforcement monitor to consistently take decisions (on finite sequences) with the possible infinite continuations of the observed sequences. Only considering properties over finite sequences allows to get rid of the aforementioned limitations [4]: any property over finite sequences is enforceable with a monitor endowed with the primitives of an edit-automaton, basically inserting and suppressing events from the input sequence. Most of the runtime enforcement endeavors consider logical time, as opposed to physical time. Suppressing events and (later) inserting them is assumed without consequence on the execution nor on the satisfiability of the property.

In this paper, we consider runtime enforcement of timed properties, initially introduced in [9]. In timed properties (over finite sequences), the time that elapses between two events matters and affects the satisfiability of the property. Moreover, it turns out that considering time when specifying the behavior of systems brings some expressiveness that can be particularly useful in some application domains when, for instance, specifying usage of resources. For example, the two following properties could specify the behavior of a server.

P1 “Resource grants and releases alternate, starting with a grant, and every grant should be released within 15 to 20 time units (t.u.)”.

P2 “Every 10 t.u., there should be a request for a resource followed by a grant. The request should occur within 5 t.u.”.

In this paper, we generalize and extend the initial approach to runtime enforcement of timed properties [9] in several directions. (See Section 7 for a detailed description of the improvements over [9].) First the approach in [9] tackled only safety and co-safety properties that allow to respectively express that “something bad should never happen” and that “something good should happen within a finite amount of time”; thus ruling out specifications P1 and P2. Indeed, in the space of regular properties (over timed words), many interesting properties of systems are neither safety nor co-safety properties but belongs to a larger set: the set of the so-called response properties that allow to specify some form of transactional behavior and cannot be considered in [9]. We propose to synthesize enforcement mechanisms for all regular timed properties. We consider enforcement mechanisms as time retardants, i.e., mechanisms that keep the same order of actions, but are allowed to increase the delay between two actions. We specify the mechanisms at several levels. The notion of enforcement function describes the behavior of an enforcement mechanism at an abstract level as an input-output relation between timed words. An enforcement monitor implements an enforcement function and describes the behavior of an enforcement mechanism in an operational way as a rule-based transition system. Enforcement algorithms describe the implemen-
tation of enforcement monitors and serve to guide the concrete implementation of enforcement mechanisms. The difficulty that arises when considering response properties is that the aforementioned enforcement mechanisms should consider input (corrected) sequences of events that alternate between satisfying and not satisfying the underlying property.

**Paper organization.**

Some motivating examples for the enforcement monitoring of response properties and illustrating the enforceability issue are given in Section 2. Section 3 introduces some preliminaries and notations. Section 4 explains how properties are specified as TA with some examples. Section 6 describes the enforcement mechanisms (enforcement function, monitor, and algorithm) in detail. Related work are discussed in Section 7. Finally, conclusions and open perspectives are drawn in Section 8.

## 2. SOME MOTIVATING EXAMPLES

Let us consider again the two properties P1 and P2 mentioned in the introduction. We shall see in Section 4 how to formalize these properties as properties represented by timed automata. Let gr denote the introduction. We shall see in Section 4 how to formalize these properties as TA with some examples. Section 6 describes the enforcement mechanisms (enforcement function, monitor, and algorithm) in detail. Related work are discussed in Section 7. Finally, conclusions and open perspectives are drawn in Section 8.

### 3. PRELIMINARIES AND NOTATION

#### 3.1 Untimed Languages

A (finite) word over an alphabet \( A \) is a finite sequence of elements of \( A \). The length of a word \( w \) is noted \( |w| \). The empty word over \( A \) is denoted by \( \epsilon \) when clear from the context. The set of all (respectively non-empty) words over \( A \) is denoted by \( A^* \) (respectively \( A^+ \)). A language over \( A \) is a set \( L \subseteq A^* \). The concatenation of two words \( w \) and \( w' \) is noted \( w \cdot w' \). For an interval \([j,k]\) of \( \mathbb{N} \), by \( \bigcup_{i\in[j,k]}(a) \) we denote the concatenation \( a_j \cdot a_{j+1} \cdot \ldots \cdot a_k \). A word \( w' \) is a prefix of a word \( w \), noted \( w' \preceq w \), whenever there exists a word \( w'' \) such that \( w = w' \cdot w'' \), and \( w' < w \) whenever \( w' \preceq w \) and \( |w'| < |w| \). For a word \( w \) and \( 1 \leq i \leq |w| \), the \( i \)-th letter (resp. prefix of length \( i \), suffix starting at position \( i \)) of \( w \) is noted \( w(i) \) (respectively \( w[|w|−i],w[|w|−i−1],\ldots,w[|w|−1] \) with the convention \( w[0] = w \)). A word \( w \) is a suffix of \( w' \) if \( w' \preceq w \) and \( |w'| < |w| \).

#### 3.2 Timed Words and Languages

A timed word over a finite alphabet \( \Sigma \) is a finite sequence \( \sigma = (\delta_1,a_1) \cdot (\delta_2,a_2) \cdot \ldots \cdot (\delta_n,a_n) \) of events. For an event \((\delta_1,a_1)\), \(a \cdot \delta, \delta, a \) is the action and \( \text{delay}(\delta_1,a_1) \) is the delay between \( a_{i−1} \) (or the initialization for \( i=1 \)) and \( a_i \). A timed language is any set \( L \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^* \) that satisfies the constraints defined in the untimed case (related to length, concatenation, prefix, etc) naturally extend to timed words. The **untimed projection** of \( \sigma \) is \( \Pi_\Sigma(\sigma) \) defined as \( a_1 \cdot a_2 \ldots a_n \) in \( \Sigma^* \) (i.e., delays are ignored). The **duration** of a timed word \( \sigma \), noted \( \text{time}(\sigma) = \sum_{i=1}^n \delta_i \), is the sum of its delays.

#### 3.3 Preliminaries to Runtime Enforcement

**Orders on timed words.**

Apart from the prefix order \( \preceq \), we define the following partial orders on timed words:

- **Delaying order** \( \succeq_{\text{d}} \); For \( \sigma, \sigma' \in (\mathbb{R}_{\geq 0} \times \Sigma)^* \), we say that \( \sigma' \) is a **delayed prefix of** \( \sigma \) (noted \( \sigma' \succeq_{\text{d}} \sigma \)) iff \( \Pi_\Sigma(\sigma') \preceq \Pi_\Sigma(\sigma) \) and \( \forall i \leq |\sigma'| : \text{delay}(\sigma'(i)) \geq \text{delay}(\sigma(i)) \), which means that the untimed projection of \( \sigma' \) is a prefix of the untimed projection of \( \sigma \), but delays in \( \sigma' \) may be greater than the delays in \( \sigma \). This order will be used to characterize outputs with respect to inputs in enforcement monitoring.

- **Lexicographical order** \( \preceq_{\text{lex}} \); Given any two timed words \( \sigma, \sigma' \) s.t. \( \Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma') \) and any two timed events with identical actions \((\delta, a)\) and \((\delta', a)\), \( \sigma \preceq_{\text{lex}} \sigma' \) is defined recursively as follows: \( \epsilon \preceq_{\text{lex}} \epsilon \), and \( (\delta, a) \cdot \sigma \preceq_{\text{lex}} (\delta', a) \cdot \sigma \) iff \( \delta \leq \delta' \) or \( \delta = \delta' \) and \( \sigma \preceq_{\text{lex}} \sigma' \). This order is useful to choose a unique timed word among some with same actions.

Some special sets and sequences.

We consider an input timed word \( \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^* \) and a timed property \( \varphi \) defined by a timed regular language.
The observation of $\sigma$ at time $t \in \mathbb{R}_{\geq 0}$ is the part of the input that can be read by the enforcement monitor at time $t$, and is defined as the maximal prefix of $\sigma$ whose duration is smaller than $t$: \[\text{obs}(\sigma, t) \overset{\text{def}}{=} \max_{\tau} \{\sigma' \in (\mathbb{R}_0 \times \Sigma)^* | \sigma' \preceq \sigma \land \text{time}(\sigma') \leq t\},\] where $\max_{\tau}$ takes the maximal sequence according to the prefix relation $\preceq$ (unique in this case).

The maximal strict prefix of $\sigma$ that belongs to $\varphi$ is noted $\max^\lt_{\varphi}(\sigma)$ and defined as: $\max^\lt_{\varphi}(\sigma) \overset{\text{def}}{=} \max_{\tau} \{\sigma' \in (\mathbb{R}_0 \times \Sigma)^* | \sigma' \prec \sigma \land \sigma' \in \varphi \cup \{\varnothing\}\}.$

### 4. PROPERTIES AS TIMED AUTOMATA

#### 4.1 Timed Automata

A timed automaton [1] (TA) is an automaton extended with a finite set of real valued clocks. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of clocks. A clock valuation for $X$ is a function $\nu$ from $X$ to $\mathbb{R}_0^X$, where $\mathbb{R}_0^X$ denotes the valuations of clocks in $X$ in the set $\mathbb{R}_0$. For $\nu \in \mathbb{R}_0^X$ and $\delta \in \mathbb{R}_0^X$, $\nu + \delta$ is the valuation assigning $\nu(x_i) + \delta$ to each clock $x_i$ of $X$. Given a set of clocks $X', \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation $\nu$ where all clocks in $X'$ are assigned to 0. $G(X)$ denotes the set of clock constraints defined as Boolean combinations of simple constraints of the form $x_i \in c$ with $x_i \in X$, $c \in \mathbb{N}$ and $\in \in \{\leq, <, =, \geq, >\}$. Given $g \in G(X)$ and $\nu \in \mathbb{R}_0^X$, we write $\nu \models g$ when $g$ holds according to $\nu$.

**Definition 1 (Timed automaton)** A timed automaton is a tuple $A = (L, l_0, X, \Sigma, \Delta, F)$, s.t. $L$ is a finite set of locations, $l_0 \in L$ is the initial location, $X$ is a finite set of clocks, $\Sigma$ is a finite set of events, $\Delta \subseteq L \times G(X) \times \Sigma \times 2^X \times L$ is the transition relation. $F \subseteq L$ is a set of accepting locations.

The semantics of a TA is a timed transition system $[A] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ where $Q = L \times \mathbb{R}^X_0$ is the (infinite) set of states, $q_0 = (l_0, 0\nu)$ is the initial state where $0\nu$ is the valuation that maps every clock in $X$ to 0, $Q_F = L \times \mathbb{R}^X_0$ is the set of accepting states, $\Gamma = \mathbb{R}_0^X \times \Sigma$ is the set of transition labels, i.e., pairs composed of a delay and an action. The transition relation $\rightarrow \subseteq Q \times \Gamma \times Q$ is the maximal set of transitions of the form $(l, \nu) \overset{(\delta, a)}{\rightarrow} (l', \nu')$ with $\nu' = (\nu + \delta)[Y' \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ s.t. $\nu + \delta \models g$ for $\nu \in \mathbb{R}^X_0$.

In the following, we consider a timed automaton $A = (L, l_0, X, \Sigma, \Delta, F)$ with its semantics $[A]$. A is deterministic whenever for any $(l, g_1, a_1, Y_1, t_1)$ and $(l, g_2, a_2, Y_2, t_2)$ in $\Delta$, $g_1 \land g_2$ is unsatisfiable. A is complete whenever for any location $l \in L$ and every event $a \in \Sigma$, the disjunction of the guards of the transitions leaving $l$ and labeled by $a$ evaluates to true (i.e., it holds according to any valuation).

In the remainder of this paper, we shall consider only deterministic timed automata, and, automata refer to timed automata.

A run $\rho$ from $q \in Q$ is a sequence of moves in $[A]$ of the form: $\rho = q \overset{(\delta_1, a_1)}{\rightarrow} q_1 \overset{(\delta_2, a_2)}{\rightarrow} \cdots \overset{(\delta_n, a_n)}{\rightarrow} q_n$, for some $n \in \mathbb{N}$. The set of runs accepted by $A$, i.e., when $q_n \in Q_F$, is denoted as $\mathcal{L}_{Q_F}(A)$. The set of runs from $q_0 \in Q$ is denoted $\text{Run}(A)$ and $\text{Run}_{Q_F}(A)$ denotes the subset of runs accepted by $A$, i.e., when $q_n \in Q_F$. The trace of a run $\rho$ is the timed word $(\delta_1, a_1) \cdot (\delta_2, a_2) \cdot \cdots (\delta_n, a_n)$. We note $\mathcal{L}(A)$ the set of traces of $\text{Run}(A)$. We extend this notation to $\mathcal{L}_{Q_F}(A)$ in a natural way, which are the set of accepted traces.

#### 4.2 Classes of Timed Properties

In this paper, a timed property is defined by a timed language $\varphi \subseteq (\mathbb{R}_0 \times \Sigma)^*$ that can be recognized by a timed automaton.

Theorem 1 (Properties as TA’s) Figure 1 presents some properties expressed as TA’s, where accepting locations are represented by squares. The safety property “The delay between consecutive allocation requests should be at least 10 t.u.” is specified by the safety TA in Figure 1a. The co-safety property “An allocation request should be followed by a grant within 5 to 7 t.u.” is specified by the co-safety TA in Figure 1b. The response property $P_1$ is specified by the response TA in Figure 1c. The response property $P_2$ is specified by the response TA in Figure 1d.

### 5. CONSTRAINTS ON AN ENFORCEMENT MECHANISM

At an abstract level, an enforcement mechanism can be seen as a function with two parameters, an input timed word, and a given time, and which returns a timed word as output.
An enforcement function $E_\varphi$ transforms some input timed word $\sigma$ which is possibly incorrect w.r.t. $\varphi$. The resulting output $E_\varphi(\sigma, t)$ at time $t$ is a timed word with same actions, but possibly increased delays between actions so that eventually it will satisfy the property. Our enforcement monitors are time retardants.

Definition 3 (Constraints on an Enforcement Mechanism) For a timed property $\varphi$, an enforcement mechanism behaves as a function $E_\varphi$ from $\mathbb{R}_{\geq 0} \times \Sigma^* \times \mathbb{R}_{\geq 0}$ to $\mathbb{R}_{\geq 0} \times \Sigma^*$, satisfying the following constraints:

- **Time retardant:**
  $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t, t' \in \mathbb{R}_{\geq 0} :$
  \[ t \leq t' \implies E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t') \quad \text{(Phy1)}. \]

- **Soundness:**
  $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0} :$
  \[ E_\varphi(\sigma, t) \neq \varepsilon \implies (\exists t' \geq t : E_\varphi(\sigma, t') \models \varphi) \quad \text{(Snd)}. \]

- **Transparency:**
  $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0} :$
  \[ E_\varphi(\sigma, t) \preceq_d \text{obs}(\sigma, t) \quad \text{(Tr)}. \]

The requirements of the enforcement function are specified by three constraints, *physically time retardant*, *soundness* and *transparency*. **(Phy1)** means that the outputs of the enforcement function are concatenated over time, i.e., what is output cannot be modified. **(Phy2)** expresses that the events cannot be released as output, before they are read as input. **Soundness (Snd)** means that if a timed word is released as output by the enforcement function, in the future, the output of the enforcement function should satisfy the property $\varphi$. In other words, no event is output before being sure that the property will be satisfied by subsequent events. **Transparency (Tr)** expresses that, at any time $t$, the output is a delayed prefix of the observed input $\text{obs}(\sigma, t)$.

6. GENERALIZED RUNTIME ENFORCEMENT OF TIMED PROPERTIES

In this section, we present how to synthesize an enforcement monitor from any property specified by a timed automaton. Rather than giving directly the enforcement algorithm, we propose several descriptions of enforcement mechanisms: as an enforcement function, as a transition system, and finally as an algorithm derived from the transition system.

6.1 Functional Definition

The purpose of the functional definition of an enforcement mechanism is to clearly distinguish each step such as i) processing the input, ii) computing the delayed timed word satisfying the property, iii) and processing the output sequence. Moreover, the enforcement function describes how these functions are composed to transform an input sequence. The resulting functional definition presented here satisfies the physical, soundness, and transparency constraints.

Definition 4 (Enforcement function) The enforcement function for a property $\varphi$ is $E_\varphi : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ defined as:

\[ E_\varphi(\sigma, t) = \text{obs} \left( \Pi_1(\text{store}_\varphi(\text{obs}(\sigma, t))), t \right). \]

where $\text{store}_\varphi : (\mathbb{R}_{\geq 0} \times \Sigma)^* \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ is defined as

\[ \text{store}_\varphi(\varepsilon) = (\varepsilon, \varepsilon) \]

\[ \text{store}_\varphi(\sigma \cdot (\delta, a)) = \begin{cases} (\sigma_s \cdot \min_{\delta \leq \sigma_s} K, \varepsilon) & \text{if } K \neq \emptyset \\ (\sigma_s, a \cdot (\delta, a)) & \text{otherwise} \end{cases} \]

with

\[ (\sigma_s, a) = \text{store}_\varphi(\delta) \]

\[ K = \kappa_{\varphi}(\text{time}(\sigma) + \delta, \sigma_s, \sigma_c \cdot (\delta, a)) \]

where $\kappa_{\varphi}(T, \sigma_s, \sigma_c) \overset{\text{def}}{=} \{ w \in (\mathbb{R}_{\geq 0} \times \Sigma)^* | w \preceq_d \sigma_c \land |w| = |\sigma_c| \land \sigma_s \cdot w \models \varphi \land \text{delay}(w(1)) \geq T - \text{time}(\sigma_s) \}.$

In the definition of $E_\varphi$, $\text{obs}(\sigma, t)$ is the prefix of the input that has been observed at time $t$, and thus can be processed by the enforcement function. From this observation, the $\text{store}_\varphi$ function computes a pair, whose first component extracted by $\Pi_1$ is processed by obs to produce the output.

The $\text{store}_\varphi$ function takes as input an observation, and outputs a pair: the first element is the transformation of a prefix of the observation for which delays have been computed (the property is satisfiable by this prefix by appropriate delaying); the second element is the suffix of the observation for which delays still have to be computed. The $\text{store}_\varphi$ function is defined inductively: initially, for an empty observation, both elements are empty; if $\sigma$ has been observed, $\text{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$, and a new event $(\delta, a)$ is observed, there are two possible cases, according to the vacuity of the set $K = \kappa_{\varphi}(\text{time}(\sigma) + \delta, \sigma_s, \sigma_c \cdot (\delta, a))$ (the set of candidate timed words appropriately delaying $\sigma_c \cdot (\delta, a)$ and satisfying $\varphi$, see below):

- if $K \neq \emptyset$, the minimal timed word in $K$ w.r.t the lexicographic order is appended to $\sigma_s$, and the second element is set to $\epsilon$.
- otherwise, $(\delta, a)$ is appended to $\sigma_c$ and $\sigma_s$ is unmodified.

The function $\kappa_{\varphi}$ has three parameters: $T$ (the duration of the current observation), $\sigma_s$, and $\sigma_c$. It computes the set of candidate timed words $w$ “appropriately delaying” $\sigma_c$ such that $\sigma_s \cdot w$ satisfies $\varphi$. The appropriate delaying is such that $w$ and $\sigma_c$ have identical actions, same length but delays of $w$ are greater than or equal to those of $\sigma_c$. Moreover the delay of the first action in $w$ should exceed the difference between the duration of the observation and the duration of $\sigma_s$. The reason for this constraint is that $\sigma_s$ will be output entirely after a duration of $\text{time}(\sigma_s)$, while the decision to output $w$ is taken after $T$ t.u., thus a smaller value for $\text{delay}(w(1))$ would cause this delay to be elapsed before the decision is taken.

Notice that upon reading an input event $(\delta, a)$, in case there are no appropriate delays, and if it is not possible to correct the subsequence $\sigma_c \cdot (\delta, a)$ (when $\kappa_{\varphi}$ is empty), then the input event $(\delta, a)$ is appended to $\sigma_c$, and in case the subsequence $\sigma_c \cdot (\delta, a)$ can be corrected, it is appended immediately to $\sigma_s$ (with appropriate delays) without relying on events which will be read later. The policy adopted is to correct the observation of the input sequence as soon as possible. Consequently, the input sequence is treated as a series of subsequences, each subsequence allowing to satisfy the property.

Proposition 1 Given some property $\varphi$, its enforcement function $E_\varphi$ as per Definition 4 satisfies the physical constraints Phy1 and Phy2 and is sound and transparent as per Definition 3.

In addition to the physical constraints, soundness and transparency, the functional definition also ensures that each subsequence is released as output as soon as possible, as expressed by the following proposition:
Proposition 2 (Optimality of an enforcement function) Given some property $\varphi$, its enforcement function $E_\varphi$ as per Definition 4 satisfies the following optimality constraint:

$$\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0} : E_\varphi(\sigma, t) \neq \epsilon \land E_\varphi(\sigma, t) \models \varphi \implies \exists m, w \in (\mathbb{R}_{\geq 0} \times \Sigma)^* : m = \max_{\sigma} E_\varphi(\sigma, t)
\land E_\varphi(\sigma, t) = m \cdot w
\land \text{time}(w) = \min \{\text{time}(w') | \text{delay}(w'(1)) \geq \text{time}(\sigma[1::m \cdot \text{delay}(\sigma)]) - \text{time}(m) \land m \cdot w' \models \varphi\}.$$ 

For any input $\sigma$, at any time $t$, if the output $E_\varphi(\sigma, t)$ is not $\epsilon$, and satisfies $\varphi$, then the output is considered as two sub-sequences $m$ followed by $w$, such that $m$ is the maximal strict prefix of $E_\varphi(\sigma, t)$, satisfying the property $\varphi$ and $w$ is the remaining sub-sequence such that $E_\varphi(\sigma, t) = m \cdot w$.

The last sub-sequence of the output which again makes the output to satisfy $\varphi$ after $m$ is $w$. The optimality constraint expresses that the sum of the delays (time required to output) of $w$ is minimal. The delay for the events in $w$ should be chosen such that $m \cdot w$ satisfies the property, is transparent, and the delay of the first event is greater than the difference between the duration of the input sequence $\sigma[1::m \cdot \text{delay}(\sigma)]$ and the duration of $m$.

Notice that if time($\sigma[1::m \cdot \text{delay}(\sigma)]$) − time($m$) is negative or null, then this means that the delay corresponding to some events in the sequence preceding $w$ (which is $m$) are increased, providing sufficient amount of time to observe the last subsequence (which is $\sigma[|\sigma|+1::|\sigma|+|w|]$) entirely. In case time($\sigma[1::m \cdot \text{delay}(\sigma)]$) − time($m$) is positive, all events in $m$ have been released as output before the last subsequence $\sigma[|\sigma|+1::|\sigma|+|w|]$ is observed entirely as input. After releasing $m$, time($\sigma[1::m \cdot \text{delay}(\sigma)]$) − time($m$) time units have elapsed and thus the last subsequence $w$ can be released as output only after a delay of time($\sigma[1::m \cdot \text{delay}(\sigma)]$) − time($m$) time units.

Example 2 (Enforcement function) We now illustrate how the Definition 4 is applied to enforce the property P1 defined by the automaton depicted in Fig. 1c with $\Sigma = \{gr, rel\}$, and the input timed word $\sigma = (3, gr) \cdot (10, rel) \cdot (3, gr) \cdot (5, gr)$. Figure 2 shows the evolution of obs, store$_\varphi$, and $E_\varphi$. The variable $t$ describes global time. The resulting output is $(13, gr) \cdot (15, rel)$, which satisfies the property P1.

Example 3 (Enforcement function: A non-enforceable property) Let us now consider another property P2, formalized by the TA in Figure 1d, with $\Sigma = \{gr, req\}$, and the input timed word $\sigma = (3, req) \cdot (4, gr) \cdot (2, req) \cdot (6, gr)$. Figure 3 shows the evolution of obs, store$_\varphi$, and $E_\varphi$. The variable $t$ describes global time. The resulting output of the enforcement function is $\epsilon$ at any time instance.

Remark 1 (Non-enforceable properties) From Example 3, notice that the output of the enforcement function is $\epsilon$, though the input sequence itself satisfies the property. As explained in Section 2, the delay associated with the first event should be at least 7 time units, but increasing the delay associated with the first event to 7, will falsify the guard on transition between $l_0$ to $l_1$, which is a possible move upon the first event $req$, and there is no transition with a reset of clock $x$ before.

It can be noticed that guards with $\sigma, \leq, \leq$ for some properties, some input sequences that can be delayed to satisfy $\varphi$ cannot be corrected by enforcement, because the delay of the first event of each subsequence may be increased, which may falsify a guard with $\sigma, \leq, =$. However, even for such properties, the enforcement function will never produce incorrect output. If the enforcement function outputs some sequence, it will satisfy the physical, soundness and transparency constraints.

6.2 Enforcement Monitor

An enforcement function $E_\varphi$ for a property $\varphi$ specified by a TA $A_\varphi$, is implemented by an enforcement monitor (EM), defined as a transition system $E$. An EM is equipped with a memory and a set of enforcement operations used to store and dump some timed events to and from the memory, respectively. The memory of an EM is basically a queue containing a timed word, the received actions (with increased delays) that are not released yet. In addition, an EM also keeps track of the state of the underlying TA, and clock values used to count time between input events and between output events.

Before presenting the definition of enforcement monitor, we introduce $update_\varphi$ as a function from $Q \times (\mathbb{R}_{\geq 0} \times \Sigma)^+ \times \mathbb{R}_{\geq 0}$ to $(\mathbb{R}_{\geq 0} \times \Sigma)^+ \times \mathbb{B}$. The update function takes as input the (current) state $(q \in Q)$ of $A_\varphi$, a timed word $\sigma^t \in (\mathbb{R}_{\geq 0} \times \Sigma)^+$, and a real number $m$, which is the difference between the duration of the input sequence observed minus the duration of the corrected sequence, and returns a timed word of length $|\sigma^t|$ and a Boolean as

\begin{align*}
    obs(\sigma, t) &= \epsilon \\
    store_\varphi(obs(\sigma, t)) &= (\epsilon, \epsilon) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t) \\
    obs(\sigma, t) &= (3, gr) \\
    store_\varphi(obs(\sigma, t)) &= (\epsilon, (3, gr)) \\
    E_\varphi(\sigma, t) &= obs((3, gr), t) \\
    obs(\sigma, t) &= (3, gr) \cdot (10, rel) \\
    store_\varphi(obs(\sigma, t)) &= ((13, gr) \cdot (15, rel), \epsilon) \\
    E_\varphi(\sigma, t) &= obs((13, gr) \cdot (15, rel), \epsilon) \\
    obs(\sigma, t) &= (3, gr) \cdot (10, rel) \cdot (3, gr) \cdot (5, gr) \\
    store_\varphi(obs(\sigma, t)) &= ((13, gr) \cdot (15, rel) \cdot (3, gr) \cdot (5, gr)) \\
    E_\varphi(\sigma, t) &= obs((13, gr) \cdot (15, rel) \cdot (3, gr) \cdot (5, gr)) \\
    obs(\sigma, t) &= (3, gr) \cdot (10, rel) \cdot (3, gr) \cdot (5, gr) \\
    store_\varphi(obs(\sigma, t)) &= ((13, gr) \cdot (15, rel) \cdot (3, gr) \cdot (5, gr)) \\
    E_\varphi(\sigma, t) &= obs((13, gr) \cdot (15, rel) \cdot (3, gr) \cdot (5, gr)) \\
    obs(\sigma, t) &= (13, gr) \cdot (15, rel) \\
    store_\varphi(obs(\sigma, t)) &= ((13, gr) \cdot (15, rel)) \\
    E_\varphi(\sigma, t) &= obs((13, gr) \cdot (15, rel)) \\
    t \in [0, 3] \\
    obs(\sigma, t) &= \epsilon \\
    store_\varphi(obs(\sigma, t)) &= (\epsilon, \epsilon) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t) \\
    t \in [3, 7] \\
    obs(\sigma, t) &= (3, gr) \\
    store_\varphi(obs(\sigma, t)) &= (\epsilon, (3, gr)) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t) \\
    t \in [7, 9] \\
    obs(\sigma, t) &= (3, gr) \cdot (4, gr) \\
    store_\varphi(obs(\sigma, t)) &= ((3, gr) \cdot (4, gr)) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t) \\
    t \in [9, 15] \\
    obs(\sigma, t) &= (3, gr) \cdot (4, gr) \cdot (2, req) \\
    store_\varphi(obs(\sigma, t)) &= ((3, gr) \cdot (4, gr) \cdot (2, req)) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t) \\
    t \in [15, \infty] \\
    obs(\sigma, t) &= (3, gr) \cdot (4, gr) \cdot (2, req) \cdot (6, gr) \\
    store_\varphi(obs(\sigma, t)) &= ((3, gr) \cdot (4, gr) \cdot (2, req) \cdot (6, gr)) \\
    E_\varphi(\sigma, t) &= obs(\epsilon, t)
\end{align*}
output.

\[ \text{update}(q, \sigma_c, m_t) \overset{\text{def}}{=} \begin{cases} (\sigma_c, \text{ff}) & \text{if } \Delta(\sigma_c, m_t, q) = \emptyset \\ (\sigma_c', \text{tt}) & \text{otherwise} \end{cases} \]

where \( \sigma_c' = \min_{\Delta \sigma_c} \Delta(\sigma_c, m_t, q) \) with \( \Delta \) defined as:

\[
\Delta(\sigma_c, m_t, q) = \bigcup_{t \in [1, m_c]} \bigcup_{i \in [-\infty, m_c]} \bigcup_{\delta_i \geq \text{delay}(\sigma(i))} \bigcup_{\forall i \in Q: q_i = \text{ff}} \bigcup_{\sigma_c(i) = \text{store}(\sigma, \delta_i, i)} \bigcup_{\sigma_c(i) = \text{act}(\sigma, \delta_i)} \bigcup_{\sigma_c(i) = \text{dump}(\sigma, \delta_i)}
\]

\( \Delta \) is the set of timed words \( \sigma_c' \) of length \( |\sigma_c| \) with same actions as \( \sigma_c \), each delay in the sequence is equal to or greater than the delay at the corresponding index in the provided input sequence \( \sigma_c \), and the first delay should be greater that or equal to \( m_t \) and an accepting state is reachable from state \( q \) upon the sequence \( \sigma_c \).

- In the first case, there are no good delays such that an accepting state is reachable from the state \( q \) upon a sequence delaying \( \sigma_c \) (\( \Delta = \emptyset \)). In this case, the update function returns the same timed word \( \sigma_c \) (which is provided as input), and a Boolean value \( \text{ff} \), indicating that no accepting state is reachable.

- The second case applies when there are good delays and an accepting state in \( q_i \in Q_F \) is reachable from \( q \) upon a sequence delayed from \( \sigma_c \). In this case, the update function returns a timed word of minimal duration belonging to \( \Delta \), chosen according to the lexicographic order; and a Boolean value \( \text{tt} \), indicating that an accepting state is reachable.

**Definition 5 (Enforcement Monitor)** An enforcement monitor \( \mathcal{E} \) for \( \varphi \) is a transition system \( (C, C_0, \Gamma, \xmapsto{} ) \text{s.t.:} \)

- \( C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times Q \) is the set of configurations,
- \( C_0 = (\emptyset, \varepsilon, 0, 0, 0, q_0) \in C \) is the initial configuration,
- \( \Gamma_C = ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\varepsilon\}) \times Op \times ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\varepsilon\}) \) is the alphabet, which are triples comprised of an optional input event, an operation, and an optional output event, where the set of possible operations is \( Op = \{\text{store}(\cdot, \cdot, \cdot, \cdot), \text{store}(\cdot, \cdot), \text{dump}(\cdot, \cdot), \text{idle}(\cdot)\}; \)
- \( C \subseteq C \times C \times C \) is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:

  1. **store-\( \varphi \):**

     \[(\sigma_c, \sigma_c', \delta, d, m_t, q) \overset{\text{store-}(\delta, a)}{\xmapsto{}} (\sigma_c, \sigma_c', (\delta, a), 0, d, m_t', q) \quad \text{if } \Delta(\sigma_c, m_t, q) = \emptyset \]

     \[m_t' = m_t + \delta \]

  2. **store-\( \varphi \):**

     \[(\sigma_c, \sigma_c', \delta, d, m_t, q) \overset{\text{store-}(\delta, a)}{\xmapsto{}} (\sigma_c, \sigma_c', (\delta, a), 0, d, m_t', q) \quad \text{if } \Delta(\sigma_c, m_t, q) = \emptyset \]

     \[m_t' = m_t + \delta \]

  3. **dump:**

     \[(\delta, a) \sigma_c, \sigma_c, s, \delta, 0, m_t, q) \overset{\text{dump}(\delta, a)}{\xmapsto{}} (\sigma_c, \sigma_c, s, 0, m_t, q) \]

  4. **idle:**

     \[(\sigma_c, \sigma_c, s, d, m_t, q) \overset{\text{idle}(\delta)}{\xmapsto{}} (\sigma_c, \sigma_c, s + \delta, d + \delta, m_t, q) \]

A configuration \( (\sigma_c, \sigma_c', \sigma_a, s, d, m_t, q) \) of the EM consists of the current stored sequence (i.e., the memory content) \( \sigma_a \) and \( \sigma_c \). The sequence which is corrected and can be released as output is denoted by \( \sigma_c' \). The sequence \( \sigma_c \) is sort of an internal memory for the store function: this is the input sequence read by the EM, but yet to be corrected. The configuration also contains two clock values \( s \) and \( d \) indicating respectively the time elapsed since the last store and dump operations, and one more counter \( m_t \), indicating the difference between the duration of the observed input sequence and the duration of the corrected sequence. \( q \) is the current state of \( [\mathcal{A}] \) reached after processing the sequence already released followed by the timed word in memory \( \sigma_c \).

Semantic rules can be understood as follows:

- Upon reception of an event \( (\delta, a) \), one of the following store rules is executed.

  - The **store-\( \varphi \)** rule is executed if the update function returns \( \text{ff} \) (indicating that \( \sigma_c \cdot (\delta, a) \) cannot be corrected). The clock \( s \) is reset to 0, and the event \( (\delta, a) \) is appended to the internal memory \( \sigma_c \). The delay corresponding to the input event \( \delta \) is added to \( m_t \).

  - The **store-\( \varphi \)** rule is executed if the update function returns \( \text{tt} \), indicating that \( \varphi \) can be satisfied for the sequence already released as output, followed by the sequence in \( \sigma_c \), followed by \( \sigma_c \cdot (\delta, a) \) with possibly increased delays. When executing this rule, \( s \) is reset to 0, and the timed word \( \sigma_c' \) returned by the update function is appended to the content of the output memory \( \sigma_a \). The delay of the input event \( \delta \) is added to \( m_t \), and the duration of the corrected sub-sequence returned by the update function, \( \text{time}(\sigma_c') \), is subtracted from \( m_t \).

  - The **dump** rule is executed if the time elapsed since the last dump operation \( d \) is equal to the delay corresponding to the first event of the timed word \( \sigma_a \) in the memory. The event \( (\delta, a) \) is released as output and removed from \( \sigma_c \) and the clock \( d \) is reset to 0.

  - The **idle** rule adds the time elapsed \( \delta \) to the current values of \( s \) and \( d \) when no store nor dump operation is possible.

**Example 4 (Execution of an enforcement monitor)** We now illustrate how the rules of Definition 5 are applied to enforce the property \( P_1 \). Let us consider the input timed word \( \sigma = (3, gr) \cdot (10, rel) \cdot (3, gr) \cdot (5, gr) \). Figure 4 shows how semantic rules are applied, and the evolution of the configurations of the enforcement monitor. In a configuration, the input (resp. output) is on the right (resp. left). The variable \( t \) describes global time. The resulting output is \((13, gr) \cdot (15, rel)\), which satisfies the property \( P_1 \). From \( t = 28 \), only the delay rule can be applied.

**Remark 2 (Simplified definitions of enforcement monitor)** To synthesize an EM for a safety or co-safety property, one can use simplified definitions. For example, for a safety property, only one timed word is needed in the configuration. Indeed, recall that \( \sigma_a \) is a sort of internal memory used to store the input events used when it may be possible to reach an accepting state if more events are observed in the future. Since a safety property is prefix-closed, upon an event that cannot be delayed to keep satisfying the property, no future extension can. Hence, \( \sigma_a \) is not necessary for safety properties. Therefore, some simplifications, that may lead to performance improvements, are possible.

**Remark 3 (Relation between enforcement function and monitor)** An enforcement monitor is an operational description of an enforcement function. For any property \( \varphi \), input \( \sigma \), and at any time \( t \), the input sequence processed by the enforcement function \( obs(\sigma, t) \) is the concatenation of all events read by EM (store) over various steps, until time \( t \). At any time \( t \), the output of the enforcement function \( E_{\varphi} \) is equal to the output behavior of the associated EM (the concatenation of all the released events (dump) until time \( t \)). Since, \( E_{\varphi} \) is sound, transparent, and satisfies the physical constraints,
with relating the input-output behavior of an EM to $E_{\varphi}$, it can be proved that EM is sound, transparent and satisfies the physical constraints.

6.3 Implementation

The implementation of an EM consists of two processes running concurrently (Store and Dump) as shown in Figure 5, and a memory. The Store process models the store rules. The memory contains the timed word $\sigma_s$: the corrected sequence that can be released as output. The memory $\sigma_s$ is realized as a queue, shared by the Store and Dump processes, where the Store process adds events which are processed and corrected to this queue. The Dump process reads events stored in the memory $\sigma_s$ and releases them as output after the required amount of time. The Store process also makes use of another internal buffer $\sigma_c$ (not shared with any other process), to store the events which are read, but cannot be corrected (to satisfy the property). In the algorithm the event primitive is used to wait for a trigger event from another process or to wait until some condition becomes true. The wait primitive is used by a process to wait for a certain amount of time, which is determined by the process itself.

Figure 4: Execution of an enforcement monitor

Algorithm 1 StoreProcess

\[
(l, \nu) \leftarrow (l_0, [X \leftarrow 0]) \\
(\sigma_s, \sigma_c) \leftarrow (\epsilon, \epsilon) \\
m_t \leftarrow 0 \\
\text{while } \tau \tau \text{ do} \\
\quad (\delta, a) \leftarrow \text{await (event)} \\
\quad \sigma_c \leftarrow \sigma_c \cdot (\delta, a) \\
\quad m_t \leftarrow m_t + \delta \\
\quad (\sigma'_c, \text{isPath}) \leftarrow \text{update}(l, \nu, \sigma_c, m_t) \\
\quad \text{if isPath = } \tau \tau \text{ then} \\
\quad \quad m_t \leftarrow m_t - \text{time}(\sigma'_c) \\
\quad \quad \sigma_s \leftarrow \sigma_s \cdot \sigma'_c \\
\quad \quad (l, \nu) \leftarrow \text{post}(l, \nu, \sigma_c) \\
\quad \quad \sigma_c \leftarrow \epsilon \\
\text{end if} \\
\text{end while}
\]

The algorithm proceeds as follows. The StoreProcess initially waits for an input event. After receiving an event as input, it is appended to the internal buffer $\sigma_c$, with the corresponding delay, and the delay corresponding to the received event $\delta$ is added to $m_t$. Then the update function is invoked providing the events stored in $\sigma_c$ as input. If the update function indicates that there is a path leading to an accepting state, (i.e., if isPath = $\tau \tau$), then the timed word $\sigma'_c$ returned by the update function, is appended to the shared memory $\sigma_c$ (since it now corrected with respect to the property, and can be released as output). Then, the duration of $\sigma'_c$ is subtracted from $m_t$. Before proceeding to the next iteration, the state of the automaton $(l, \nu)$ is updated, and the internal memory $\sigma_c$ is cleared.

Algorithm 2 DumpProcess

\[
d \leftarrow 0 \\
\text{while } \tau \tau \text{ do} \\
\quad \text{await } (\sigma_s \neq \epsilon) \\
\quad (\delta, a) \leftarrow \text{dequeue}(\sigma_s) \\
\quad \text{wait}(\delta - d) \\
\quad \text{dump}(a) \\
\quad d \leftarrow 0 \\
\text{end while}
\]

The DumpProcess algorithm (see Algorithm 2) is an infinite loop that scrutinizes the memory and proceeds as follows: Initially, the clock $d$ is set to 0. If the memory is empty ($\sigma_s = \epsilon$), the dump process waits until a new element $(\delta, a)$ is stored in the memory. Else (the memory is not empty), it proceeds with the first element in the memory. Using the dequeue operation, the first element stored in the memory is removed, and is stored as $(\delta, a)$. Meanwhile, $d$ keeps track of the time elapsed since the last dump operation. The DumpProcess waits for $(\delta - d)$ time units before performing the dump(a) operation, releasing the action a as output (which amounts to appending $(\delta, a)$ to the output of the enforcement monitor). Finally, the clock $d$ is reset to 0 before the next iteration starts.

7. RELATED WORK

Several approaches for the runtime enforcement of properties are related to the one proposed in this paper.
Most of the work on this topic was dedicated to the enforcement of untimed properties (see [3] for a short overview). Schneider introduced security automata as the first runtime mechanism for enforcing safety properties [10]. Then the set of enforceable properties was later refined by Schneider, Hamlen, and Morrisett by showing that security automata were actually restrained by the computational limits exhibited by Viswanathan and Kim [11]: the set of co-recursively enumerable safety properties is a strict upper limit of the power of (execution) enforcement monitors defined as security automata. Ligatti et al. [7] later introduced edit-automata as enforcement monitors. Edit-automata can either insert a new action by replacing the current input, or suppress it. The set of properties enforced by edit-automata is called the set of infinite renewal properties: it is a super-set of safety properties and contains some liveness properties (but not all). Similar to edit-automata are generic enforcement monitors proposed that are able to enforce the set of response properties in the safety-progress classification of (untimed) properties. Moreover, some variants of edit-automata differ in how they ensure the transparency constraints (see e.g., [2]). However, none of these approaches is able to synthesize an enforcement mechanism from a property.

In previous work in [9] we introduced the problem of runtime enforcement for timed properties. We similarly proposed several notions of enforcement mechanisms: enforcement function, enforcement monitor, and enforcement algorithm. Only safety and co-safety properties were considered and different definitions of mechanisms were proposed for each of the classes. In this paper, given any timed automaton (representing any regular property), we synthesize enforcement functions, monitors and algorithms according to one general definition and for more properties: the set of regular properties. Moreover, when the input timed automaton recognizes a safety or a co-safety property, we could optimize the data-structures used by the enforcement monitors and algorithms.

Finally, for the enforcement of co-safety properties, the approach in [9] assumes that time elapses differently for input and output sequences (the sequences are desynchronized). More precisely, the delay of the first event of the output sequence is computed from the moment an enforcement mechanism detects that its input sequence can be corrected (that is, the mechanisms has read a sequence that can be delayed into a correct sequence). Our approach is more realistic as it does not suffer from this "shift" problem.

Finally, Matteucci proposed to synthesize controller operations to enforce safety and information-flow properties using process-algebra [8]. The approach targets discrete-time properties and systems are modelled as timed processes expressed in CCS. Monitors are described at an abstract level and resemble Schneider's security automata: monitors only halt the underlying program and do not consider timing issues.

8. CONCLUSION AND FUTURE WORK

Conclusion.
This paper presented a general enforcement monitoring framework for systems with timing requirements. We generalized the results of [9], showed how to synthesize enforcement mechanisms for any regular timed property (modeled with a timed automaton). Enforcement mechanisms are described at several levels of abstraction (enforcement function, monitor, and algorithm), thus facilitating the design and implementation of such mechanisms.

Future work.
Several avenues for future work are open by this paper.

First, we believe it is important to study and delineate the set of enforceable timed properties. As shown informally by this paper, some timed properties should be characterized as non-enforceable. For this purpose, an enforceability condition should be defined and used to delineate enforceable properties. Such a criterion should ideally also be expressible on timed automata. Note that even for properties which are non-enforceable, enforcement monitors can be built, which may not be able to correct some input sequences, but outputs are always sound.

Specifications are currently modeled with timed automata. We consider synthesizing enforcement mechanisms from more expressive formalisms. For instance, we will consider formalisms such as context-free timed languages (which can be useful for recursive specifications) or introduce data into requirements (which can be useful in some application domains). Implementing efficient enforcement monitors is another important aspect and should be done in a particular application domain.

Alternative enforcement primitives can be afforded to timed retardant. For instance, we could relax the constraint of only augmenting delays of events. For instance, time retardants that delay the current observation of the input (while being allowed to shorten the delay of some events) have yet to be studied. We believe that suppressing events also can be considered, by erasing some events which are stored in the memory.

A prototype is implemented based on the algorithms described in Section 6.3. Regarding efficiency, some experiments should be done using the prototype to determine to what extent using more specific enforcement mechanisms for a sub-class (e.g., the safety class), can give better performance, compared to using the general enforcement monitoring mechanism.

9. REFERENCES