Runtime Enforcement of Regular Timed Properties by Suppressing and Delaying Events

Yliès Falcone, Thierry Jéron, Hervé Marchand, Srinivas Pinisetty

Runtime Enforcement of Regular Timed Properties by Suppressing and Delaying Events

Yliès Falcone[a,*], Thierry Jéron[b], Hervé Marchand[b], Srinivas Pinisetty[c]

[a] *Univ. Grenoble Alpes, Inria, LIG, F-38000 Grenoble, France*
[b] *INRIA Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex*
[c] *Aalto University, Finland*

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 where the physical time elapsing between actions matters. Executions are thus modelled as timed words (i.e., sequences of actions with dates). We consider runtime enforcement for timed specifications modelled as timed automata. Our enforcement mechanisms have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus possibly allowing for longer executions. To ease their design and their correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input-output behaviour in terms of transformations of timed words, constraints that should be satisfied by such functions, enforcement monitors that describe the operational behaviour of enforcement functions, 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 from timed automata.

Keywords: verification, monitoring, runtime enforcement, timed specifications.

1. Introduction

Runtime enforcement [1, 2, 3, 4, 5] is a verification and validation technique aiming at correcting possibly-incorrect executions of a system of interest. In traditional (untimed) approaches, the enforcement mechanism is a *monitor* modelled as a transducer that inputs, corrects, and outputs a sequence of events. How a monitor transforms the input sequence is done according to a specification of correct sequences, formalised as a property. Moreover, a monitor should satisfy some requirements: it should be *sound* in the sense that only (prefixes of) correct sequences are output; it should also be *transparent* meaning that the output sequence preserves some relation with the input sequence, depending on the authorised operations.

Runtime enforcement monitors can be used in various application domains. For instance, enforcement monitors can be used for the design of firewalls, to verify the control-flow integrity and memory access of low-level code [6], or implemented in security kernels or virtual machines to protect the access to sensitive system resources (e.g., [7]). In [8], we discuss some other uses of enforcement monitors such as resource allocation and the implementation of robust mail servers.

*Corresponding author
Email addresses: ylies.falcone@imag.fr (Yliès Falcone), thierry.jeron@inria.fr (Thierry Jéron), herve.marchand@inria.fr (Hervé Marchand), srinivas.pinisetty@aalto.fi (Srinivas Pinisetty)

In this paper, we consider *runtime enforcement of timed properties*, initially introduced in [5, 9]. In timed properties (over finite sequences), not only the order of events matters, but also their occurrence dates affect the satisfaction of the property. It turns out that considering time constraints when specifying the behaviour of systems brings some expressiveness that can be particularly useful in some application domains when, for instance, specifying the usage of resources. In Section 2, we present some running and motivating examples of timed specifications related to the access of resources by processes. We shall see that, in contrast to the untimed case, the amount of time an event is stored influences the satisfaction of properties.

In [5], we propose preliminary enforcement mechanisms restricted to safety and co-safety timed properties. Safety and co-safety properties allow to express that "something bad should never happen" and that "something good should happen within a finite amount of time", respectively. In [9], we generalise and extend the initial approach of [5] to the whole class of timed regular properties. Indeed, some regular properties may express interesting behaviors of systems belonging to a larger class that allows to specify some form of transactional behaviour. Regular properties are, in general, neither prefix nor extension closed, meaning that the evaluation of an input sequence w.r.t. the property also depends on its possible future continuations. For instance, an incorrect input sequence alone may not be correctable by an enforcement mechanism, but the reception of some events in the future may allow some correction. Hence, the difficulty that arises is that the enforcement mechanism should take conservative decisions and change its behaviour over time taking into account the evaluation (w.r.t. the property) of the current input sequence and its possible continuations. Roughly speaking, in [5, 9], enforcement mechanisms receive sequences of events composed of actions and delays between them, and can only increase those delays to satisfy the desired timed property; while in this paper, we consider absolute dates and allow to reduce delays between events (as described in detail in the following paragraph).

Contributions. In this paper, we extend [9] in several directions. The main extension consists in increasing the power of enforcement mechanisms by allowing them to suppress input events, when the monitor determines that it is not possible to correct the input sequence, whatever is its continuation. Consequently, enforcement mechanisms can continue operating, and outputting events, while in our previous approaches the output would have been blocked forever. This feature and other considerations also drove us to revisit and simplify the formalisation of enforcement mechanisms. We now consider events composed of actions and absolute dates, and enforcement mechanisms are *time retardant with suppression* in the following sense: monitors should keep the same order of the actions that are not suppressed, and are allowed to increase the absolute dates of actions in order to satisfy timing constraints. Note, this allows to decrease delays between actions, while it is not allowed in [5, 9]. As in [5, 9], we specify the mechanisms at several levels, but in a revised and simplified manner: the notion of enforcement function describes the behaviour of an enforcement mechanism at an abstract level as an input-output relation between timed words; requested properties of these functions are formalised as soundness, transparency, optimality, and additional physical constraints[1]; we design adequate enforcement functions and prove that they satisfy those properties; the operational behaviour of enforcement functions is described as enforcement monitors, and it is proved that those monitors correctly implement the enforcement functions; finally enforcement algorithms describe the implementation of enforcement monitors and serve to guide the concrete implementation of enforcement mechanisms. Interestingly, although all untimed regular properties over finite sequences can be enforced [10], some enforcement limitations arise for timed properties (over finite sequences). Indeed, we show that storing events in the timed setting influences the output of enforcement mechanisms. In particular, because of physical

[1] The two latter constraints are specific to runtime enforcement of timed properties.

time, an enforcement mechanism might not be able to output certain correct input sequences. Finally, we propose an implementation of the enforcement mechanisms for all regular properties specified by one-clock timed automata (while [5, 9] feature an implementation for safety and co-safety properties only).

Paper organisation. The rest of this paper is organised as follows. In Section 2, we introduce some motivating and running examples for the enforcement monitoring of timed properties, and illustrate the behaviour of enforcement mechanisms and the enforceability issues that arise. Section 3 introduces some preliminaries and notations. Section 4 recalls timed automata. Section 5 introduces our enforcement monitoring framework and specifies the constraints that should be satisfied by enforcement mechanisms. Section 6 defines enforcement functions as functional descriptions of enforcement mechanisms. Section 7 defines enforcement monitors as operational description of enforcement mechanisms in the form of transition systems. Section 8 proposes algorithms that effectively implement enforcement monitors. In Section 9, we present an implementation of enforcement mechanism in Python and evaluate the performance of synthesised enforcement mechanisms. In Section 10, we discuss related work. In Section 11, we draw conclusions and open perspectives. Finally, to ease the reading of this article, some proofs are sketched and their complete versions can be found in Appendix A.

2. General principles and motivating examples

In this section, we describe the general principles of enforcement monitoring of timed properties, and illustrate the expected input/output behavior of enforcement mechanisms on several examples.

2.1. General principles of enforcement monitoring in a timed context

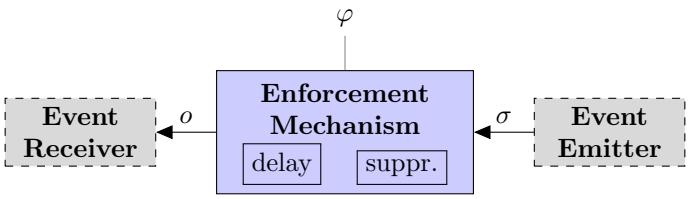


Figure 1: Illustration of the principle of enforcement monitoring.

As illustrated in Figure 1, the purpose of enforcement monitoring is to read some (possibly incorrect) input sequence of events σ produced by a system, referred to as the event emitter, to transform it into an output sequence of events o that is correct w.r.t. a specification formalised by a property φ. This output sequence is then transmitted to an event receiver. In our timed setting, events are actions with their occurrence dates. Input and output sequences of events are then formalised by timed words and enforcement mechanisms can be seen as transformers of timed words.

Figure 2 illustrates the behavior of an enforcement mechanism when correcting an input sequence. The dashed and solid curves respectively represent input and output sequences of events (occurrence dates in abscissa and actions in ordinate). The behavior of an enforcement mechanism should satisfy some constraints, namely *physical constraint*, *soundness*, and *transparency*. Intuitively, the physical constraint states that an enforcement mechanism cannot modify what it has already output, i.e., the output forms a continuously-growing sequence of events; soundness states that the output sequence should be correct w.r.t. the property (note, soundness is not represented in the figure, since this would require to represent an area containing only the sequences admitted by the property); transparency states

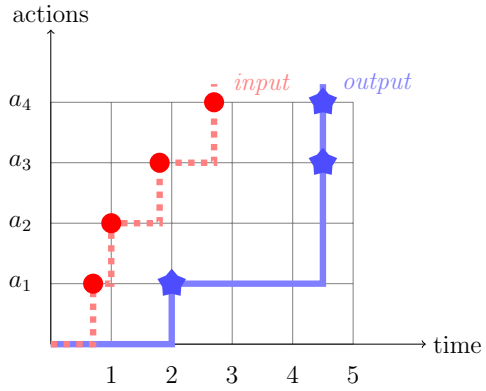


Figure 2: Behavior of an enforcement mechanism.

that the output sequence is obtained by delaying or suppressing actions from the input sequence (and not changing the order of actions); thus, if the events of the input curve are not suppressed, they appear later in the output curve, in the same order. For example, actions a_1, a_3 and a_4 are delayed but a_2 is suppressed. Notice that by delaying dates of events the enforcement mechanism allows to reduce delays between events. For example, action a_4 occurs strictly after action a_3, but both actions are released at the same date. Moreover, the actions should be released as output as soon as possible, which will be described by an *optimality property*.

2.2. Motivating examples

We introduce some running and motivating examples related to the usage of resources by some processes. We also provide some intuition on the expected behavior of our enforcement mechanisms, and point out some issues arising in the timed context. We discuss further these issues and their relation to the expected constraints on enforcement mechanisms.

Let us consider the situation where two processes access to and operate on a common resource. Each process i (with $i \in \{1, 2\}$) has three interactions with the resource: acquisition (acq_i), release (rel_i), and a specific operation (op_i). Both processes can also execute a common action op. System initialisation is denoted by action $init$. In the following, variable t keeps track of global time. Figures 3, 4, and 5 illustrate the behavior of enforcement mechanisms for several specifications on the behavior of the processes and for particular input sequences.[2]

Specification S_1. The specification states that "*Each process should acquire the resource before performing operations on it and should release it afterwards. Each process should keep the resource for at least 10 time units (t.u). There should be at least 1 t.u. between any two operations.*"

Let us consider the input sequence $\sigma_1 = (1, acq_1) \cdot (3, op_1) \cdot (3.5, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (10, rel_1)$ (where each event is composed of an action associated with a date, indicating the time instant at which the action is received as input). The monitor receives the first action acq_1 at $t = 1$, followed by op_1 at $t = 3$, etc. At $t = 1$ (resp. $t = 3$), the monitor can output action acq_1 (resp. op_1) because both sequences $(3, op_1)$ and $(1, acq_1) \cdot (3, op_1)$ satisfy specification S_1. At $t = 3.5$, when the second action op_1 is input, the enforcer determines that this action should be delayed by 0.5 t.u. to ensure the constraint that 1 t.u. should elapse between occurrences of op_1 actions. Hence, the second

[2] We shall see in Section 3.2 how to formalise these specifications by timed automata.

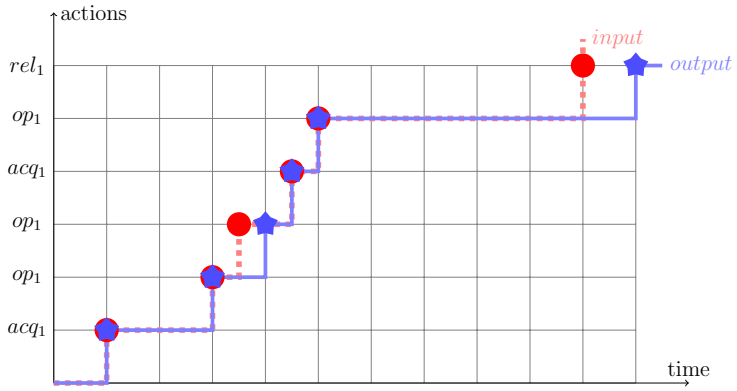


Figure 3: Behavior of an enforcement mechanism for specification S_1 on σ_1.

action op_1 is released at $t = 4$. At $t = 4.5$, when action acq_1 is received, the enforcer releases it immediately since this action is allowed by the specification with no time constraint. Similarly, at $t = 5$, an op_1 action is received and is released immediately because at least 1 t.u. elapsed since the previous op_1 action was released as output. At $t = 10$, when action rel_1 is received, it is delayed by 1 t.u. to ensure that the resource is kept for at least 10 t.u. (the first acq_1 action was released at $t = 1$). Henceforth, as shown in Figure 3, the output of the enforcement mechanism for σ_1 is $(1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (11, rel_1)$.

Specification S_2. The specification states that "*After system initialisation, both processes should perform an operation (actions op_i) before 10 t.u. The operations of the different processes should be separated by 3 t.u.*" Let us consider the input sequence $\sigma_2 = (1, init_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (5, op_2) \cdot (6, op_2)$.

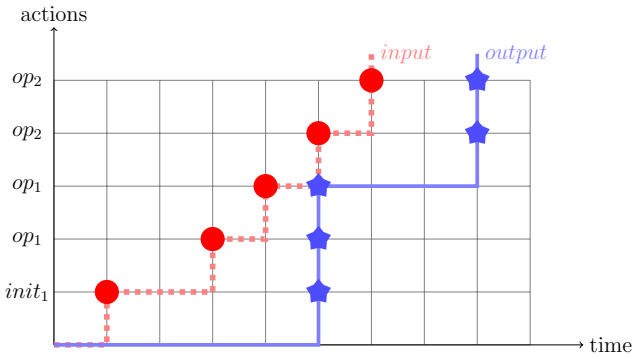


Figure 4: Behavior of an enforcement mechanism for specification S_2 on σ_2.

At $t = 1, 3, 4$, when the enforcement mechanism receives the actions, it cannot release them as output but memorises them since, upon each reception, the sequence of actions it received so far cannot be delayed so that a known continuation may satisfy specification S_2. At $t = 5$, upon the reception of action op_2, the sequence received so far can be delayed to satisfy specification S_2. Action $init_1$ is released at $t = 5$ because it is the earliest possible date: a smaller date would be already elapsed. The two actions op_1 are also released at $t = 5$, because there are no timing constraints on them. The first action op_2 is released at $t = 8$ to ensure a delay of at least 3 t.u. with the first op_1 action. The second action op_2,

received at $t = 6$, is also released at $t = 8$, since it does not need to be delayed more than after the preceding action. Henceforth, as shown in Figure 4, the output of the enforcement mechanism for σ_2 is $(5, init_1) \cdot (5, op_1) \cdot (5, op_1) \cdot (8, op_2) \cdot (8, op_2)$.

Specification S_3. The specification states that "*Operations op_1 and op_2 should execute in a transactional manner. Both actions should be executed, in any order, and any transaction should contain one occurrence of op_1 and op_2. Each transaction should complete within 10 t.u. Between operations op_1 and op_2, occurrences of operation op can occur. There is at least 2 t.u. between any two occurrences of any operation.*"

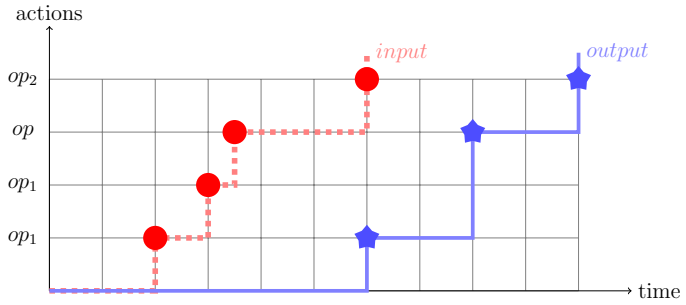


Figure 5: Behavior of an enforcement mechanism for specification S_3 on σ_3.

Let us consider the input sequence $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. At $t = 2$, the monitor can not output action op_1 because this action alone does not satisfy the specification (and the monitor does not yet know the next events i.e., actions and dates). If the next action was op_2, then, at the date of its reception, the monitor could output action op_1 followed by op_2, as it could choose dates for both actions in order to satisfy the timing constraints. At $t = 3$ the monitor receives a second op_1 action. Clearly, there is no possible date for these two op_1 actions to satisfy specification S_3, and no continuation could solve the situation. The monitor thus suppresses the second op_1 action, since this action is the one that prevents satisfiability in the future. At $t = 3.5$, when the monitor receives action op, the input sequence still does not satisfy the specification, but there exists an appropriate delaying of such action so that with future events, the specification can be satisfied. At $t = 6$, the monitor receives action op_2, it can decide that action op_1 followed by op and op_2 can be released as output with appropriate delaying. Thus, the date associated with the first op_1 action is set to 6 (the earliest possible date, since this decision is taken at $t = 6$), 8 for action op (since 2 is the minimal delay between those actions satisfying the timing constraint), and 10 for action op_2. Henceforth, as shown in Figure 5, the output of the enforcer for σ_3 is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$.

Specification S_4. The specification states that "*Processes should behave in a transactional manner, where each transaction consists of an acquisition of the resource, at least one operation on it, and then its release. After the acquisition of the resource, the operations on the resource should be done within 10 t.u. The resource should not be released less than 10 t.u. after acquisition. There should be no more than 10 t.u. without any ongoing transaction.*"

Let us consider the input sequence $\sigma_4 = (1, acq_i) \cdot (2, op_i) \cdot (3, rel_i)$. Before $t = 3$, no output can be produced, since no transaction is complete, and events must be stored. At $t = 3$, when the monitor receives rel_i, it can decide that the three events acq_i, op_i, and rel_i can be released as output with appropriate delaying. Thus, the date associated with the two first actions acq_i and op_i is set to 3, since this is the minimal decision date. Moreover, to satisfy the timing constraint on release actions

after acquisitions, the date associated to the last event rel_i is set to 13. The output of the enforcement mechanism for σ_4 is then $(3, acq_i) \cdot (3, op_i) \cdot (13, rel_i)$.

Let us now consider the input sequence $\sigma'_4 = (3, acq_i) \cdot (7, op_i) \cdot (13, rel_i)$. The monitor observes action acq_i followed by an op_i and a rel_i actions only at date $t = 13$. Hence, the date associated with the first action in the output should be at least 13, which is the minimal decision date. However, if the monitor chooses a date for acq_i which is strictly greater than 10, the timing constraint cannot be satisfied. Consequently, the output of the monitor remains always empty. Notice however that the input sequence provided to the monitor satisfies the specification. Nevertheless, the monitor cannot release any event as output as it cannot take a decision until it receives action rel_i at date $t = 13$, which affects the date (i.e., the absolute time instant when it can be released as output) of the first action acq_i, thus falsifying the constraints.

Discussion. Specification S_4 illustrates an important issue of enforcement in the timed setting, exhibited in this paper: because input timed words are seen as streams of events with dates, for some properties, there exist some input timed words that cannot be enforced, even though they either already satisfy the specification, or could be delayed to satisfy the specification (if they were known in advance). For instance, we shall see that specifications S_1, S_2, and S_3 do not suffer from this issue, while S_4 does. Actually, it turns out that enforcement monitors face some constraints due to streaming: they need to memorise input timed events before taking decision, but meanwhile, time elapses and this influences the possibility to satisfy the considered specification. Nevertheless, the synthesis of enforcement mechanisms proposed in this paper works for all regular timed properties, which means that the synthesised enforcement mechanisms still satisfy their requirements (soundness, transparency, optimality, and physical constraint), even though the output may be empty for some input timed words.

3. Preliminaries and notation

We first recall some basic notions on untimed languages (Section 3.1). We then introduce timed words and languages (Section 3.2) and extend previous notions in a timed setting (Section 3.2). Finally, we introduce some orders on timed words that will be used in runtime enforcement (Section 3.3).

3.1. Untimed languages

A (finite) word over an alphabet A is a finite sequence $w = a_1 \cdot a_2 \cdots a_n$ of elements of A. The *length* of w is n and is noted $|w|$. The empty word over A is denoted by ϵ_A, or ϵ 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 any subset $\mathcal{L}$ of A^*.

The *concatenation* of two words w and w' is noted $w \cdot w'$. A word w' is a *prefix* of a word w, noted $w' \preccurlyeq w$, 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 $\mathrm{pref}(w)$ denotes the *set of prefixes* of w and subsequently, $\mathrm{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathcal{L}} \mathrm{pref}(w)$ is the set of prefixes of words in $\mathcal{L}$. A language $\mathcal{L}$ is *prefix-closed* if $\mathrm{pref}(\mathcal{L}) = \mathcal{L}$ and *extension-closed* if $\mathcal{L} \cdot A^* = \mathcal{L}$.

Given two words u and v, $v^{-1} \cdot u$ is the *residual* of u by v and denotes the word w, such that $v \cdot w = u$, 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 $\mathcal{L} \subseteq A^*$ and a word $v \in A^*$, the residual of $\mathcal{L}$ by v is the language $v^{-1} \cdot \mathcal{L} \stackrel{\text{def}}{=} \{w \in A^* \mid v \cdot w \in \mathcal{L}\}$. It is the set of suffixes of words that, 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 prefix v.

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

Given two words w and w', we say that w' *is a subsequence of* w, noted $w' \triangleleft w$, if there exists an increasing mapping $k : [1, |w'|] \to [1, |w|]$ (i.e., $\forall i, j \in [1, |w'|] : i < j \implies k(i) < k(j)$) such that $\forall i \in [1, |w'|] : w'_{[i]} = w_{[k(i)]}$. Notice that, k being increasing entails that $|w'| \leq |w|$. Intuitively, the image of $[1, |w'|]$ by function k is the set of indexes of letters of w that are "kept" in w'.

Given an n-tuple of symbols $e = (e_1, \ldots, e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$). Operator Π_i is naturally extended to sequences of n-tuples of symbols to produce the sequence formed by the concatenation of the projections on the i-th element of each tuple.

3.2. Timed words and languages

As sketched in Section 2, input and output streams are seen as sequences of events composed of a date and an action, where the date is interpreted as the absolute date when the action is received by the enforcement mechanism. In what follows, we formalise input and output streams with timed words, and related notions, generalising the untimed setting.

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

A timed word over alphabet Σ is a finite sequence of events $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$, where $(t_i)_{i \in [1,n]}$ is a non-decreasing sequence in $\mathbb{R}_{\geq 0}$. We denote by $\mathrm{start}(\sigma) \stackrel{\text{def}}{=} t_1$ the starting date of σ and $\mathrm{end}(\sigma) \stackrel{\text{def}}{=} t_n$ its ending date (with the convention that the starting and ending dates are equal to 0 for the empty timed word ϵ).

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

The concatenation of timed words however requires more attention, as when concatenating two timed words, one should ensure that the result is a timed word, i.e., dates should be non-decreasing. This is ensured as soon as the ending date of the first timed word does not exceed the starting date of the second one. 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 $\mathrm{end}(\sigma) \leq \mathrm{start}(\sigma')$, 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 $\sigma \cdot \epsilon \stackrel{\text{def}}{=} \epsilon \cdot \sigma \stackrel{\text{def}}{=} \sigma$. 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 Σ^* (i.e., dates are ignored).

Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in \mathrm{tw}(\Sigma)$, we define the *observation of σ at date t* as the prefix of σ that can be observed at date t. It is defined as the maximal prefix of σ whose ending date is lower than t:

$$\mathrm{obs}(\sigma, t) \stackrel{\text{def}}{=} \max_{\preccurlyeq} \{\sigma' \in \mathrm{pref}(\sigma) \mid \mathrm{end}(\sigma') \leq t\}.$$

3.3. Preliminaries to runtime enforcement

Apart from the prefix order $\preccurlyeq$ (defined in Section 3.1), the following partial orders on timed words will be useful for enforcement.

Delaying order $\succcurlyeq_d$. For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, we say that σ' *delays* σ (noted $\sigma' \succcurlyeq_d \sigma$) iff they have the same untimed projection but the dates of events in σ' exceed the dates of corresponding events in σ. Formally:

$$\sigma' \succcurlyeq_d \sigma \stackrel{\text{def}}{=} \Pi_\Sigma(\sigma') = \Pi_\Sigma(\sigma) \wedge \forall i \in [1, |\sigma|] : \mathrm{date}(\sigma'_{[i]}) \geq \mathrm{date}(\sigma_{[i]}).$$

Sequence σ' is obtained from σ by keeping all actions, but with a potential increase in dates.

For example, $(4, a) \cdot (7, b) \cdot (9, c) \succcurlyeq_d (3, a) \cdot (5, b) \cdot (8, c)$. Note that delays between events may be decreased, e.g., between b and c, but absolute dates are increased.

Delaying subsequence order $\lhd_d$. For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, we say that σ' *is a delayed subsequence of* σ (noted $\sigma' \lhd_d \sigma$) iff there exists a subsequence σ'' of σ such that σ' delays σ''. Formally:

$$\sigma' \lhd_d \sigma \stackrel{\text{def}}{=} \exists \sigma'' \in \mathrm{tw}(\Sigma) : (\sigma'' \lhd \sigma \wedge \sigma' \succcurlyeq_d \sigma'') .$$

Sequence σ' is obtained from σ by first suppressing some actions, and then increasing the dates of the actions that are kept. This order will be used to characterise output timed words with respect to input timed words in enforcement monitoring when suppressing and delaying events.

For example, $(4, a) \cdot (9, c) \lhd_d (3, a) \cdot (5, b) \cdot (8, c)$ (event $(5, b)$ has been suppressed while a and c are shifted in time).

Lexical order $\preceq_{\mathrm{lex}}$. This order is useful to choose a unique timed word among some with same untimed projection. For two timed words σ, σ' with same untimed projection (i.e., $\Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$), the order $\preceq_{\mathrm{lex}}$ is defined inductively as follows: $\epsilon \preceq_{\mathrm{lex}} \epsilon$, and for two events with identical actions (t, a) and (t', a), $(t, a) \cdot \sigma \preceq_{\mathrm{lex}} (t', a) \cdot \sigma'$ if $t \leq t' \vee (t = t' \wedge \sigma \preceq_{\mathrm{lex}} \sigma')$. For example $(3, a) \cdot (5, b) \cdot (8, c) \cdot (11, d) \preceq_{\mathrm{lex}} (3, a) \cdot (5, b) \cdot (9, c) \cdot (10, d)$.

Choosing a unique timed word with minimal duration $\min_{\preceq_{\mathrm{lex}},\mathrm{end}}$. Given a set of timed words with same untimed projection, $\min_{\preceq_{\mathrm{lex}},\mathrm{end}}$ selects the minimal timed word w.r.t. the lexical order among timed words with minimal ending date: 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 $E \subseteq \mathrm{tw}(\Sigma)$ 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 $\min_{\preceq_{\mathrm{lex}},\mathrm{end}}(E) = \min_{\preceq_{\mathrm{lex}}}(\min_{\preceq_{\mathrm{end}}}(E))$ where $\sigma \preceq_{\mathrm{end}} \sigma'$ if $\mathrm{end}(\sigma) \leq \mathrm{end}(\sigma')$, for $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$.

4. Timed languages and properties as timed automata

Timed automaton is a usual model used to specify properties of sequences of events where timing between them matters. In this section, we introduce timed automata as a specification formalism for timed properties (Section 4.1). We describe a partitioning of the states of timed automata (Section 4.2). The partitioning allows to distinguish behaviours according to i) whether they currently satisfy or violate the property, and ii) whether or not this remains true for future behaviours. Finally, we present some sub-classes of regular properties (Section 4.3).

4.1. Timed automata

A timed automaton [11] is a finite 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 an element of $\mathbb{R}^X_{\geq 0}$, that is, a function from X to $\mathbb{R}_{\geq 0}$. For $\nu \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ is the valuation assigning $\nu(x) + \delta$ to each clock x of X. Given a set of clocks $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation ν where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of *guards*, i.e., 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 $\nu \in \mathbb{R}^X_{\geq 0}$, we write $\nu \models g$ when g holds according to ν.

Definition 1 (Timed automata). A *timed automaton* (TA) is a tuple $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, such that L is a finite set of *locations* with $l_0 \in L$ the *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 a set of *accepting locations*.

Example 1 (Timed automata). Let us consider again the specifications introduced in Section 2 where two processes access to and operate on a common resource. The global alphabet of events is $\Sigma \stackrel{\text{def}}{=} \{init, acq_1, rel_1, op_1, acq_2, rel_2, op_2, op\}$. The specifications on the behaviour of the processes introduced in Section 2 are formalised with the TAs in Figure 6. Accepting locations are denoted by squares.

S_1 The specification is formalised by the automaton depicted in Figure 6a with alphabet $\Sigma_1^i \stackrel{\text{def}}{=} \{rel_i, acq_i, op_i\}$ for process i, $i \in \{1, 2\}$. The automaton has two clocks x and y, where clock x serves to keep track of the duration of the resource acquisition whereas clock y keeps track of the time elapsing between two operations. Both locations of the automaton are accepting and there are two implicit transitions from location l_1 to a trap state: i) upon action rel_i when the value of clock x is strictly lower than 10, and ii) upon action op_i when the value of clock y is strictly lower than 1.

S_2 The specification is formalised by the automaton depicted in Figure 6b with alphabet $\Sigma_2 \stackrel{\text{def}}{=} \{init, op_1, op_2\}$. The automaton has two clocks, where clock x keeps track of the time elapsed since initialisation, whereas clock y keeps track of the time elapsing between the operations of the two different processes.

S_3 The specification is formalised by the automaton depicted in Figure 6c with alphabet $\Sigma_3 \stackrel{\text{def}}{=} \{op, op_1, op_2\}$. Clock x keeps track of the time elapsing since the beginning of the transaction, whereas clock y keeps track of the time elapsing between any two operations.

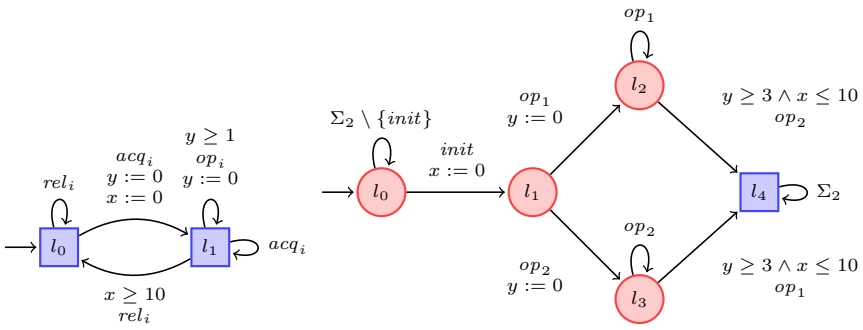
S_4 The specification is formalised by the automaton depicted in Figure 6d with alphabet $\Sigma_4^i \stackrel{\text{def}}{=} \{acq_i, op_i, rel_i\}$. Clock x keeps track of the duration of a currently executing transaction, whereas clock y keeps track of the time elapsing between two transactions.

The semantics of a TA is defined as follows.

Definition 2 (Semantics of timed automata). The *semantics* of a TA is a *timed transition system* $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ where $Q = L \times \mathbb{R}_{\geq 0}^X$ is the (infinite) set of *states*, $q_0 = (l_0, \nu_0)$ is the *initial state* where ν_0 is the valuation that maps every clock in X to 0, $Q_F = F \times \mathbb{R}_{\geq 0}^X$ is the set of *accepting states*, $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ 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 a set of transitions of the form $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ with $\nu' = (\nu + \delta)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ such that $\nu + \delta \models g$ for $\delta \in \mathbb{R}_{\geq 0}$.

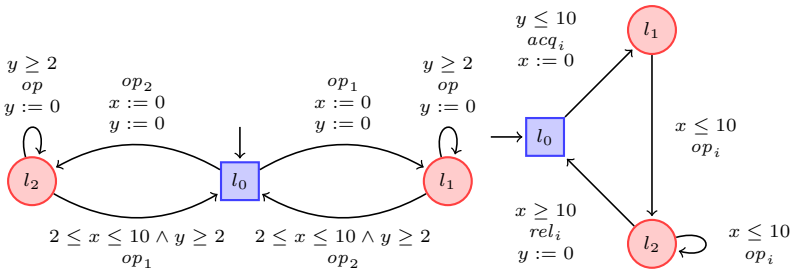
In the following, we consider a timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ with its semantics $[\![\mathcal{A}]\!]$. $\mathcal{A}$ is said to be *deterministic* whenever for any location l and any two distinct transitions (l, g_1, a, Y_1, l_1') and (l, g_2, a, Y_2, l_2') with source l and same action a in Δ, the conjunction of guards g_1 and g_2 is unsatisfiable. $\mathcal{A}$ is said to be *complete* whenever for any location $l \in L$ and any action $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labelled by a is valid. In the remainder of this paper, we shall consider only deterministic and complete timed automata, and, automata refer to timed automata.

Remark 1 (Completeness and determinism). Although we restrict the presentation to deterministic TAs, results may easily be extended to non-deterministic TAs, with slight adaptations required to the vocabulary and when synthesising an enforcement monitor. Regarding completeness, for readability of TA examples, if no transition can be triggered upon the reception of an event, a TA implicitly moves to a non-accepting trap location (i.e., where all actions are looping with no timing constraint).



(a) A safety automaton for S_1.

(b) A co-safety automaton for S_2.



(c) A regular automaton for S_3.

(d) A regular automaton for S_4. Note the implicit trap state reached from l_0 when $y > 10$ ensuring that there is no more than 10 t.u. without transaction.

Figure 6: Some examples of timed automata.

Remark 2 (Other definitions of timed automata). The definition of timed automata used in this paper is as the initial (and general) one proposed in [11] except that we do not use the Büchi acceptance condition because we deal with finite words. Even though we restrict constants in guards to be integers, and will see in Section 7 that TAs with rational constants may be necessary in the computation, but those TAs can be transformed into integral TAs. Other definitions of timed automata have been proposed (see e.g., [12] for details). For instance, timed safety automata [13] are a simplified version of the original timed automata where invariants on locations replace the Büchi condition, as used in UPPAAL [14]. Several classes of determinisable automata with restrictions on the resets of clocks have been proposed. Event-recording (resp. event-predicting) timed automata [15] are timed automata with a clock associated to each action that records (resp. predicts) the time elapsed since the last occurrence (resp. the time of the next occurrence) of that action; event-clock automata have event-recording and event-predicting clocks.

A *run* ρ of $\mathcal{A}$ from a state $q \in Q$ 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 $n \in \mathbb{N}$. The set of runs from the initial state $q_0 \in Q$ is denoted $\mathrm{Run}(\mathcal{A})$ and $\mathrm{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs starting in q_0 and *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$.

The *trace* started at date t of the run ρ is the timed word $(t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ where $\forall i \in [1, n] : t_i = t + \sum_{j=1}^{i} \delta_j$ (the date of a_i is the sum of delays of the i first events plus t). We note

$q \xrightarrow{w}_t q_n$ in this case, and generalise to $q \xrightarrow{w}_t P$ when $q_n \in P$ for a subset P of Q. We note $\xrightarrow{w}$ for $\xrightarrow{w}_0$. We note $\mathcal{L}(\mathcal{A})$ the set of traces started at date 0 of $\mathrm{Run}(\mathcal{A})$. We extend this notation to $\mathcal{L}_{Q_F}(\mathcal{A})$ as the set of traces of runs in $\mathrm{Run}_{Q_F}(\mathcal{A})$. We thus say that a timed word is accepted by $\mathcal{A}$ if it is the trace started at date 0 of an accepted run.

Example 2 (Runs and traces of a timed automaton). Consider the automaton in Figure 6a. A possible *run* of this automaton from the initial state $(l_0, 0, 0)$ is the sequence of moves $(l_0, 0, 0) \xrightarrow{(1, acq_1)} (l_1, 0, 0) \xrightarrow{(2, op_1)} (l_0, 2, 0) \xrightarrow{(1, op_1)} (l_1, 3, 0) \xrightarrow{(0.5, acq_1)} (l_1, 3.5, 0.5) \xrightarrow{(0.5, op_1)} (l_1, 4, 0)$. The trace starting at date 0 of this run is the timed word $w_t = (1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1)$. We have $(l_0, 0, 0) \xrightarrow{w_t} (l_1, 4, 0)$.

We now introduce the product of timed automata which is useful to intersect languages recognized by timed automata.

Definition 3 (Product of timed automata). Given two TAs $\mathcal{A}_1 = (L_1, l_1^0, X_1, \Sigma, \Delta_1, F_1)$ and $\mathcal{A}_2 = (L_2, l_2^0, X_2, \Sigma, \Delta_2, F_2)$ with disjoint sets of clocks, their product is the TA $\mathcal{A}_1 \times \mathcal{A}_2 \stackrel{\text{def}}{=} (L, l_0, X, \Sigma, \Delta, F)$ where $L = L_1 \times L_2$, $l_0 = (l_0^1, l_0^2)$, $X = X_1 \cup X_2$, $F = F_1 \times F_2$, and $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*, with $((l_1, l_2), g_1 \wedge g_2, a, Y_1 \cup Y_2, (l_1', l_2')) \in \Delta$ if $(l_1, g_1, a, Y_1, l_1') \in \Delta_1$ and $(l_2, g_2, a, Y_2, l_2') \in \Delta_2$.

It is easy to check that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

4.2. Partition of states of $[\![\mathcal{A}]\!]$

Given a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, with semantics $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \to, Q_F)$, the set of states Q of $[\![\mathcal{A}]\!]$ can be partitioned into four subsets *good* (G), *currently good* (G^c), *currently bad* (B^c) and *bad* (B), based on whether a state is accepting or not, and whether accepting or non accepting states are reachable or not.

Formally, Q is partitioned into $Q = G^C \cup G \cup B^C \cup B$ where $Q_F = G^C \cup G$ and $Q \setminus Q_F = B^C \cup B$ and

- $G^C = Q_F \cap \mathrm{pre}^*(Q \setminus Q_F)$ i.e., the set of *currently good* states is the subset of accepting states from which non-accepting states are reachable,
- $G = Q_F \setminus G^C = Q_F \setminus \mathrm{pre}^*(Q \setminus Q_F)$ i.e., the set of *good* states is the subset of accepting states from which only accepting states are reachable,
- $B^C = (Q \setminus Q_F) \cap \mathrm{pre}^*(Q_F)$ i.e., the set of *currently bad* states is the subset of non-accepting states from which accepting states are reachable,
- $B = (Q \setminus Q_F) \setminus \mathrm{pre}^*(Q_F)$ i.e., the set of *bad* states is the subset of non-accepting states from which only non-accepting states are reachable.

where, for a subset P of Q, $\mathrm{pre}^*(P)$ denotes the set of states from which the set P is reachable.

It is well known that reachability of a set of locations is decidable using the classical zone (or region) symbolic representation (see [16]) and is PSPACE-complete. Since Q_F is the set of states with location F, this result can be used to compute the partition of Q.

By definition, from good (resp. bad) states, one can only reach good (resp. bad) states. Consequently, a run of a TA traverses currently good and/or currently bad states, and may eventually reach a good state and remain in good states, or a bad state and remain in bad states, or in pathological cases, it can directly start in good or bad states. This partition will be useful to characterise the classes of safety and co-safety timed properties, as explained in Section 4.3, and later for the synthesis of enforcement mechanisms.

4.3. Some sub-classes of regular timed properties

Regular, safety, and co-safety timed properties. In this paper, a timed property is defined by a timed language $\varphi \subseteq \mathrm{tw}(\Sigma)$ that can be recognised by a timed automaton. That is, we consider the set of regular timed properties. Given a timed word $\sigma \in \mathrm{tw}(\Sigma)$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$. Safety (resp. co-safety) properties are sub-classes of regular timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). In this paper, the classes are characterised as follows:

Definition 4 (Regular, safety, and co-safety properties). We consider the following three classes of timed properties.

- Regular properties are the properties that can be defined by languages accepted by a TA.
- Safety properties are the non-empty prefix-closed timed languages that can be accepted by a TA.
- Co-safety properties are the non-universal[3] extension-closed timed languages that can be accepted by a TA.

The sets of safety and co-safety properties are subsets of the set of regular properties.

Safety and co-safety timed automata. In the sequel, we shall only consider the properties that can be defined by deterministic and complete timed automata (Definition 1). Note that some of these properties can be defined using a timed temporal logic such as a subclass of MTL, which can be transformed into timed automata using the technique described in [17, 18].

We now define syntactic restrictions on TAs that guarantee that a regular property defined by a TA defines a safety or a co-safety property.

Definition 5 (Safety and co-safety TA). Let $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ be a complete and deterministic TA, where $F \subseteq L$ is the set of accepting locations. $\mathcal{A}$ is said to be:

- a *safety* TA if $l_0 \in F \wedge \nexists (l, g, a, Y, l') \in \Delta : l \in L \setminus F \wedge l' \in F$;
- a *co-safety* TA if $l_0 \notin F \wedge \nexists (l, g, a, Y, l') \in \Delta : l \in F \wedge l' \in L \setminus F$.

It is then easy to check that safety (respectively co-safety) TAs define safety (respectively co-safety) properties[4]. Intuitively, a safety TA starts in the accepting location l_0 and has no transition from non-accepting to accepting locations. Thus, either all reachable locations are accepting (in this case, the TA recognises the universal language since it is complete), or the TA stays in accepting locations before possibly jumping definitively to non-accepting locations. At the semantical level a safety TA either has only good states (case of the universal language), or its runs start in the set of currently good states and may definitively jump in either the set of bad or the set of good states (no currently bad state can be reached). Thus, a safety TA defines a prefix-closed language. Conversely, a co-safety TA starts in the non-accepting location l_0 and has no transition from accepting to non-accepting locations. Thus, either all reachable locations are non-accepting (in this case, the TA recognises the empty language), or it stays in non-accepting locations before possibly jumping definitively to accepting locations. At the semantic level, a co-safety TA either only has bad states (case of the empty language), or its runs start in the set of currently bad states and may definitively jump in either the set of good states or the set of bad states (currently good state cannot be reached). Thus, a co-safety TA defines an extension-closed language.

[3] The universal property over $\mathbb{R}_{\geq 0} \times \Sigma$ is $\mathrm{tw}(\Sigma)$.

[4] As one can observe, these definitions of safety and co-safety TAs slightly differ from the usual ones by expressing constraints on the initial state. As a consequence of these constraints, consistently with Definition 4, the empty and universal properties are ruled out from the set of safety and co-safety properties, respectively.

Example 3 (Classes of timed automata). Let us consider again the specifications introduced in Example 6. We formalise specification S_i as property φ_i, $i = 1, \ldots, 4$. Property φ_1 is a safety property specified by the safety TA in Figure 6a (leaving accepting locations is definitive). Property φ_2 is a co-safety property specified by the co-safety TA in Figure 6b (leaving non-accepting locations is definitive). Property φ_3 is specified by the TA in Figure 6c. Property φ_4 is specified by the TA in Figure 6d. Both properties φ_3 and φ_4 are regular, but neither safety nor co-safety properties. In the underlying automata, runs may alternate between accepting and non-accepting locations, thus the languages that they define are neither prefix nor extension-closed.

5. Enforcement monitoring in a timed context

We now introduce our enforcement monitoring framework (Section 5.1) and specify the expected constraints on the input/output behaviour of enforcement mechanisms (Section 5.2).

5.1. General principles

To ease the design and implementation of enforcement monitoring mechanisms in a timed context, we describe enforcement mechanisms at three levels of abstraction: *enforcement functions*, *enforcement monitors*, and *enforcement algorithms*. An enforcement function describes the transformation of an input timed word into an output timed word at an abstract level where the whole input timed word is considered. In this section, we first formalise the constraints enforcement functions must satisfy, which reflect both physical constraints related to time, and required properties relating the input to the output. In Section 6, we shall define such enforcement functions, and prove that they satisfy the constraints. An enforcement monitor is a more concrete view and defines the operational behaviour of the enforcement mechanism over time. In Section 7, we shall define enforcement monitors as an extended transition systems and we prove that, for a given property φ, the associated enforcement monitor implements the corresponding enforcement function. In other words, an enforcement function serves as an abstract description (black-box view) of an enforcement monitor, and an enforcement monitor is the operational description of an enforcement function. An enforcement algorithm (see Section 8) is an implementation of an enforcement monitor.

5.2. Constraints on an enforcement mechanism

At an abstract level, an enforcement mechanism for a given property φ can be seen as a function which takes as input a timed word and outputs a timed word. At this level, the input is considered as a whole, and the output is the corresponding whole timed word eventually produced, after an unbounded time elapse. In other words, the delay to observe the input and to produce the output is not considered. This is schematised in Figure 7 and defined in Definition 6.

Definition 6 (Enforcement function signature). For a timed property φ, an enforcement mechanism behaves as a function, called *enforcement function* $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$.

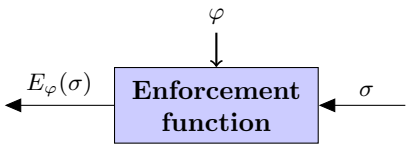


Figure 7: Enforcement function.

An enforcement function E_φ models a mechanism that reads some input timed word σ from an event emitter, which is possibly incorrect w.r.t. φ, and transforms it into a timed word that satisfies φ which is output to the event receiver.

Before providing the actual definition of enforcement function in Section 6, we define the constraints that should be satisfied by an enforcement mechanism. The following constraints can serve as a specification of the expected behaviour of enforcement mechanisms for timed properties, that can delay and suppress events.

An enforcement mechanism should first satisfy some *physical constraint* reflecting the streaming of events: the output stream can only be modified by appending new events to its tail. Second, it should be *sound* w.r.t. the monitored property, meaning that it should correct input words according to φ if possible, and otherwise produce an empty output. Third, it should be *transparent*, which means that it is only allowed to shift events in time while keeping their order (we refer to such kind of mechanisms as time retardants) and suppress some events. These constraints are formalised in the following definition:

Definition 7 (Constraints on an enforcement mechanism). Given a timed property φ, an enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, should satisfy the following constraints:

- **Physical constraint:**

$$\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma') \qquad (\mathbf{Phy}).$$

- **Soundness:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon \qquad (\mathbf{Snd}).$$

- **Transparency:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \lhd_d \sigma \qquad (\mathbf{Tr}).$$

The physical constraint **(Phy)** means that E_φ is monotonic: the output produced for an extension σ' of an input word σ extends the output produced for σ. This stems from the fact that, over time, what is actually output by the enforcement function is a continuously growing timed word, i.e., what is output for a given input timed word can only be modified by appending new events with greater dates. Soundness **(Snd)** means that the output either satisfies property φ, or is empty. This allows to output nothing if there is no way to satisfy φ. Note that, together with the physical constraint, soundness implies that no event can be appended to the output before being sure that the property will be eventually satisfied with subsequent output events. Transparency **(Tr)** means that the output is a delayed subsequence of the input σ, thus the enforcement function is allowed to either suppress input events, or increase their dates while preserving their order.

It can be easily checked on the examples in Section 2 that the output sequences satisfy the constraints of enforcement mechanisms.

Remark 3. The soundness, transparency, and physical constraints describe the expected input/output behaviour of an enforcement function for the whole input sequence. Note that it however does not strongly constrain the output. In particular, an enforcement function that never produces any output complies to these constraints. However, to be practical, an actual enforcement function should also provide some guarantees on the output sequence it produces in terms of length and delay w.r.t. the input sequence. Such guarantees are specified by an optimality property in Section 6.

6. Enforcement functions: input/output description of enforcement mechanisms

We now define an enforcement function dedicated to a desired property φ. Its purpose is to define, at an abstract level, for any input word σ, the output word $E_\varphi(\sigma)$ expected from an enforcement mech-

anism that works as a delayer with suppression, where suppression only happens upon the reception of an event that prevents any satisfaction of the property in the future.

First, we discuss some preliminaries (Section 6.1) regarding the consequences of the choice of suppression strategy on efficiency and on the possible output sequences of the enforcement function. Then, we define the enforcement function itself, and prove in Section 6.2 that this functional definition satisfies the physical, soundness, and transparency constraints. We also prove that the enforcement function satisfies some optimality criterion with the chosen suppression strategy. Finally, in Section 6.3, we explain how the enforcement function behaves over time (how a given input sequence is consumed over time, and how the output is released in an incremental fashion).

6.1. Preliminaries to the definition of enforcement functions

An enforcement mechanism needs to memorise events since, for some properties (typically co-safety properties), upon the reception of some input timed word, the property might not be yet satisfiable by delaying, but a continuation of the input may allow satisfaction. For more general properties (which are neither safety nor co-safety properties), there may exist some prefix for which the property is satisfiable by delaying the input, thus dates can be chosen for these events. For efficiency reasons, and for our enforcement mechanisms to be amenable to online implementations, we also want to build the output in a fashion that is as incremental as possible. Enforcement mechanisms should thus take decisions on dates of output events as soon as possible. Still for efficiency considerations, we impose that suppression should occur only when necessary, i.e., when, upon the reception of a new event, there is no possibility to satisfy the property, whatever is the continuation of the input. Moreover, we decide to suppress the last received event only because it causes the unsatisfiability (even) if delayed. Note, when an enforcement mechanism decides not to suppress an action, it should not modify its decision in the future. Our choice of suppression strategy mainly stems from efficiency reasons. We discuss our choice and possible alternatives in Remark 4 in Section 6.2 (p. 18).

6.2. Definition of enforcement functions

As intuitively explained in the motivating examples of Section 2, during enforcement of an input timed word σ, some subsequence of events σ_c is temporarily stored, until some new event (t, a) eventually allows to satisfy the property for the first time, or satisfy it again, by delaying the sequence $\sigma'_c = \sigma_c \cdot (t, a)$. For such a sequence σ'_c, the definition of an enforcement function shall use the set $\mathrm{CanD}(\sigma'_c)$ of candidate delayed sequences of σ'_c, independently of the property φ.

$$\mathrm{CanD}(\sigma'_c) = \{w \in \mathrm{tw}(\Sigma) \mid w \succcurlyeq_d \sigma'_c \wedge \mathrm{start}(w) \geq \mathrm{end}(\sigma'_c)\}\,.$$

The set $\mathrm{CanD}(\sigma'_c)$ is the set of timed words w that delay σ'_c, and start at or after the ending date of σ'_c (which is the date t of the last event (t, a) of σ'_c). As we shall see, $w \succcurlyeq_d \sigma'_c$ stems from the fact that we consider enforcement mechanisms as time retardants, while $\mathrm{start}(w) \geq \mathrm{end}(\sigma'_c)$ means that the eligible timed words should not start before the date of its last event (which is the current date t), as illustrated informally with specification S_3 in Section 2 and further discussed in Section 6.3.

With this preliminary notation, the enforcement function for a property φ can be defined as follows:

Definition 8 (Enforcement function). The enforcement function for a property φ is the function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ defined as:

$$E_\varphi(\sigma) = \Pi_1\left(\mathrm{store}_\varphi\left(\sigma\right)\right),$$

where $\mathrm{store}_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma) \times \mathrm{tw}(\Sigma)$ is defined as

$$\begin{array}{rcl} \mathrm{store}_\varphi(\epsilon) & = & (\epsilon, \epsilon) \\ \mathrm{store}_\varphi(\sigma \cdot (t,a)) & = & \begin{cases} (\sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} (\kappa_\varphi(\sigma_s, \sigma'_c)), \epsilon) & \text{if } \kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset, \\ (\sigma_s, \sigma_c) & \text{if } \kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c) = \emptyset, \\ (\sigma_s, \sigma'_c) & \text{otherwise,} \end{cases} \\ & & \text{with } \sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}, a \in \Sigma, \\ & & (\sigma_s, \sigma_c) = \mathrm{store}_\varphi(\sigma), \text{ and } \sigma'_c = \sigma_c \cdot (t,a), \end{array}$$

where, for $\mathcal{L} \subseteq \mathrm{tw}(\Sigma)$,

$$\kappa_{\mathcal{L}}(\sigma_s, \sigma'_c) \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma'_c) \cap \sigma_s^{-1} \cdot \mathcal{L}.$$

For a given input σ, function store_φ computes a pair (σ_s, σ_c) of timed words: σ_s, which is extracted by the projection function Π_1 to produce the output $E_\varphi(\sigma)$; σ_c is used as a temporary memory. The pair (σ_s, σ_c) should be understood as follows:

- σ_s is a delayed subsequence of the input σ, in fact of its prefix of maximal length for which the absolute dates can be computed to satisfy property φ;
- σ_c is a subsequence of the remaining suffix of σ for which the releasing dates of events, still have to be computed. It is a subsequence (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.

Function E_φ 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 (ϵ, ϵ). Otherwise, suppose that for the input σ the result of $\mathrm{store}_\varphi(\sigma)$ is (σ_s, σ_c) and consider a new received event (t,a). Now, the new timed word to correct is $\sigma'_c = \sigma_c \cdot (t,a)$. There are three possible cases, according to the vacuity of the two sets $\kappa_\varphi(\sigma_s, \sigma'_c)$ and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c)$. These sets are obtained respectively as the intersection of the set $\mathrm{CanD}(\sigma'_c)$ with $\sigma_s^{-1} \cdot \varphi$ and $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi)$. Let us recall that:

- $\mathrm{CanD}(\sigma'_c)$ is the set of timed words that delay σ'_c, and start at or after the ending date of σ'_c (i.e., the date of its last event (t,a)), since choosing an earlier date would cause the date to be already elapsed before the event could be released as output;
- $\sigma_s^{-1} \cdot \varphi = \{w \in \mathrm{tw}(\Sigma) \mid \sigma_s \cdot w \models \varphi\}$ is the set of timed words that satisfy φ after reading σ_s; similarly, since $\mathrm{pref}(\varphi) = \{v \in \mathrm{tw}(\Sigma) \mid \exists w' \in \mathrm{tw}(\Sigma) : v \cdot w' \models \varphi\}$ we get that $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi) = \{w \in \mathrm{tw}(\Sigma) \mid \exists w' \in \mathrm{tw}(\Sigma) : \sigma_s \cdot w \cdot w' \models \varphi\}$, and thus $\sigma_s^{-1} \cdot \mathrm{pref}(\varphi)$ is the set of timed words for which a continuation satisfies φ after reading σ_s.

Thus $\kappa_\varphi(\sigma_s, \sigma'_c)$ is the set of timed words w that belong to the candidate delayed sequences of σ'_c and such that $\sigma_s \cdot w$ satisfies φ; and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c)$ is the set of timed words w that belong to the candidate delayed sequences of σ'_c, and such that some additional continuation w' may satisfy φ, i.e., $\sigma_s \cdot w \cdot w' \models \varphi$. Note that, since $\kappa_\varphi(\sigma_s, \sigma'_c) \subseteq \kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c)$, we distinguish three different cases:

- If $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$ (and thus $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c) \neq \emptyset$), it is possible to choose appropriate dates for the timed word $\sigma'_c = \sigma_c \cdot (t,a)$ to satisfy φ. The minimal timed word in $\kappa_\varphi(\sigma_s, \sigma'_c)$ w.r.t. the lexicographic order is chosen among those with minimal ending date, and appended to σ_s; the second element of the pair is set to ϵ since all events memorised in $\sigma_c \cdot (t,a)$ are corrected and appended to σ_s.
- If $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c) = \emptyset$ (and thus $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$), it means that, whatever is the continuation of the current input $\sigma \cdot (t,a)$, there is no chance to find a correct delaying for (t,a). Thus, event (t,a) should be suppressed, leaving σ_c and σ_s unmodified.

- Otherwise, i.e., when $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$ but $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c) \neq \emptyset$, it means that it is not yet possible to choose appropriate dates for $\sigma'_c = \sigma_c \cdot (t, a)$ to satisfy φ, but there is still a chance to do it in the future, depending on the continuation of the input, if any. Thus σ_c is modified into $\sigma'_c = \sigma_c \cdot (t, a)$ in memory, but σ_s is left unmodified.

Remark 4 (Alternative strategies to suppress events). When there is no possibility to continue correcting the input sequence (i.e., $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$), we choose to erase only the last received event (t, a), since it is the one that causes this impossibility. However, other policies to suppress events could be chosen. In fact, one could choose to suppress any events in $\sigma_c \cdot (t, a)$, since dates of these events have not yet been chosen. This would then require to choose among all such subsequences, using an appropriate order. This may be rather complex to define, and, more importantly, computationally expensive because one would have to face the combinatorial explosion induced when considering the $2^{|\sigma_c \cdot (t,a)|}$ possible subsets of actions to suppress. Moreover, let us notice that enforcement mechanisms are purposed to work in an online fashion, hence making decisions on each reception of a new event. For this purpose, not reconsidering the suppression choices makes them definitive and lowers the computation related to suppression.

Proposition 1 *Given some property φ, its enforcement function E_φ as per Definition 8 satisfies the physical* ***(Phy)****, soundness* **(Snd)***, and transparency* **(Tr)** *constraints as per Definition 7.*

PROOF (OF PROPOSITION 1 - SKETCH ONLY). The proof of the physical constraint is a direct consequence of the definition of store_φ. The proofs of soundness and transparency follow the same pattern: they rely on an induction on the length of the input word σ. The induction steps use a case analysis, depending on whether the last input subsequence (i.e, the events in $\sigma_c \cdot (t, a)$) can be corrected or not. The complete proofs are given in Appendix A.1.

In addition to the physical, soundness, and transparency constraints, the functional definition also ensures that each subsequence is output as soon as possible, as expressed by the following proposition.

Proposition 2 (Optimality of enforcement functions) *Given some property φ, its enforcement function E_φ as per Definition 8 satisfies the following optimality constraint:*

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

where $\max^{\varphi}_{\prec,\epsilon}(\sigma)$ is the maximal strict prefix of σ belonging to φ, formally:

$$\max^{\varphi}_{\prec,\epsilon}(\sigma) \stackrel{\mathrm{def}}{=} \max_{\preceq}\left(\{\sigma' \in \varphi \mid \sigma' \prec \sigma\} \cup \{\epsilon\}\right).$$

For any input σ, if the output $E_\varphi(\sigma)$ is not empty, then (it satisfies φ by soundness and) the output can be separated into a prefix m which is the maximal strict prefix of $E_\varphi(\sigma)$ satisfying property φ, and a suffix w. The optimality condition focuses on this last part, which is the suffix that allows to satisfy (again) the property. However, since the property considers any input σ, the same holds for every prefix of the input that allows to satisfy φ by enforcement, thus for any such (temporary) last subsequence.

The optimality constraint expresses that, among those sequences w' that could have been chosen (see below), w is the minimal one in terms of ending date, and lexical order (this second minimality

ensures uniqueness). The "sequences that could have been chosen" are those such that $m \cdot w'$ satisfies the property, have the same events (thus can be produced by suppressing the same events), are delayed subsequences of the input σ, and have a starting date greater than or equal to $\mathrm{end}(\sigma)$, since $\mathrm{end}(\sigma)$ is the date at which w' is appended to the output, and thus a smaller date would be in the past of the output event.

PROOF (OF PROPOSITION 2 - SKETCH ONLY). The proofs rely on an induction on the length of the input word σ. The induction step uses a case analysis, depending on whether the last input subsequence (i.e, the events in $\sigma_c \cdot (t, a)$) can be corrected or not. The proof is given in Appendix A.2 (p. 47).

Remark 5 (On the optimality condition). Note that the condition $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ in Proposition 2 stems from the strategy chosen to suppress events (see Remark 4). If an enforcement function is defined, such that it is allowed to suppress any event in $\sigma_c \cdot (t, a)$, then the condition $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ in optimality should be removed. Moreover, note that optimality has to be defined in a recursive manner. Indeed, since enforcement mechanisms should produce output sequences in an incremental fashion, the optimal output that should be produced for an input sequence depends on the optimal outputs that have been produced for the prefixes of the input sequence. Because of the performance reasons mentioned in Remark 4, defining a more general notion of optimality (possibly parameterised by the suppression strategy) is left for future work.

6.3. Behavior of enforcement functions over time

At an abstract level, an enforcement function takes as input a timed word and computes as output the timed word that is eventually produced by the enforcement mechanism after some unbounded delay. However, at a more concrete level, enforcement obeys some temporal constraints relative to the current date t. Firstly, since the enforcement mechanism reads the input timed word σ as a stream, what it can effectively observe from σ at date t is its prefix $\mathrm{obs}(\sigma, t)$. Consequently, at date t, what it can compute from this observation is $E_\varphi(\mathrm{obs}(\sigma, t))$. Note that it is legal to do so since, by definition $\mathrm{obs}(\sigma, t)$ is a prefix of σ, and thus, by the physical constraint **(Phy)**, $E_\varphi(\mathrm{obs}(\sigma, t))$ is a prefix of the complete output $E_\varphi(\sigma)$. Now, $E_\varphi(\mathrm{obs}(\sigma, t))$ is a timed word where dates attached to events model the date when they should eventually be released as output. But at date t, only its prefix $\mathrm{obs}(E_\varphi(\mathrm{obs}(\sigma, t)), t)$ is effectively released as output. Now, notice that, since E_φ behaves as a time retardant (i.e., dates attached to output events exceed dates of corresponding input events), and $E_\varphi(\mathrm{obs}(\sigma, t))$ is a prefix of $E_\varphi(\sigma)$, we get $\mathrm{obs}(E_\varphi(\mathrm{obs}(\sigma, t)), t) = \mathrm{obs}(E_\varphi(\sigma), t)$. From this, we conclude that the released output at date t is $\mathrm{obs}(E_\varphi(\sigma), t)$. Finally, what is ready to be released at date t, but not yet released is the residual of $E_\varphi(\mathrm{obs}(\sigma, t))$ after observing $\mathrm{obs}(E_\varphi(\sigma), t)$ thus $\mathrm{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\mathrm{obs}(\sigma, t))$. The enforcement monitor described in the next section, which implements the enforcement function, takes care of this temporal behaviour.

Example 4 (Behavior of enforcement functions over time (see Figure 8)). Let us consider the input timed word $\sigma = (t_1, a_1) \cdot (t_2, a_2) \cdot (t_3, a_3) \cdot (t_4, a_4) \cdot (t_5, a_5) \cdot (t_6, a_6) \cdot (t_7, a_7)$, and let the output of the enforcement function be $E_\varphi(\sigma) = (t'_1, a_1) \cdot (t'_2, a_2) \cdot (t'_4, a_4) \cdot (t'_5, a_5) \cdot (t'_7, a_7)$. At time instant t:

- the observation of σ is $\mathrm{obs}(\sigma, t) = (t_1, a_1) \cdot (t_2, a_2) \cdot (t_3, a_3) \cdot (t_4, a_4) \cdot (t_5, a_5)$,
- the subsequence of remaining suffix of $\mathrm{obs}(\sigma, t)$ for which the releasing dates still have to be computed is $\sigma^t_{\mathrm{mc}} = (t_5, a_5)$,
- the output that the enforcement function can compute from $\mathrm{obs}(\sigma, t)$ is $E_\varphi(\mathrm{obs}(\sigma, t)) = (t'_1, a_1) \cdot (t'_2, a_2) \cdot (t'_4, a_4)$,
- the released output is $\mathrm{obs}(E_\varphi(\mathrm{obs}(\sigma, t)), t) = \mathrm{obs}(E_\varphi(\sigma), t) = (t'_1, a_1) \cdot (t'_2, a_2)$;
- the timed word ready to be released, denoted by σ^t_{ms}, is $\mathrm{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\mathrm{obs}(\sigma, t)) = (t'_4, a_4)$.

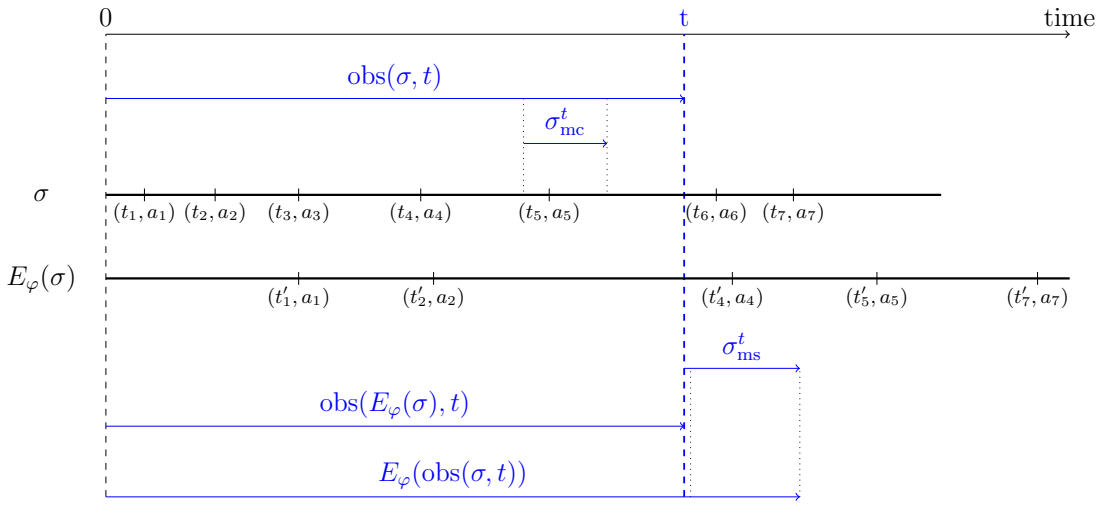


Figure 8: Behavior of enforcement functions over time.

Example 5 (Enforcement function). We illustrate how Definition 8 is applied to enforce specification S_3 (see Section 2), formalised by property φ_3, recognised by the automaton depicted in Figure 6c with Σ_3 ($= \{op_1, op_2, op\}$), and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Figure 9 shows the evolution of the observed input timed word $\mathrm{obs}(\sigma_3, t)$, the output of the store_φ function when the input timed word is $\mathrm{obs}(\sigma_3, t)$, and E_{φ_3}. Variable t keeps track of physical time, i.e., it contains the current date. When $t < 6$, the observed output is empty (since $E_{\varphi_3}(\mathrm{obs}(\sigma_3, t)) = \epsilon$). When $t \geq 6$, the observed output, is $\mathrm{obs}((6, op_1) \cdot (8, op) \cdot (10, op_2), t)$ (since $E_{\varphi_3}(\mathrm{obs}(\sigma_3, t)) = (6, op_1) \cdot (8, op) \cdot (10, op_2)$).

Example 6 (Enforcement function: a non-enforceable property). Consider specification S_4 formalised by property φ_4, recognised by the automaton depicted in Figure 6d with $\Sigma_4^i = \{acq_i, op_i, rel_i\}$, and the input timed word $\sigma_4 = (3, acq_i) \cdot (7, op_i) \cdot (12, rel_i)$. Figure 10 shows the evolution of the observation of the input timed word $\mathrm{obs}(\sigma_4, t)$, the output of the store_φ function when the input timed word is $\mathrm{obs}(\sigma_4, t)$, and E_{φ_4}. The output of the enforcement function is ϵ at any date because delaying action acq_i for 9 t.u. (i.e., until observing action rel_i) violates the timing constraint of 10 t.u. without transaction.

Remark 6 (Simplified enforcement functions for safety properties). Because of the characteristics of safety properties, the enforcement function for such a property φ can be simplified. A safety property φ is prefix closed thus $\mathrm{pref}(\varphi) = \varphi$, which implies that the two functions $\kappa_{\mathrm{pref}(\varphi)}$ and κ_φ are identical. Thus, the two first cases in the definition of $\mathrm{store}_\varphi(\sigma \cdot (t, a))$ can be simplified and distinguished according to whether $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$ or not; and the third case never happens. Moreover, since σ_c is initially empty, and the the two first cases in the definition of $\mathrm{store}_\varphi(\sigma \cdot (t, a))$ do not modify σ_c, by a simple induction on the input sequence, we observe that σ_c always remains empty. Thus, the second output parameter of function store_φ (i.e., the internal memory) can be suppressed. Additionally, in the first case, the first argument of the output can be simplified as it is always called with the last read event (t, a) (see below).

$t \in [0, 2)$	$\text{obs}(\sigma_3, t) = \epsilon$ $\text{store}_{\varphi_3}(\text{obs}(\sigma_3, t)) = (\epsilon, \epsilon)$ $\text{obs}(E_{\varphi_3}(\text{obs}(\sigma_3, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [2, 3)$	$\text{obs}(\sigma_3, t) = (2, op_1)$ $\text{store}_{\varphi_3}(\text{obs}(\sigma_3, t)) = (\epsilon, (2, op_1))$ $\text{obs}(E_{\varphi_3}(\text{obs}(\sigma_3, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [3, 3.5)$	$\text{obs}(\sigma_3, t) = (2, op_1) \cdot (3, op_1)$ $\text{store}_{\varphi_3}(\text{obs}(\sigma_3, t)) = (\epsilon, (2, op_1))$ $\text{obs}(E_{\varphi_3}(\text{obs}(\sigma_3, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [3.5, 6)$	$\text{obs}(\sigma_3, t) = (2, op_1) \cdot (3, op_1) \cdot (3.5, op)$ $\text{store}_{\varphi_3}(\text{obs}(\sigma_3, t)) = (\epsilon, (2, op_1) \cdot (3.5, op))$ $\text{obs}(E_{\varphi_3}(\text{obs}(\sigma_3, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [6, \infty)$	$\text{obs}(\sigma_3, t) = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$ $\text{store}_{\varphi_3}(\text{obs}(\sigma_3, t)) = ((6, op_1) \cdot (8, op) \cdot (10, op_2), \epsilon)$ $\text{obs}(E_{\varphi_3}(\text{obs}(\sigma_3, t)), t) = \text{obs}((6, op_1) \cdot (8, op) \cdot (10, op_2), t)$

Figure 9: Evolution over time of the values of the enforcement function for property φ_3 specifying the transactional execution of actions op_1 and op_2.

$t \in [0, 3)$	$\text{obs}(\sigma_4, t) = \epsilon$ $\text{store}_{\varphi_4}(\text{obs}(\sigma_4, t)) = (\epsilon, \epsilon)$ $\text{obs}(E_{\varphi_4}(\text{obs}(\sigma_4, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [3, 7)$	$\text{obs}(\sigma_4, t) = (3, acq_i)$ $\text{store}_{\varphi_4}(\text{obs}(\sigma_4, t)) = (\epsilon, (3, acq_i))$ $\text{obs}(E_{\varphi_4}(\text{obs}(\sigma_4, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [7, 12)$	$\text{obs}(\sigma_4, t) = (3, acq_i) \cdot (7, op_i)$ $\text{store}_{\varphi_4}(\text{obs}(\sigma_4, t)) = (\epsilon, (3, acq_i) \cdot (7, op_i))$ $\text{obs}(E_{\varphi_4}(\text{obs}(\sigma_4, t)), t) = \text{obs}(\epsilon, t) = \epsilon$
$t \in [12, \infty)$	$\text{obs}(\sigma_4, t) = (3, acq_i) \cdot (7, op_i) \cdot (12, rel_i)$ $\text{store}_{\varphi_4}(\text{obs}(\sigma_4, t)) = (\epsilon, (3, acq_i) \cdot (7, op_i))$ $\text{obs}(E_{\varphi_4}(\text{obs}(\sigma_4, t)), t) = \text{obs}(\epsilon, t) = \epsilon$

Figure 10: Evolution over time of the values of the enforcement function for property φ_4 (a non-enforceable property).

The enforcement function for safety properties $\text{store}^{\text{sa}}_{\varphi} : \text{tw}(\Sigma) \to \text{tw}(\Sigma)$ can be defined as follows:

$$\begin{array}{rcl} \text{store}^{\text{sa}}_{\varphi}(\epsilon) & = & \epsilon \\ \text{store}^{\text{sa}}_{\varphi}(\sigma \cdot (t, a)) & = & \begin{cases} \text{store}^{\text{sa}}_{\varphi}(\sigma) \cdot (\min(K(\sigma, (t, a))), a) & \text{if } K(\sigma, (t, a)) \neq \emptyset, \\ \text{store}^{\text{sa}}_{\varphi}(\sigma) & \text{otherwise,} \end{cases} \end{array}$$

where $K(\sigma, (t, a)) \stackrel{\text{def}}{=} \{t' \in \mathbb{R}_{\geq 0} \mid t' \geq t \wedge \text{store}^{\text{sa}}_{\varphi}(\sigma) \cdot (t', a) \lhd_d \sigma \cdot (t, a) \wedge \text{store}^{\text{sa}}_{\varphi}(\sigma) \cdot (t', a) \in \varphi\}$ is the set of dates $t' \geq t$ that can be associated to a such that the extension $\text{store}^{\text{sa}}_{\varphi}(\sigma) \cdot (t', a)$ of $\text{store}^{\text{sa}}_{\varphi}(\sigma)$ is a delayed subsequence of $\sigma \cdot (t, a)$ and still satisfies property φ.

7. Enforcement monitors: operational description of enforcement mechanisms

The enforcement function defined in Section 6 describes inductively how an input stream of events is transformed according to a property. It provides a functional view of enforcement mechanisms and

could be implemented using functional programming constructs such as recursion and lazy evaluation.

However, a concern is that the computation of dates upon the reception of a new event also depends on the sequence of events σ_s that have been already corrected, through functions κ_φ and $\kappa_{\mathrm{pref}(\varphi)}$. Consequently, implementing directly an enforcement function would require the enforcement mechanism to store σ_s in its memory, that would grow over time and never be emptied.

Instead, we implement an enforcement function E_φ for a property φ specified by a TA $\mathcal{A}_\varphi$ with an enforcement monitor (EM). An EM has an operational semantics: it is defined as a transition system $\mathcal{E}$, and has explicit state information. It keeps track of and uses information such as time elapsed, and the state reached after reading (or simulating) σ_s in the underlying TA to release the actions stored in σ_s at appropriate dates. Hence, an EM does not need to store σ_s.

7.1. Preliminaries to the definition of enforcement monitors

In contrast with an enforcement function which, at an abstract level, takes a timed word as input and produces a timed word as output, an enforcement monitor $\mathcal{E}$ also needs to take into account physical time (i.e., the actual date t), the current observation $\mathrm{obs}(\sigma, t)$ of the input stream σ at date t, the release of events to the environment which is $\mathrm{obs}(E_\varphi(\sigma), t)$, and the residual of $E_\varphi(\mathrm{obs}(\sigma, t))$ after releasing $\mathrm{obs}(E_\varphi(\sigma), t)$. Note, since storing these sequences would be impractical at runtime, an enforcement monitor encodes equivalent information as described below.

An EM $\mathcal{E}$ is equipped with: a clock which keeps track of the current date t; two memories and a set of enforcement operations used to store and release some timed events to and from the memories, respectively. The memories are basically queues, each of them containing a timed word:

- σ_{mc} manages the input queue, more precisely the subsequence of the input $\mathrm{obs}(\sigma, t)$ consisting of non-suppressed events for which dates could not yet be chosen to satisfy the property. This exactly corresponds to the timed word σ_c in function store_φ (see Definition 8);
- σ_{ms} is the output queue which manages the part of the output $E_\varphi(\sigma)$ which is computed at date t but not yet released; since at date t only prefix $\mathrm{obs}(\sigma, t)$ has been observed, and $\mathrm{obs}(E_\varphi(\sigma), t)$ has already been released, σ_{ms} contains the residual $\mathrm{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\mathrm{obs}(\sigma, t))$, i.e., the timed word that is ready to be released but not yet released.

An EM also keeps track of the current state q of the underlying LTS of the TA $\mathcal{A}_\varphi$ that encodes φ and the date t_F at which q is reached. The current state q is the one reached after reading the timed word $E_\varphi(\mathrm{obs}(\sigma, t))$ (that also corresponds to σ_s in the definition of E_φ), which is the output that can be computed from the current observation $\mathrm{obs}(\sigma, t)$. By definition q is either q_0 or a state in Q_F. The date t_F is the date $\mathrm{end}(E_\varphi(\mathrm{obs}(\sigma, t)))$ (and evaluates to 0 if $E_\varphi(\mathrm{obs}(\sigma, t)) = \epsilon$).

7.2. Function update

Before defining enforcement monitors, we introduce function update which takes as input the current state $q \in Q_F \cup \{q_0\}$ of $[\![\mathcal{A}_\varphi]\!]$[5] reached after reading $E_\varphi(\mathrm{obs}(\sigma, t))$, the arrival date t_F in this state q, a timed word $\sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma)$ that has to be corrected, and the last received event (t, a). Function update possibly updates the current state, and outputs a marker used by $\mathcal{E}$ to make decisions, according to whether σ_{mc} can be corrected or not, and in the negative case, whether an extension could be.

[5]The partitioning of the states Q of $[\![\mathcal{A}]\!]$ into four subsets G, G^C, B^C and B is defined in Section 4.2.

Definition 9 (Function update**).** update is a function from $Q \times \mathbb{R}_{\geq 0} \times \mathrm{tw}(\Sigma) \times (\mathbb{R}_{\geq 0} \times \Sigma)$ to $Q \times \mathrm{tw}(\Sigma) \times \{\texttt{ok}, \texttt{c\_bad}, \texttt{bad}\}$ defined as follows:

$$\mathrm{update}(q, t_F, \sigma_{\mathrm{mc}}, (t,a)) \stackrel{\text{def}}{=} \begin{cases} \big(q', w_{\min}, \texttt{ok}\big) & \text{if } k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a)) \neq \emptyset \wedge q \stackrel{w_{\min}}{\rightarrow}_{t_F} q', \\ (q, \sigma_{\mathrm{mc}}, \texttt{bad}) & \text{if } k_{Q_F \cup B^C}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a)) = \emptyset, \\ (q, \sigma_{\mathrm{mc}} \cdot (t,a), \texttt{c\_bad}) & \text{otherwise,} \end{cases}$$

where, for $\mathcal{Q} \subseteq Q$,

$$k_{\mathcal{Q}}(q, t_F, \sigma) = \Big\{ w \in \mathrm{tw}(\Sigma) \mid q \stackrel{w}{\rightarrow}_{t_F} \mathcal{Q} \Big\} \cap \mathrm{CanD}(\sigma),$$

and $w_{\min} = \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a))$.

Recall that $\mathrm{CanD}(\sigma)$ (defined in section 6.2) computes the set of timed words that delay σ and start at or after $\mathrm{end}(\sigma)$. Function $k_{\mathcal{Q}}$ explicitly uses the semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ of the TA $\mathcal{A}_\varphi$ defining property φ, and, using function CanD, mimics the computation of the sets $\kappa_\varphi(\sigma_s, \sigma'_c)$, and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma'_c)$ defined in section 6.2. Function $k_{\mathcal{Q}}$ is parameterised with a set of states $\mathcal{Q} \subseteq Q$ and called with three parameters: the current state q, a date t_F, and a sequence σ. It returns the set of timed words leading to a state in $\mathcal{Q}$ from state q starting at date t_F, among sequences in $\mathrm{CanD}(\sigma)$.

The three cases in the definition of update encode the three cases in the definition of function store_φ, in the same order:

- In the first case, $\mathcal{Q} = Q_F$ and $k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a))$ is not empty, i.e., appropriate delaying dates can be chosen for the events in $\sigma_{\mathrm{mc}} \cdot (t,a)$ such that an accepting state $q' \in Q_F$ is reached from q, starting at date t_F. In this case, function update returns q', $w_{\min}$, and marker ok: $w_{\min}$ is the minimal word w.r.t. the lexical order among those timed words of minimal ending date in $k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a))$, $q' \in Q_F$ is the state reached from q with $w_{\min}$, and marker ok indicates that Q_F is reached.
- In the second case, $\mathcal{Q} = Q_F \cup B^C$ and $k_{Q_F \cup B^C}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t,a))$ is empty; it is thus impossible to correct $\sigma_{\mathrm{mc}} \cdot (t,a)$ in the future, since no candidate sequence delaying $\sigma_{\mathrm{mc}} \cdot (t,a)$ leads to a state in $Q_F \cup B^C$, i.e., accepting states or states from which a path leads to an accepting state (they all lead to bad states B). This reflects the fact that $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t,a))$ is empty, since the set of accepting states of $\mathrm{pref}(\varphi)$ is $Q_F \cup B^C$. In this case, function update returns state q and timed word σ_{mc} unmodified, indicating that event (t,a) is suppressed, and marker bad indicates that no accepting state could be reached in the future if (t,a) was retained in memory.
- In the third case, function update returns state q unmodified, but returns the timed word $\sigma_{\mathrm{mc}} \cdot (t,a)$, and a marker c_bad. The marker indicates that $\sigma_{\mathrm{mc}} \cdot (t,a)$ can not be delayed to reach an accepting state, but there it is still possible to reach a new accepting state after observing more events in the future.

On the computation of function update. Function update can be computed using operations on TAs and known algorithms solving classical problems, with the help of the standard symbolic representation of behaviours of TAs by region or zone graphs, and refinement of these, using Difference Bound Matrices (DBM) to encode timing constraints. However, one needs to adapt TAs following practical considerations as explained below. We first introduce the sub-problems involved in the computation of update and references to their algorithmic solutions. Next, we explain some considerations to extend the kind of TAs handled by these algorithms. Finally, we explain how to encode the computation of update into these algorithms and standard operations on TAs.

Reachability problem: For a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, check whether F is reachable. Recall that reachability is PSPACE-complete in the size of the TA $\mathcal{A}$ [11] and can be solved using the symbolic region or zone representations and forward or backward analysis, e.g., using UPPAAL [14].

Optimal reachability problem: For a TA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, check whether F is reachable and if yes, find a run with minimal duration. It can be proven that this problem is also PSPACE-complete in the size of $\mathcal{A}$. PSPACE-hardness is a direct consequence of the fact that reachability in TAs is already PSPACE-hard. PSPACE-easiness is a consequence of the fact that a more general problem, the optimal cost reachability problem for weighted timed automata (WTAs), is proven to be PSPACE-complete in [19], and can be solved by the exploration of the *weighted directed graph*. The weighted directed graph is a refinement of the region graph in which the durations of time transitions are arbitrarily close to integers, and edges are augmented with cost functions which are polynomials in the constants of $\mathcal{A}$. Cost-optimal paths can be found among those where the durations spent in locations are arbitrarily close to integers. Moreover, in the case of TAs with only non-strict guards, the optimal timed words indeed have integral dates.

We now state four considerations that allow to apply these algorithms in our context:

Consideration 1 In a real runtime environment, dates of input events are observed by a digital clock with limited precision. Observed dates can thus be considered as rationals, more precisely integral multiples of a sampling rate $1/D$ of a clock, rather than reals as in the idealized model of timed words. As a consequence of this, of the computation of update and its use in the enforcement monitor, the computed output dates are also rationals (obtained by reverse scaling of integer dates obtained by optimal reachability, see below).

Consideration 2 In our definition of TAs, all constants in guards are integers. These TAs are sometimes called *integral* TAs. As will be clear later, and in particular because of Consideration 1, we shall also consider *rational* TAs, i.e., TAs where constants can be rational. A rational TA can be transformed into an integral TA by considering $1/d$ as the new unit value, where d is the least common multiple of all denominators of rational constants. Note that the value $1/d$, which will be useful in the sequel, will always be a multiple of the observation sampling rate $1/D$, thus one can simply take $1/D$. Since the size of the regions/zones graph depends on the maximal constant, there is a tradeoff between the precision of the observation and the cost of reachability analysis.

Consideration 3 Still due to Consideration 1, in the use of update we will have to solve (optimal) reachability not only from the initial state, but from some state q where the location l may differ from l_0 and clocks have a rational valuation ν. First, as is the case with Consideration 2, one can scale this TA by multiplying all constants by the least common multiple of denominators of this valuation (and constants if rational) in order to get an integral TA starting in an integral valuation ν'. Second, the construction of the region/zone automaton will be as usual, except that it should start in (l, ν').

Consideration 4 As seen above with optimal reachability, for TAs with strict (lower) guards, infimum may not be realizable even though reachability is achieved. However, a timed word arbitrarily close to the infimum exists as soon as reachability is achieved. Alternatively, one may approximate the TA with a TA exhibiting only non-strict guards. Since output dates should be multiple of the sampling rate $1/D$, the approximation consists in transforming all strict guards of the form $x > c$, where c is an integer, into guards of the form $x \geq c + 1/D$, and then use Consideration 2 to transform this rational TA into an integral TA, and get guards of the form $x \geq D * c + 1$.

Remark 7. Most of the above considerations concern rational constants (or initial rational valuations) in TAs and have their roots in the precision of the observation (Consideration 1). An alternative way to understand those considerations, and avoid problems of rational constants in all the algorithms, is simply to consider the observation sampling $1/D$ as the new time unit, and thus to observe events at

integral dates and rescale TA $\mathcal{A}_\varphi$ according to this new time unit. As a consequence, all TAs that have to be built would be integral TAs.

We now come back to the computation of function update. First, $\mathrm{CanD}(\sigma_{\mathrm{mc}} \cdot (t, a))$ can be represented by a rational TA C with a new clock $y \notin X$ (that does not belong to the set of clocks of the TA $\mathcal{A}_\varphi$ of the property) initialized to 0, and $|\sigma_{\mathrm{mc}} \cdot (t, a)|$ transitions in sequence, one transition per action in $\sigma_{\mathrm{mc}} \cdot (t, a)$, the first transition with constraint $y \geq t$, the other ones with no timing constraint, no reset on any transition, and one accepting location in set F_C at the end, with no outgoing transition. Clearly, this automaton recognises timed words delaying $\sigma_{\mathrm{mc}} \cdot (t, a)$ and starting after t. Since t is the date at which a is observed, by Consideration 1 we suppose that it is a rational, multiple of the observation sampling $1/D$, thus C is a rational TA. For technical reasons, in the following we will rather consider the rational TA C' obtained from C by replacing $y \geq t$ by $y \geq t - t_F$ in the first transition, where t_F is the arrival date in state q (note that it can be easily proven by induction on the use of update that t_F is rational, since computed output dates are rational). C' recognizes the same timed words as C, with all dates decreased by the duration t_F.

For the first case in the definition of update, one needs first to check whether $k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t, a)) \neq \emptyset$ and then to pick a timed word with minimal duration in this set. This can thus be done as follows: let $\mathcal{A}_\varphi(q)$ be the same TA as $\mathcal{A}_\varphi$, but starting in the initial state q, where q is a pair (l, ν) with $l \in L$ and ν a rational valuation of the clocks in X. Now build the product TA $\mathcal{A}_\varphi(q) \times C'$ and check whether $F \times F_C$ is reachable. For this purpose, Consideration 4 is used to transform strict guards into non-strict ones, and then Considerations 2 and 3 are used to transform this rational TA initialized with a rational valuation into an integral TA initialized with an integral valuation. If $F \times F_C$ is reachable, computing the timed word with minimum duration can be done using the algorithm described in [19], resulting in an integral timed word. Next, one has to rescale this integral timed word into a rational timed word by division by the scalings used to transform rational TAs to integral TAs. Finally, the resulting timed word is increased by the duration t_F to get the final result.

For the second case, one needs to check whether $k_{Q_F \cup B^C}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t, a)) = \emptyset$. This can be done as follows: let C'' be the same automaton as C', except that the accepting locations in F_C loop on any action; C'' then recognises extensions of timed words in $\mathrm{CanD}(\sigma_{\mathrm{mc}} \cdot (t, a))$, but again decreased by the duration t_F; build the product $\mathcal{A}_\varphi(q) \times C''$, and check whether $F \times F_C$ is reachable in this TA, using Considerations 2 and 3 again to transform this rational TA initialized with a rational valuation into an integral TA initialized with an integral valuation. If the answer is no, $k_{Q_F \cup B^C}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t, a)) = \emptyset$.

An operational definition of function update as an algorithm is described in Section 8.

Complexity. Recall that for an integral timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, reachability can be solved by first constructing the region graph or zone graph which size is in $\mathcal{O}((|\Delta| + |L|).(2M + 2)^{|X|}.|X|!.2^{|X|})$, where M is the maximal constant appearing in guards, $|L|$ is the number of locations, $|\Delta|$ the number of edges, and $|X|$ the number of clocks, and solving reachability in this finite graph. Since reachability in finite graphs is NLOGSPACE-complete (in the size of the finite graph), globally, this algorithm is in PSPACE, and it is proven that the problem is PSPACE-complete [11].

As previously mentioned, optimal reachability is PSPACE-complete in the size of $\mathcal{A}$. It is a consequence of the PSPACE-completeness of the more general problem of optimal reachability for weighted time automata. Weighted timed automata are extensions of timed automata where a cost function $\mathcal{C}$ assigns integer costs to both locations and transitions, with the semantics that firing a transition e induces a cost $\mathcal{C}(e)$, and spending τ time units in a location l induces a cost $\mathcal{C}(l).\tau$. The optimal reachability problem for TAs can thus be reduced to the cost optimal reachability problem for WTAs in which a null cost is assigned to transitions and a cost of 1 is assigned to every location. The optimal reachability problem is solved by first constructing the weighted directed graph, which refines the region graph by

focusing on what happens close to integral corners of regions, and labelling transitions with a cost function. The size of the weighted directed graph is $|X|+1$ times bigger than the region graph. Optimality is then solved by traversing on-the-fly this graph and comparing the weights of elementary paths.

For simplicity, the complexity of both algorithms are in general abstracted to $\mathcal{O}(2^{|\mathcal{A}|})$ where $|\mathcal{A}|$ takes into account the number of transitions, locations, the maximal constant, and the number of clocks in $\mathcal{A}$.

Note that to solve those problems for a rational TA, one first needs to build an integral TA according to the observation sampling $1/D$, by multiplying constants by D. The size of the region graph thus becomes $\mathcal{O}((|\Delta|+|L|).(2.D.M+2)^{|X|}.|X|!.2^{|X|})$, and both problems are still in $\mathcal{O}(2^{|\mathcal{A}|})$. For the product of two TAs $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, and $\mathcal{B} = (L', l'_0, X', \Sigma, \Delta', F')$, with respective maximal constants M and M', the size of the region graph thus becomes $\mathcal{O}((|\Delta|.|\Delta'|+|L|.|L'|).(2.\max(M, M')+2)^{|X|+|X'|}.(|X|+|X'|)!.2^{|X|+|X'|})$, and the complexity of (optimal) reachability becomes $\mathcal{O}(2^{|\mathcal{A}|+|\mathcal{B}|})$.

Now, let us come to the complexity of update, or more precisely to the orders of sizes of the region graphs and weighted directed graphs that need to be traversed, since these are the key elements in the complexity of the algorithms. For a given input memory $\sigma_{\mathrm{mc}} \cdot (t, a)$, the computation of update requires to solve the optimal reachability problem on the automaton $\mathcal{A}_\varphi(q) \times C'$ and the reachability problem on the automaton $\mathcal{A}_\varphi(q) \times C''$ where C' and C'' are built from $\sigma_{\mathrm{mc}} \cdot (t, a)$ as explained above, and $\mathcal{A}_\varphi(q)$ is obtained from a TA $\mathcal{A}_\varphi = (L, l_0, X, \Sigma, \Delta, F)$ (with maximal constant M) by shifting the initial state to q. Firstly, note that the automaton $\mathcal{A}_\varphi(q)$ is of same size as $\mathcal{A}_\varphi$, but is a rational TA, thus the maximal constant of the corresponding integral automaton is $M.D$ when scaling to integral automata with observation sampling $1/D$. Secondly, the automata C' and C'' both have $\mathcal{O}(|\sigma_{\mathrm{mc}}|)$ locations and respectively $\mathcal{O}(|\sigma_{\mathrm{mc}}|)$ and $\mathcal{O}(|\sigma_{\mathrm{mc}}| + |\Sigma|)$ transitions. They both have one clock and the maximal constant of their corresponding integral TAs is the integer $D.(t - t_F)$.

For the product TAs $\mathcal{A}_\varphi(q) \times C'$ and $\mathcal{A}_\varphi(q) \times C''$, we get $\mathcal{O}(|\sigma_{\mathrm{mc}}|.|L|)$ locations and respectively $\mathcal{O}(|\sigma_{\mathrm{mc}}|.|\Delta|)$ and $\mathcal{O}((|\sigma_{\mathrm{mc}}| + |\Sigma|).|\Delta|)$ transitions. The maximal constant is $D.\max(M, t - t_F)$ and both automata have $|X| + 1$ clocks.

In the first case of function update, solving the optimal reachability problem in the TA $\mathcal{A}_\varphi(q) \times C'$ induces the (partial) construction and traversal of a weighted directed graph which is of size $\mathcal{O}((|X| + 2).|\sigma_{\mathrm{mc}}|.(|\Delta| + |L|).(2.D.\max(M, t - t_F) + 2)^{|X|+1}.(|X| + 1)!.2^{|X|+1})$.

In the second case, the reachability problem in the TA $\mathcal{A}_\varphi(q) \times C''$ can be solved by building a region graph of size $\mathcal{O}(((|\sigma_{\mathrm{mc}}|+|\Sigma|).|\Delta|)+(|\sigma_{\mathrm{mc}}|.|L|)).(2.D.\max(M, t-t_F)+2)^{|X|+1}.(|X|+1)!.2^{|X|+1})$.

In spite of these complexities, these problems can be efficiently solved, e.g., in UPPAAL [14], using zones and their encoding with DBMs. One key to efficiency is the choice of the right observation sampling $1/D$ which influences the size of the maximal constant in integral TAs. The smaller is $1/D$, the tighter is the observation, but the larger is the region graph. It should also be noted that even though the maximal size of the product automata is in the product of sizes of component automata, in practice only paths in $\mathcal{A}_\varphi$ along the untimed projection of $\sigma_{\mathrm{mc}} \cdot (t, a)$ have to be considered, which strongly restricts the region graph or weighted directed graph that need to be built when searching for (optimal) accepted timed words. Finally, as will be clear later, update is called with sequences $\sigma_{\mathrm{mc}} \cdot (t, a)$ of increasing length (in the case where no suppression occurs), starting from 1, until it can be corrected to satisfy φ, in which case its length is reinitialized to 1. The worst case complexity is reached when no prefix can be corrected (but a possible extension always could) before the arrival of the sequence. But in general, we may expect that φ can be regularly satisfied by correcting the input.

7.3. Definition of enforcement monitors

We can now define the enforcement monitor using function update defined in Section 7.2.

Definition 10 (Enforcement Monitor). Let us consider a regular property φ recognised by the TA $\mathcal{A}_\varphi$ with semantics $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Gamma, \to, Q_F)$. The enforcement monitor for φ is the transition system $\mathcal{E}_\varphi = (C^{\mathcal{E}_\varphi}, c_0^{\mathcal{E}_\varphi}, \Gamma^{\mathcal{E}_\varphi}, \hookrightarrow_{\mathcal{E}_\varphi})$ s.t.:

- $C^{\mathcal{E}_\varphi} = \mathrm{tw}(\Sigma) \times \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \times Q \times \mathbb{R}_{\geq 0}$ is the set of configurations of the form $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$, where σ_{ms}, σ_{mc} are timed words to memorise events, t is a positive real number to keep track of time, q is a state in the semantics of the TA and t_F keeps track of the arrival date in q,
- $c_0^{\mathcal{E}_\varphi} = (\epsilon, \epsilon, 0, q_0, 0) \in C^{\mathcal{E}_\varphi}$ is the initial configuration,
- $\Gamma^{\mathcal{E}_\varphi} = ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}) \times Op \times ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})$ is the alphabet, i.e., the set of triples comprised of an optional input event, an operation, and an optional output event, where the set of possible operations is $Op = \{\text{store-}\overline{\varphi}(\cdot), \text{store}_{\sup}\text{-}\overline{\varphi}(\cdot), \text{store-}\varphi(\cdot), \text{release}(\cdot), \text{idle}(\cdot)\}$;
- $\hookrightarrow_{\mathcal{E}_\varphi} \subseteq C^{\mathcal{E}_\varphi} \times \Gamma^{\mathcal{E}_\varphi} \times C^{\mathcal{E}_\varphi}$ is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:
  - **1. *store-*φ**:
    $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \overset{(t,a)/\text{store}-\varphi(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\mathrm{ms}} \cdot w, \epsilon, t, q', \mathrm{end}(w))$, if $\mathrm{update}(q, t_F, \sigma_{\mathrm{mc}}, (t,a))) = (q', w, \texttt{ok})$,
  - **2. *store*$_{\sup}$-$\overline{\varphi}$**:
    $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \overset{(t,a)/\text{store}_{\sup}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$, if $\mathrm{update}(q, t_F, \sigma_{\mathrm{mc}}, (t,a)) = (q, \sigma_{\mathrm{mc}}, \texttt{bad})$,
  - **3. *store-*$\overline{\varphi}$**:
    $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \overset{(t,a)/\text{store}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \cdot (t,a), t, q, t_F)$, if $\mathrm{update}(q, t_F, \sigma_{\mathrm{mc}}, (t,a)) = (q, \sigma_{\mathrm{mc}} \cdot (t,a), \texttt{c\_bad})$,
  - **4. *release***:
    $((t,a) \cdot \sigma'_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \overset{\epsilon/\text{release}(t,a)/(t,a)}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma'_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$,
  - **5. *idle***:
    $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \overset{\epsilon/\text{idle}(\delta)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t+\delta, q, t_F)$ if $\delta \in \mathbb{R}_{>0}$ is a delay such that, for all $\delta' < \delta$, no other rule can be applied to $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t+\delta', q, t_F)$[6],

  where $c \overset{e/\text{op}(p)/e'}{\hookrightarrow_{\mathcal{E}_\varphi}} c'$ denotes the fact that the enforcement monitor moves from configuration c to configuration c' by reading e, executing operation op parameterised by p, and outputting e'.

A configuration $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$ of the EM consists of the following elements: σ_{ms} is the sequence which is corrected and can be released as output; σ_{mc} is the input sequence read by the EM, but yet to be corrected, except for events that are suppressed; t indicates the current date; q is the current state of $[\![\mathcal{A}_\varphi]\!]$ reached after processing the sequence already released, followed by the timed word in memory σ_{ms}, i.e., $E_\varphi(\mathrm{obs}(\sigma, t))$; t_F is the arrival date in q.

Semantic rules can be understood as follows:

- Upon the reception of an event (t,a) (i.e., when t is the date in the configuration and (t,a) is read), one of the following rules is executed. Notice that their conditions are exclusive of each others.
  - **1.** Rule ***store-*φ** is executed if function update returns state $q' \in Q_F$, timed word w and marker ok, indicating that φ can be satisfied by the sequence already released as output,

[6]The allowed delays are obviously not known in the starting configuration of rule ***idle***. In practice, at the implementation level, allowed delays are determined using busy-waiting.

followed by σ_{ms}, and followed by w which minimally delays $\sigma_{\mathrm{mc}} \cdot (t, a)$ to satisfy φ. When executing the rule, sequence w is appended to the content of output memory σ_{ms}, the input memory σ_{mc} is emptied, q' is the new state and $\mathrm{end}(w)$ is the new arrival date.

  - **2.** Rule ***store***$_{\mathrm{sup}}$-$\overline{\varphi}$ is executed if the update function returns marker `bad`, indicating that $\sigma_{\mathrm{mc}} \cdot (t, a)$ followed by any sequence cannot be corrected. Event (t, a) is then suppressed, and the configuration remains unchanged.
  - **3.** Rule ***store***-$\overline{\varphi}$ is executed if the update function returns marker `c_bad`, indicating that $\sigma_{\mathrm{mc}} \cdot (t, a)$ cannot be corrected yet. The event (t, a) is then appended to the internal memory σ_{mc}, but σ_{ms}, q and t_F remain unchanged.

- When no event can be received, one of the following rules is applied, with decreasing priority:
  - **4.** Rule ***release*** is executed if the current date t is equal to the date corresponding to the first event of the timed word $\sigma_{\mathrm{ms}} = (t, a) \cdot \sigma'_{\mathrm{ms}}$ in the memory. The event is released as output and removed from σ_{ms} in the resulting configuration.
  - **5.** Rule ***idle*** adds the time elapsed δ to the current value of t when neither store nor release operations are possible at any time instant between t and $t + \delta$.

Note, all rules except rule ***idle*** execute in zero time. Moreover, it is important to notice that the definition of update entails that the state q inside a configuration is either initial (initially $q = q_0$) or accepting (it is only modified by a ***store***-φ rule which makes it jump to a state $q \in Q_F$ as a result of update), one case not excluding the other (e.g., for safety properties).

Example 7 (Execution of an enforcement monitor). We illustrate how the rules of Definition 10 are applied to enforce property φ_3 (see Section 2), recognised by the automaton depicted in Figure 6c with $\Sigma_3 = \{op_1, op_2, op\}$, and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Figure 11 shows how semantic rules are applied according to the current date t, and the evolution of the configurations of the enforcement monitor, together with input and output. More precisely, each line is of the form $O/c/I$, where O is the sequence of released events, c is a configuration, and I is the residual of the input σ after its observation at date t. The resulting (final) output is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$, which satisfies property φ_3. Note that after $t = 10$, only rule ***idle*** can be applied.

7.4. Relating Enforcement functions and enforcement monitors

We show how the definitions of enforcement function and enforcement monitor are related: given a property φ, any input sequence σ, at any date t, the output of the associated enforcement function and the output behaviour of the associated enforcement monitor are equal.

Preliminaries. We first describe how an enforcement monitor reacts to an input sequence. In the remainder of this section, we consider an enforcement monitor $\mathcal{E} = (C^{\mathcal{E}}, c_0^{\mathcal{E}}, \Gamma^{\mathcal{E}}, \hookrightarrow_{\mathcal{E}})$, not related to a property. Enforcement monitors, described in Section 7, are deterministic. By determinism, we mean that, given an input sequence, the observable output sequence is unique. Moreover, given $\sigma \in \mathrm{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, how an enforcement monitor reads σ until date t is unique: it goes through a unique sequence of configurations. Since rule ***idle*** does not read nor produce any event, ϵ belongs to the input alphabet. Thus, given an input sequence σ and a date t, there is possibly an infinite set of corresponding sequences over the *input-operation-output* alphabet (as in Definition 10). All these sequences are equivalent: they involve the same configurations for the enforcement monitor and the same output sequence. Consequently, the rules of transition relations are ordered in such a way that reading ϵ will always be

$$
\begin{array}{ll}
t = 0 & \epsilon/(\epsilon, \epsilon, 0, (l_0, 0, 0), 0)/(2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow \epsilon/\text{idle}(2)/\epsilon \\
t = 2 & \epsilon/(\epsilon, \epsilon, 2, (l_0, 0, 0), 0)/(2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow (2, op_1)/\text{store-}\overline{\varphi}(2, op_1)/\epsilon \\
t = 2 & \epsilon/(\epsilon, (2, op_1), 2, (l_0, 0, 0), 0)/(3, op_1) \cdot (3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow \epsilon/\text{idle}(1)/\epsilon \\
t = 3 & \epsilon/(\epsilon, (2, op_1), 3, (l_0, 0, 0), 0)/(3, op_1) \cdot (3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow (3, op_1)/\text{store}_{\text{sup}}\text{-}\overline{\varphi}(3, op_1)/\epsilon \\
t = 3 & \epsilon/(\epsilon, (2, op_1), 3, (l_0, 0, 0), 0)/(3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow \epsilon/\text{idle}(0.5)/\epsilon \\
t = 3.5 & \epsilon/(\epsilon, (2, op_1), 3.5, (l_0, 0, 0), 0)/(3.5, op) \cdot (6, op_2) \\
 & \quad \downarrow (3.5, op)/\text{store-}\overline{\varphi}(3.5, op)/\epsilon \\
t = 3.5 & \epsilon/(\epsilon, (2, op_1) \cdot (3.5, op), 3.5, (l_0, 0, 0), 0)/(6, op_2) \\
 & \quad \downarrow \epsilon/\text{idle}(2.5)/\epsilon \\
t = 6 & \epsilon/(\epsilon, (2, op_1) \cdot (3.5, op), 6, (l_0, 0, 0), 0)/(6, op_2) \\
 & \quad \downarrow (6, op_2)/\text{store-}\varphi(6, op_2)/\epsilon \\
t = 6 & \epsilon/((6, op_1) \cdot (8, op) \cdot (10, op_2), \epsilon, 6, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{release}(6, op_1)/(6, op_1) \\
t = 6 & (6, op_1)/((8, op) \cdot (10, op_2), \epsilon, 6, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{idle}(2)/\epsilon \\
t = 8 & (6, op_1)/((8, op) \cdot (10, op_2), \epsilon, 8, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{release}(8, op)/(8, op) \\
t = 8 & (6, op_1) \cdot (8, op)/((10, op_2), \epsilon, 8, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{idle}(2)/\epsilon \\
t = 10 & (6, op_1) \cdot (8, op)/((10, op_2), \epsilon, 10, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{release}(10, op_2)/(10, op_2) \\
t = 10 & (6, op_1) \cdot (8, op) \cdot (10, op_2)/(\epsilon, \epsilon, 10, (l_0, 4, 2), 10)/\epsilon \\
 & \quad \downarrow \epsilon/\text{idle}(t)/\epsilon \\
t > 10 & (6, op_1) \cdot (8, op) \cdot (10, op_2)/(\epsilon, \epsilon, t, (l_0, 4, 2), 10)/\epsilon
\end{array}
$$

Figure 11: Execution of an enforcement monitor for φ_3. The enforcement monitor ensures that if the automaton for φ_3 reads the (entire) output sequence, it remains in its accepting states.

the transition with least priority. Consequently, given an input sequence, reading ϵ (and doing other operations such as outputting some event) is always possible when the monitor cannot read an input.

More formally, let us define $\mathcal{E}^{\mathrm{ioo}}(\sigma,t) \in (\Gamma^{\mathcal{E}})^*$ to be the unique sequence of transitions (triples comprised of an optional input event, an operation, and an optional output event) that is "triggered" from the initial configuration, when the enforcement monitor reads σ until date t:

Definition 11 (Input-Operation-Output sequence). Given an input sequence $\sigma \in \mathrm{tw}(\Sigma)$ and some date $t \in \mathbb{R}_{\geq 0}$, we define the input-operation-output sequence, denoted as $\mathcal{E}^{\mathrm{ioo}}(\sigma,t)$, as the unique[7] sequence of $(\Gamma^{\mathcal{E}})^*$ such that:

$$\begin{aligned}\exists c \in C^{\mathcal{E}} : \quad & c_0^{\mathcal{E}} \overset{\mathcal{E}^{\mathrm{ioo}}(\sigma,t)}{\hookrightarrow}{}^{*}_{\mathcal{E}}\ c \\ \wedge \quad & \Pi_1(\mathcal{E}^{\mathrm{ioo}}(\sigma,t)) = \mathrm{obs}(\sigma,t) \\ \wedge \quad & \mathrm{timeop}(\Pi_2(\mathcal{E}^{\mathrm{ioo}}(\sigma,t))) = t \\ \wedge \quad & \neg\left(\exists c' \in C^{\mathcal{E}}, e \in (\mathbb{R}_{\geq 0} \times \Sigma) : c \overset{(\epsilon,\mathrm{release}(e),e)}{\hookrightarrow}_{\mathcal{E}} c'\right),\end{aligned}$$

where the timeop function indicates the duration of a sequence of enforcement operations and says that only the idle enforcement operation consumes time. Formally:

$$\begin{aligned}\mathrm{timeop}(\epsilon) &= 0; \\ \mathrm{timeop}(op \cdot ops) &= \begin{cases} d + \mathrm{timeop}(ops) & \text{if } \exists d \in \mathbb{R}_{>0} : op = \mathrm{idle}(d), \\ \mathrm{timeop}(ops) & \text{otherwise.}\end{cases}\end{aligned}$$

The observation of the input timed word σ at any date t, corresponding to $\mathrm{obs}(\sigma,t)$, is the concatenation of all the input events read/consumed by the enforcement monitor over various steps. Observe that, because of the assumptions on $\Gamma^{\mathcal{E}}$, only rule ***idle*** applies to configuration c: rule ***release*** does not apply by definition of $\mathcal{E}^{\mathrm{ioo}}(\sigma,t)$ and none of the ***store*** rules applies because $\Pi_1(\mathcal{E}^{\mathrm{ioo}}(\sigma,t)) = \mathrm{obs}(\sigma,t)$.

Relating enforcement functions and enforcement monitors. We now relate the enforcement function E_φ and the enforcement monitor $\mathcal{E}_\varphi$, for a property φ, using the input-operation-output behaviour $\mathcal{E}_\varphi^{\mathrm{ioo}}$ of $\mathcal{E}_\varphi$ as per Definition 11. Seen from the outside, an enforcement monitor $\mathcal{E}_\varphi$ behaves as a device reading and producing timed words. Overloading notations, this input/output behaviour can be characterised as a function $\mathcal{E}_\varphi : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \mathrm{tw}(\Sigma)$ defined as:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathcal{E}_\varphi(\sigma,t) = \Pi_3\left(\mathcal{E}_\varphi^{\mathrm{ioo}}(\sigma,t)\right).$$

The corresponding output timed word $\mathcal{E}_\varphi(\sigma,t)$, at any date t, is the concatenation of all the output events produced by the enforcement monitor over various steps of the enforcement monitor (where all ϵ's are erased through concatenation). In the following, we do not distinguish between an enforcement monitor and the function that characterises its behaviour.

Finally, we define an implementation relation between enforcement monitors and enforcement functions as follows.

Definition 12 (Implementation relation). Given an enforcement function E_φ (as per Definition 8) and an enforcement monitor (as per Definition 10) whose behaviour is characterised by a function $\mathcal{E}_\varphi$, we say that $\mathcal{E}_\varphi$ *implements* E_φ iff:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma),t) = \mathcal{E}_\varphi(\sigma,t).$$

[7] The uniqueness of $\mathcal{E}^{\mathrm{ioo}}(\sigma,t)$ is discussed in Remark 9 in Appendix A.3.

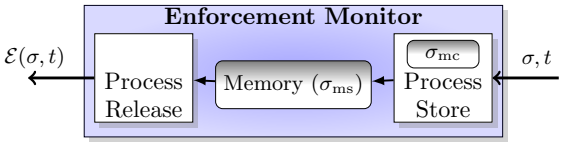


Figure 12: Realising an EM.

Proposition 3 (Relation between enforcement function and enforcement monitor) *Given a property φ, its enforcement function E_φ (as per Definition 8, p. 16), and its enforcement monitor $\mathcal{E}_\varphi$ (as per Definition 10, p. 26), $\mathcal{E}_\varphi$ implements E_φ in the sense of Definition 12.*

PROOF (OF PROPOSITION 3 - SKETCH ONLY). The proof is given in Appendix A.4, p. 53. The proof relies on an induction on the length of the input word σ. The induction step uses a case analysis, depending on whether the input is completely observed or not at date t, whether the input can be delayed into a correct output or not, and whether the memory content (σ_{ms}) is completely released or not at date t. The proof also uses several intermediate lemmas that characterise some special configurations (e.g., value of the clock variable, content of the memory σ_{ms}) of an enforcement monitor.

8. Enforcement algorithms: implementation of enforcement mechanisms

An enforcement monitor remains an abstract view of a real enforcement mechanism, and needs to be further concretised into an implementation. The implementation of an enforcement monitor consists of two processes running concurrently (called hereafter StoreProcess and ReleaseProcess) and started simultaneously, and a shared memory, as shown in Figure 12. StoreProcess implements the ***store*** rules of the enforcement monitor. The memory contains the timed word σ_{ms}: the corrected sequence that can be released as output. The memory is realised as a queue, shared between the StoreProcess and ReleaseProcess, where the StoreProcess adds events, which are processed and corrected, to this queue. ReleaseProcess reads the events stored in the memory σ_{ms} and releases the action corresponding to each event as output, when time reaches the date associated to the event. StoreProcess also makes use of another internal buffer σ_{mc} (not shared with any other process), to store the events which are read, but can not be corrected yet, to satisfy the property. In the algorithms, primitive await is used to wait for a trigger event from another process or to wait until some condition becomes true. Primitive wait is used by a process to wait for some amount of time determined by the process itself.

In the following, we first present algorithm update used by algorithm StoreProcess, then present algorithm StoreProcess, and finally algorithm ReleaseProcess.

Algorithm update *(see Algorithm 1).* Algorithm update implements function update from Definition 9. It takes as input q, the current state, t_F, the arrival date in state q, the events stored in the internal memory σ_{mc} of StoreProcess, and the new event (t, a), and returns a new state q', a timed word σ'_{mc}, and a marker in the set $\{\mathtt{ok}, \mathtt{c\_bad}, \mathtt{bad}\}$, indicating whether $\sigma_{\mathrm{mc}} \cdot (t, a)$ can be delayed to satisfy φ.

The algorithm makes use of the following functions. Function computeReach computes all the reachable paths[8] from the current state q upon events in $\sigma_{\mathrm{mc}} \cdot (t, a)$ that start after date t, where time starts at date t_F, the arrival date in q. Formally, it computes $\left\{ w \in \mathrm{tw}(\Sigma) \mid q \xrightarrow{w}_{t_F} \right\} \cap \mathrm{CanD}(\sigma_{\mathrm{mc}} \cdot (t, a))$. Function getAccPaths takes as input all the paths returned by computeReach and returns only those

[8] A path is a run in the symbolic (zone) graph.

Algorithm 1 update$(q, t_F, \sigma_{\mathrm{mc}}, (t, a))$

```
allPaths ← computeReach(σ_mc · (t, a), q, t_F)
accPaths ← getAccPaths(allPaths)
if accpaths ≠ ∅ then
    σ'_mc ← getOptimalWord(accPaths, σ_mc · (t, a))
    return(post(q, σ'_mc), σ'_mc, ok)
else
    isReachable ← checkReachAcc(allPaths)
    if isReachable = ff then
        return(q, σ_mc, bad)
    else
        return(q, σ_mc, c_bad)
    end if
end if
```

that lead to a state in Q_F. Formally it computes $k_{Q_F}(q, t_F, \sigma_{\mathrm{mc}} \cdot (t, a)) = \left\{ w \in \mathrm{tw}(\Sigma) \mid q \xrightarrow{w}_{t_F} Q_F \right\} \cap \mathrm{CanD}(\sigma_{\mathrm{mc}} \cdot (t, a))$. Both functions use forward analysis, zone abstraction, and operations on zones such as the resetting of clocks and intersection of guards [16].

Function $\mathrm{getOptimalWord}$ takes all the accepting paths and a sequence $\sigma_{\mathrm{mc}} \cdot (t, a)$ and computes optimal delays for events in $\sigma_{\mathrm{mc}} \cdot (t, a)$. This function first computes an optimal date for each event, for all accepting paths. Finally, it picks a path among the set of accepting paths whose ending date is minimal, and returns it as the result. Function $\mathrm{getOptimalWord}$ implements the computation described in Section 7.2 (§ *On the computation of function* update) using a simplified version of the algorithm in [19]. Function post takes a state of the automaton defining the property, a timed word, and computes the state reached by the automaton. Function $\mathrm{checkReachAcc}$ takes a set of paths as input. From the last state in each path, it checks if an accepting state in the input TA is reachable or not (i.e., whether a state in Q_F is reachable). It returns `tt`, if an accepting state is reachable, and `ff` otherwise. Formally it checks whether $k_{Q^F \cup B^C}(q, t_F, \sigma)$ is empty.

The algorithm proceeds as follows. If the set of accepting paths is not empty (i.e., a state in Q_F is reachable upon delaying $\sigma_{\mathrm{mc}} \cdot (t, a)$), then function update returns `ok`, the optimal word computed using $\mathrm{getOptimalWord}$, and the state reached in the TA (computed using the function post). Otherwise, it checks if it is possible to reach an accepting state in the future (computed using function $\mathrm{checkReachAcc}$). If it is impossible to reach an accepting state (i.e., from all the states reached upon delaying $\sigma_{\mathrm{mc}} \cdot (t, a)$, Q_F is not reachable), then function update returns `bad`, σ_{mc}, and the current state q. Otherwise, it returns the current state q, $\sigma_{\mathrm{mc}} \cdot (t, a)$, and `c_bad`.

Algorithm StoreProcess *(see Algorithm 2).* Algorithm StoreProcess is an infinite loop that scrutinises the system for input events. In the algorithm, q represents the state of the property automaton.

The algorithm proceeds as follows. StoreProcess initially sets its clock t to 0. This clock keeps track of the time elapsed and increases with physical time. Variable t_F is initialised to 0. This variable contains the date of the last event of σ_{ms}, if σ_{ms} is not empty, and the date of the last released event otherwise. The algorithm also initialises q to q_0, and the two memories σ_{ms} and σ_{mc} to ϵ. It then enters an infinite loop where it waits for an input event ($\mathrm{await}(event)$). When receiving an action a at date t, it stores event (t, a). It then invokes function update with the current state q, the arrival date t_F, the events stored in σ_{mc} and the new event (t, a). Then, function update returns a new state q', a timed word σ'_{mc} and the marker $isPath$. If marker $isPath =$ `ok`, it means that $\sigma_{\mathrm{mc}} \cdot (t, a)$ can be corrected into the timed word σ'_{mc} computed by update and this word leads from state q to state q' in the underlying

Algorithm 2 StoreProcess

$t \leftarrow 0$
$(q, t_F) \leftarrow (q_0, 0)$
$(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}) \leftarrow (\epsilon, \epsilon)$
while `tt` **do**
  $(t, a) \leftarrow \mathrm{await}(event)$ /* i.e., action a is received at date t */
  $(q', \sigma'_{\mathrm{mc}}, isPath) \leftarrow \mathrm{update}(q, t_F, \sigma_{\mathrm{mc}}, (t, a))$
  if $isPath = \mathtt{ok}$ **then**
    $\sigma_{\mathrm{ms}} \leftarrow \sigma_{\mathrm{ms}} \cdot \sigma'_{\mathrm{mc}}$
    $\sigma_{\mathrm{mc}} \leftarrow \epsilon$
    $q \leftarrow q'$
    $t_F \leftarrow \mathrm{end}(\sigma'_{\mathrm{mc}})$
  else
    $\sigma_{\mathrm{mc}} \leftarrow \sigma'_{\mathrm{mc}}$
  end if
end while

semantics of the timed automaton, at date $\mathrm{end}(\sigma'_{\mathrm{mc}})$. Then, timed word σ'_{mc} is appended to shared memory σ_{ms} (since σ'_{mc} leads to an accepting state q' from state q), the internal memory σ_{mc} is cleared, state q is updated to q' and t_F to $\mathrm{end}(\sigma'_{\mathrm{mc}})$. In all other cases, σ_{mc} is set to σ'_{mc}, the result of update, which is either σ_{mc} if $isPath = \mathtt{bad}$ (it is impossible to correct the input sequence σ_{mc} whatever are the future events) or $\sigma_{\mathrm{mc}} \cdot (t, a)$ if $isPath = \mathtt{c\_bad}$. Event (t, a) is thus deleted. In both cases, state q, t_F and memory σ_{ms} are not modified.

Algorithm 3 ReleaseProcess

$d \leftarrow 0$
while `tt` **do**
  $\mathrm{await}(\sigma_{\mathrm{ms}} \neq \epsilon)$
  $(t, a) \leftarrow \mathrm{dequeue}(\sigma_{\mathrm{ms}})$
  $\mathrm{wait}(t - d)$
  $\mathrm{release}(a)$
end while

Algorithm ReleaseProcess *(see Algorithm 3).* Algorithm ReleaseProcess is an infinite loop that scrutinises memory σ_{ms} and releases actions as output.

The algorithm proceeds as follows. Initially, clock d, which keeps track of the time elapsed, is set to 0 and then increases with physical time. ReleaseProcess waits until the memory is not empty ($\sigma_{\mathrm{ms}} \neq \epsilon$). Using operation $\mathrm{dequeue}$, the first element stored in the memory is removed, and is stored as (t, a). Since d time units elapsed, process ReleaseProcess waits for $(t - d)$ time units before performing operation $\mathrm{release}(a)$, releasing action a as output at date t (which amounts to appending (t, a) to the output of the enforcement monitor).

Remark 8 (Launching StoreProcess **and** ReleaseProcess**).** In order to respect the semantics of the enforcement monitor, the two processes StoreProcess and ReleaseProcess should be launched simultaneously. This ensures that their current dates (encoded by t for StoreProcess and d for ReleaseProcess) are always equal.

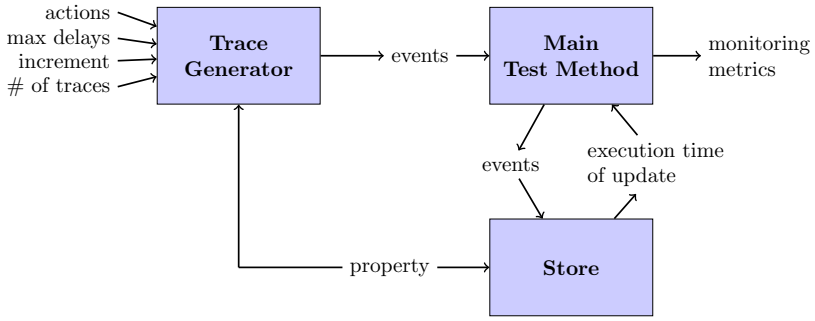


Figure 13: Experimental framework.

9. Implementation and evaluation

We implemented the algorithms in Section 8 and developed an experimentation framework called TiPEX: (Timed Properties Enforcement during eXecution)[9] in order to:

1. validate through experiments the architecture and feasibility of enforcement monitoring, and
2. measure and analyse the performance of the update function of the StoreProcess.

From [5], we completely re-implemented the synthesis of enforcement monitors. TiPEX supports now all regular properties. The prototype presented in [5] handles only safety and co-safety properties, with independent algorithms and prototype implementations for each class. Now, following the algorithms proposed in this paper, TiPEX supports all regular properties defined by deterministic one-clock timed automata. In [20], we describe the implementation of a simplified version of function update that does not allow to suppress events. We recently implemented another version of function update based on the enforcement mechanisms and algorithms described in Section 8. In this section, we compare the performance of the implementations of these functions. Note, when we consider suppression, when an accepting state is not reachable with the events received so far, we need to perform another additional computationally-expensive analysis (checkReachAcc in Algorithm 1), to decide whether or not the last event should be suppressed.

The rest of this section is organised as follows. Section 9.1 describes our experimental framework. Section 9.2 present the properties used in the evaluation. Section 9.3 discusses the evaluation results.

9.1. Experimental framework

The experimental framework is depicted in Figure 13. As mentioned in [5], regarding algorithm StoreProcess, the most computationally intensive step is the call to function update. We thus focus on this function in the evaluation.

Module Main uses module Trace Generator that provides a set of input traces to test the module Store. Module Trace Generator takes as input the alphabet of actions, the range of possible delays between actions, the desired number of traces, and the increment in length per trace. For example, if the number of traces is 5 and the increment in length per trace is 100, then 5 traces will be generated, where the first trace is of length 100 and the second trace of length 200 and so on. For each event, Trace Generator picks an action (from the set of possible actions), and a random delay (from the set of

[9] Available at `http://srinivaspinisetty.github.io/Timed-Enforcement-Tools/`.

possible delays) which is the time elapsed after the previous event or the system initialization for the first event. For this purpose, Trace Generator uses methods from the Python random module.

Module Store takes as input a property and one trace, and returns the total execution time of the update function to process the given input trace. The TA modelling the property is a UPPAAL [21] model written in XML. Module Store uses the pyuppaal library to parse the UPPAAL model (input property), and the UPPAAL DBM library to implement the update function.[10] The sequence of events received by the enforcement monitor is modelled by a second UPPAAL model. Module Main Test Method sends this sequence to module Store (using the property), and keeps track of the result returned by the Store module for each trace.

Experiments were conducted on an Intel Core i5-4210U at 1.70GHz CPU, with 4 GB RAM, and running on Ubuntu 14.04 LTS.

9.2. Description of the properties

We describe the properties used in our experiments and discuss the results of the performance analysis.

The properties follow different patterns [22], and belong to different classes. They are inspired from the properties introduced in Example 1. They are recognised by one-clock timed automata since this is a limitation of our current implementation (extension to more than one clock is ongoing). We however expect the trends exposed in the following to be similar when the complexity of automata grows, since it induces heavier computation for each call to function update.

- Property φ_{s} is a safety property expressing that "*There should be a delay of at least* 5 *time units (t.u.) between any two request actions*".
- Property φ_{cs} is a co-safety property expressing that "*A request should be immediately followed by a grant, and there should be a delay of at least* 6 *t.u. between them*".
- Property φ_{re} is a regular property, but neither a safety nor a co-safety property, and expresses that "*Resource grant and release should alternate. After a grant, a request should occur between 15 to 20 t.u.*".

The automata defining the above properties can be found in [23].

9.3. Performance evaluation of function update

Results of the performance analysis for the properties are reported in Tables 1, 2, and 3. The reported numbers are mean values over 10 runs. Note, 10 runs were sufficient to obtain 95% confidence for all metrics, and the measurement error was less than 1%. The entry $|tr|$ indicates the length of the input trace (i.e., the number of events input to the enforcement monitor). The entry t_update (resp. t_update-sup) indicates the total execution time of the function update without (resp. with) suppression in seconds. The entry *mem* (resp. *mem*-sup) indicates the maximum memory used by the Main Test Method when using function update without (resp. with) suppression; both measured in megabytes.

[10] The `pyuppaal` and `DBM` libraries are provided by Aalborg University and can be downloaded at `http://people.cs.aau.dk/~adavid/python/`.

Table 1: Performance analysis of enforcement monitors for φ_{s}.

$\lvert tr\rvert$	t_update	t_update-sup	*mem*	*mem*-sup
$10,000$	6.44	6.64	17.8	17.9
$20,000$	12.73	13.44	19.6	19.6
$30,000$	19.51	20.16	21.3	21.3
$40,000$	26.41	26.50	22.6	22.7
$50,000$	31.88	33.10	24.3	24.3
$60,000$	38.44	39.84	26.2	26.2
$70,000$	45.16	45.92	27.7	27.8
$80,000$	51.21	53.34	29.1	29.1

Strategy for generating traces.. To have a meaningful performance assessment of function update, module Trace Generator uses a strategy to ensure that calls to function update yields computation using σ_{mc}. For (the safety) property φ_{s}, module Trace Generator generates events so that each event of the trace leads to a call to function update to correct the date of the input event. This strategy allows to assess the performance of function update when it is extensively used with buffer σ_{mc} empty. For (the co-safety) property φ_{cs}, module Trace Generator ensures that input sequences can be corrected only on the last event (hence implying that, for a sequence of length n, function update is called n times where the buffer containing σ_{mc} is of size $i-1$ on the i^{th} call). This strategy allows to assess the performance of function update when σ_{mc} is used significantly. For (the regular property) φ_{re}, module Trace Generator ensures that the property can be corrected every two events, which is the length of the minimal path between accepting locations of the underlying automaton of φ_{re}. This strategy allows to asses the performance of function update when alternating between finding a correction of the input sequence using buffer σ_{mc} and buffering corrected events in buffer σ_{ms}.

Safety property φ_{s} (see Table 1). We can observe that t_update, and t_update-sup increase linearly with the length of the input trace. Moreover, the time taken per call to update (i.e., $^{t\_\mathrm{update}}/_{\lvert tr\rvert}$) does not depend on the length of the trace. This behaviour is as expected for a safety property. Indeed, function update is always called with only one event which is read as input (the internal buffer σ_{mc} remains empty). Consequently, the state of the TA is updated after each event, and after receiving a new event, the possible transitions leading to a good state from the current state are explored. For the same input trace, there is no significant variation in the values of t_update, and t_update-sup. This behaviour is as expected because for the considered safety property (φ_{s}) and input traces, after receiving a new event, it is always possible to compute a delay to satisfy the property. Thus, in the function update with suppression, $\mathrm{checkReachAcc}$ is never invoked.

Regarding memory usage, we can notice that by increasing the length of the input trace by 10,000, the peak memory usage increases by less than 2 MB. For the same input trace, there is no variation in memory usage (*mem* and *mem*-sup are equal).

Regular property φ_{re} (see Table 2). Recall that the considered input traces are generated in such a way that they can be corrected every two events. Consequently, function update is invoked with either one or two events. For the considered input traces, the time taken per call to function update does not depend on the length of the trace. Moreover, for input traces of same length, the value of $\mathrm{t\_update}$ (resp. t_update-sup) is higher for φ_{re} than the value of $\mathrm{t\_update}$ (resp. t_update-sup) for φ_{s}. This stems from the fact that, for a safety property, function update is invoked only with one event. Furthermore, for the same input trace, t_update-sup is greater than t_update. This stems from the fact that, for

Table 2: Performance analysis of enforcement monitors for φ_{re}.

$\lvert tr\rvert$	t_update	t_update-sup	*mem*	*mem*-sup
10,000	10.21	20.33	17.6	17.6
20,000	20.56	39.32	19.0	19.0
30,000	30.95	61.20	20.2	20.2
40,000	42.37	82.23	21.6	21.6
50,000	53.67	101.46	22.8	22.8
60,000	62.06	121.55	24.2	24.2
70,000	81.63	137.49	25.4	25.4
80,000	91.89	167.16	26.8	26.8

the considered input traces (where it is possible to correct every two events) the function update with suppression invokes function $\mathrm{checkReachAcc}$ $\lvert tr\rvert/2$ times.

Regarding memory usage, by increasing the length of the input trace by 10,000, the peak memory usage increases by less than 2 MB. For input traces of same length, there is no significant variation in the values of *mem* and *mem*-sup between φ_{re} and φ_{s}.

Table 3: Performance analysis of enforcement monitors for φ_{cs}.

$\lvert tr\rvert$	t_update	t_update-sup	*mem*	*mem*-sup
100	2.022	2.256	16.4	16.4
200	8.124	8.547	16.4	16.4
300	18.207	18.868	16.4	16.4

Co-safety property φ_{cs} (see Table 3). Recall that the considered input traces are generated in such a way that they can be corrected only upon the last event. From the results presented in Table 3, notice that t_update, and t_update-sup are now quadratic. Moreover, the average time per call to function update increases with $\lvert tr\rvert$. For the considered input traces, this behaviour is as expected for a co-safety property because the length of the internal buffer σ_{mc} increases after each event, and thus function update is invoked with a growing sequence.

For the same input trace, t_update-sup is greater than t_update. For example, for input traces of length 100, t_update-sup is around 0.2 seconds greater than t_update. Indeed, for the considered input traces (where it is possible to correct the input sequence only upon the last event) the function update with suppression invokes function $\mathrm{checkReachAcc}$ $\lvert tr\rvert - 1$ times. We can also observe that t_update-sup $-t\_\mathrm{update}$ increases linearly with $\lvert tr\rvert$.

Regarding memory usage, since we consider small increments of the input traces, we can not notice significant variation. For input trace of length 100, peak memory usage noticed for φ_{s} is 16.5 MB. Thus we can notice that, for input traces of same length, for φ_{re}, φ_{s}, and φ_{cs}, there is no significant variation in the value of *mem*.

10. Related work

Several approaches for the runtime verification and enforcement of properties are related to the one proposed in this paper.

10.1. Runtime verification

As a verification/validation technique, runtime enforcement is related to runtime verification. At an abstract level, a runtime verification approach consists in synthesising a verification monitor (cf. [24]), i.e., a decision procedure used at runtime. The monitor observes the system under scrutiny and emits verdicts regarding the satisfaction or violation of the property of interest. See [25, 26, 27, 28] for short tutorials and surveys on runtime verification. Runtime verification principles have been used in many concrete application domains and for various purposes such as the safety checking of cyber-physical systems [29, 30, 31, 32], the security of financial and IT systems [33, 34, 35], and many more.

10.2. Runtime verification of timed properties

We discuss more specifically some approaches for the runtime verification of timed properties for real-time systems. One can also refer to the survey of Goodloe and Pike [36] which presents some approaches to monitoring hard real-time systems and potential application-domains when monitoring safety properties.

Several approaches consider the problem of synthesising automata-based monitors from formulae in temporal logics that handle physical time (as opposed to logical time). Sokolsky et al. [37] introduced an expressive first-order logic tailored for runtime verification. The logic features event attributes (aka parametric events) and dynamic indexing of properties (to handle the dynamic creation of monitors at runtime). Models of the logic also refers to physical time. Bauer et al. [38] synthesised monitors for timed-bounded properties expressed in a variant of Timed Linear Temporal Logic tailored for monitoring. Nickovic et al. [17, 18] synthesised timed automata from Metric Temporal Logic (a temporal logic with a dense notion of time). Still for MTL, Thati [39] use rewriting of formulae for online monitoring. All these approaches are compatible with ours since they are purposed to synthesise decision procedures for logic-based timed specification formalisms. More specifically, the synthesised automata-based monitors can be used as input in our approach as replacements of timed automata.

Basin et al [40] provided a general comparison of monitoring algorithms for real-time systems. Time models are categorised as i) either point-based algorithms or interval-based, and ii) either dense or discrete depending on the underlying ordering of time points (i.e., finitely or infinitely many time points). Basin et al. presented and compared monitoring algorithms for the past-only fragment of propositional metric temporal logic.

Several tools have been proposed for monitoring timed properties. RT-MaC [41] verifies timeliness and reliability correctness properties at runtime. The Analog Monitoring Tool [42] verifies formulae in Signal Temporal Logic over continuous signals. LARVA [43, 44] verifies properties (over Java programs) expressed in several specification formalisms by translating input specifications into the so-called Dynamic Automata with Timers and Events which basically resemble timed automata with stop watches. Contrary to these approaches, the monitors presented in this paper differ in their objectives and how they are interfaced with the system: the monitors are not intended to modify the internal state of the system but rather to modify a sequence of timed events between two systems.

10.3. Runtime enforcement of untimed properties

Roughly speaking, the research efforts in runtime enforcement aims at defining and implementing enforcement primitives that supplement the monitors used in runtime verification. Most of the work in runtime enforcement was dedicated to untimed properties (see [45] for a short overview). Schneider introduced security automata as the first runtime mechanism for enforcing safety properties [1]. Ligatti et al. [3] later introduced edit-automata as enforcement monitors. Edit-automata can insert a new action by replacing the current input, or suppress it. Similar to edit-automata are generic enforcement monitors [4] which are finite-state machines augmented with a memory and parameterised with enforcement

primitives operating on the input and memory. Moreover, some variants of edit-automata differ in how they ensure the transparency constraints (see e.g., [46]). Synthesis techniques of enforcement mechanisms from a property have been proposed only for generic enforcement monitors [4] and restricted forms of edit-automata [3].

Note, several runtime verification tools allow the user to define some treatment of errors through the (manual) definition of some form of enforcement primitives. For instance, JavaMOP and the RV system [47] define the notion of *code handler* which are user-defined code-snippets that can be attached to monitor states. LARVA allows the user to specify corrective actions [48] that can be used for undoing the effects of previous actions carried out by the system.

10.4. Runtime enforcement of timed properties

The endeavours on runtime enforcement discussed in the previous subsection consider logical time, as opposed to physical time. Moreover, storing an event is assumed without consequence on the execution nor on the satisfiability of the property, i.e., the duration during which an event is retained in memory has no influence. In the following of this subsection, we discuss the approaches on runtime enforcement that consider physical time.

Basin et al. [49] refined the work of Schneider on security automata to take into account discrete-time constraints by modelling the passing of time as uncontrollable events. Similarly, we consider elapsing of time as uncontrollable but consider dense time. The enforcement mechanisms in [49] differ from ours in several aspects: they consider only truncation automata (and they are thus limited to safety properties, not necessarily regular). Moreover, our enforcement mechanisms have additional enforcement primitives: buffering of actions (which basically amounts to letting time elapse) and suppression of actions which allows for longer inputs to be processed by enforcement mechanisms.

In previous work [5, 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 algorithms. In [5], only safety and co-safety properties are considered and different definitions of mechanisms are proposed for each class. In [9], all regular properties are considered. Given a timed automaton, enforcement functions, monitors and algorithms are synthesised according to one general definition. Also, for the enforcement of co-safety properties, the approach in [5] assumes that time elapses differently for input and output sequences (the sequences are desynchronised). 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 mechanism has read a sequence that can be delayed into a correct sequence). Compared to [5], the approaches in [9] and this paper are more realistic as they do not suffer from this "shift" problem.

10.5. Monitorability and enforceability

In this paper, we identify some timed properties that are not enforceable by mechanisms that comply to the constraints mentioned in Section 5.2 (see Example 6). Characterising monitorable properties (i.e., properties that can be runtime verified) and enforceable properties are two important endeavours. We briefly discuss some of the main approaches on these topics in the following and discuss in Section 11.2 how we plan to characterise enforceable timed properties in the future.

Monitorable properties. Kim et al. [50] first defined monitorable properties as the co-recursively enumerable safety properties. Pnueli et al. [51] generalised the definition to the properties for which it is always possible to determine a definitive satisfaction or violation at runtime. Bauer et al. [38] showed that safety and co-safety properties are monitorable in the sense of [51]. Later, Falcone et al. [10] showed that obligation properties form a strict subset of the set of monitorable properties in the sense of [51], but that less properties should be monitored in practice. Sistla et al. [52] defined necessary and

sufficient conditions for the monitorability of hybrid systems where an Extended Hidden Markov system is monitorable if there exists an arbitrarily-precise monitor stating verdicts on the system outputs. More recently, Rosu [53] defined monitorable properties as safety properties arguing that these can be specified by general (finite-state machine) monitors.

Enforceable properties. Enforceable properties are the properties for which a sound and transparent enforcement monitor can be synthesised. The set of enforceable properties depends on the primitives conferred to enforcement monitors. Security automata [1] can enforce safety properties. Note, Schneider, Hamlen, and Morrisett [2] showed that security automata can only monitor co-recursively enumerable safety properties because of computational limits exhibited by Viswanathan and Kim [54]. Edit-automata [3] can enforce infinite renewal properties. (The set of infinite renewal properties is a super-set of safety properties and contains some liveness properties.) Generalised enforcement monitors [4] can enforce response properties in the safety-progress classification. In addition to enforcement primitives and computability constraints, enforceability limitations arise when properties are expressed over infinite sequences (see [45, 4] for a comparison of enforceable untimed properties over infinite sequences). However, any property over finite sequences is enforceable with a monitor endowed with the primitives of an edit-automaton (see Section 10.3 and [3]) [10]. More recently, Basin et al. [49] showed that security automata can enforce the safety properties that cannot be violated through a sequence of uncontrollable events.

11. Conclusions

11.1. Summary

This paper presents a general enforcement monitoring framework for systems with timing requirements. We show how to synthesise enforcement mechanisms for any regular timed property (modelled by a timed automaton). The enforcement mechanisms proposed in this paper are more powerful than the ones in our previous research endeavours [5, 9]. In particular, in this paper, we propose enforcement mechanisms that delay the absolute dates of events of the observed input (while being allowed to shorten the delay between some events). Moreover, suppressing events is also introduced. An event is suppressed if it is not possible to satisfy the property by delaying, whatever are the future continuations of the input sequence (i.e., the underlying TA can only reach non-accepting states from which no accepting state can be reached). Formalising suppression required us to revisit the formalisation of all enforcement mechanisms. Enforcement mechanisms are described at several levels of abstraction (enforcement function, monitor, and algorithms), thus facilitating the design and implementation of such mechanisms. We propose a prototype implementation and our experiments demonstrate the feasibility of enforcement monitoring for timed properties.

11.2. 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 characterised as non-enforceable. For this purpose, an enforceability condition should be defined and used to delineate enforceable properties. Such a criterion should also ideally be expressible on timed automata. Note however that, even for non-enforceable properties, enforcement monitors can be built, but may not be able to output some correct input sequences. The output sequences of our enforcement mechanisms are however always either correct or empty.

Specifications are currently modelled with timed automata. One can consider synthesising enforcement mechanisms from more expressive formalisms. For instance, we could 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, as shown for safety properties in [8]).

Implementing efficient enforcement monitors is another important aspect and should be done in a particular application domain. We propose TiPEX, a Python implementation of enforcement mechanisms with the objectives of i) making a quick prototype that shows feasibility of enforcement monitoring in a timed context, and ii) reusing some existing UPPAAL libraries. In the future, we will consider implementing our enforcement monitors in other languages such as C or Java, and we expect even better performance and a more stand-alone implementation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their remarks and suggestions on an early version of this article.

The work reported in this article has been done in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

Bibliography

[1] F. B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50. doi:10.1145/353323.353382.

[2] K. W. Hamlen, G. Morrisett, F. B. Schneider, Computability classes for enforcement mechanisms, ACM Trans. Program. Lang. Syst. 28 (1) (2006) 175–205. doi:10.1145/1111596.1111601.

[3] J. Ligatti, L. Bauer, D. Walker, Run-time enforcement of nonsafety policies, ACM Trans. Inf. Syst. Secur. 12 (3) (2009) 19:1–19:41. doi:10.1145/1455526.1455532.

[4] Y. Falcone, L. Mounier, J.-C. Fernandez, J.-L. Richier, Runtime enforcement monitors: composition, synthesis, and enforcement abilities, Formal Methods in System Design 38 (3) (2011) 223–262. doi:10.1007/s10703-011-0114-4.

[5] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, O. L. N. Timo, Runtime enforcement of timed properties, in: S. Qadeer, S. Tasiran (Eds.), Proceedings of the Third International Conference on Runtime Verification (RV 2012), Vol. 7687 of Lecture Notes in Computer Science, Springer, 2012, pp. 229–244. doi:10.1007/978-3-642-35632-2_23.

[6] B. Zeng, G. Tan, Ú. Erlingsson, Strato: A retargetable framework for low-level inlined-reference monitors, in: S. T. King (Ed.), Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013, USENIX Association, 2013, pp. 369–382.

[7] Ú. Erlingsson, F. B. Schneider, IRM enforcement of Java stack inspection, in: 2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000, IEEE Computer Society, 2000, pp. 246–255. doi:10.1109/SECPRI.2000.848461.

[8] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, Runtime enforcement of parametric timed properties with practical applications, in: J. Lesage, J. Faure, J. E. R. Cury, B. Lennartson (Eds.), 12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014., International Federation of Automatic Control, 2014, pp. 420–427. doi:10.3182/20140514-3-FR-4046.00041.

[9] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, Runtime enforcement of regular timed properties, in: Y. Cho, S. Y. Shin, S.-W. Kim, C.-C. Hung, J. Hong (Eds.), Proceedings of the ACM Symposium on Applied Computing (SAC-SVT), ACM, 2014, pp. 1279–1286. doi:10.1145/2554850.2554967.

[10] Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime?, STTT 14 (3) (2012) 349–382. doi:10.1007/s10009-011-0196-8.

[11] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Comp. Science 126 (1994) 183–235. doi:10.1016/0304-3975(94)90010-8.

[12] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Lectures on Concurrency and Petri Nets, Advances in Petri Nets, Vol. 3098 of Lecture Notes in Computer Science, Springer, 2003, pp. 87–124. doi:10.1007/978-3-540-27755-2_3.

[13] T. A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model checking for real-time systems, Inf. Comput. 111 (2) (1994) 193–244. doi:10.1006/inco.1994.1045.

[14] G. Behrmann, A. David, K. G. Larsen, A tutorial on UPPAAL, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures, Vol. 3185 of Lecture Notes in Computer Science, Springer, 2004, pp. 200–236. doi:10.1007/978-3-540-30080-9_7.

[15] R. Alur, L. Fix, T. A. Henzinger, Event-clock automata: A determinizable class of timed automata, Theor. Comput. Sci. 211 (1-2) (1999) 253–273. doi:10.1016/S0304-3975(97)00173-4.

[16] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: J. Desel, W. Reisig, G. Rozenberg (Eds.), Proceedings of the 4th Advanced Course on Petri Nets - Lecture Notes on Concurrency and Petri Nets, Vol. 3098 of LNCS, Springer, 2003, pp. 87–124. doi:10.1007/978-3-540-27755-2_3.

[17] O. Maler, D. Nickovic, A. Pnueli, From MITL to timed automata, in: E. Asarin, P. Bouyer (Eds.), Proceedings of the 4th international conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2006), Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 274–289. doi:10.1007/11867340_20.

[18] D. Nickovic, N. Piterman, From MTL to deterministic timed automata, in: K. Chatterjee, T. A. Henzinger (Eds.), Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2010), Vol. 6246 of Lecture Notes in Computer Science, Springer, 2010, pp. 152–167. doi:10.1007/978-3-642-15297-9_13.

[19] P. Bouyer, T. Brihaye, V. Bruyère, J.-F. Raskin, On the optimal reachability problem of weighted timed automata, Formal Methods in System Design 31 (2) (2007) 135–175. doi:10.1007/s10703-007-0035-4.

[20] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, TiPEX: A tool chain for timed property enforcement during execution, in: Bartocci and Majumdar [56], pp. 306–320. doi:10.1007/978-3-319-23820-3_22.

[21] K. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, STTT 1 (1-2) (1997) 134–152. doi:10.1007/s100090050010.

[22] V. Gruhn, R. Laue, Patterns for timed property specifications, Electronic Notes in Theoretical Computer Science 153 (2) (2006) 117–133. doi:10.1016/j.entcs.2005.10.035.

[23] S. Pinisetty, Y. Falcone, T. Jéron, H. Marchand, A. Rollet, O. Nguena-Timo, Runtime enforcement of timed properties revisited, Formal Methods in System Design 45 (3) (2014) 381–422. doi:10.1007/s10703-014-0215-y.

[24] K. Havelund, G. Rosu, Synthesizing monitors for safety properties, in: J. Katoen, P. Stevens (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, Vol. 2280 of Lecture Notes in Computer Science, Springer, 2002, pp. 342–356. doi:10.1007/3-540-46002-0_24.

[25] K. Havelund, A. Goldberg, Verify your runs, in: B. Meyer, J. Woodcock (Eds.), Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions, Vol. 4171 of Lecture Notes in Computer Science, Springer, 2005, pp. 374–383. doi:10.1007/978-3-540-69149-5_40.

[26] M. Leucker, C. Schallhart, A brief account of runtime verification, J. Log. Algebr. Program. 78 (5) (2009) 293–303. doi:10.1016/j.jlap.2008.08.004.

[27] O. Sokolsky, K. Havelund, I. Lee, Introduction to the special section on runtime verification, STTT 14 (3) (2012) 243–247. doi:10.1007/s10009-011-0218-6.

[28] Y. Falcone, K. Havelund, G. Reger, A tutorial on runtime verification, in: M. Broy, D. Peled, G. Kalus (Eds.), Engineering Dependable Software Systems, Vol. 34 of NATO Science for Peace and Security Series, D: Information and Communication Security, IOS Press, 2013, pp. 141–175. doi:10.3233/978-1-61499-207-3-141.

[29] D. Seto, B. Krogh, L. Sha, A. Chutinan, The simplex architecture for safe online control system upgrades, in: American Control Conference, 1998. Proceedings of the 1998, Vol. 6, 1998, pp. 3504–3508 vol.6. doi:10.1109/ACC.1998.703255.

[30] S. Bak, K. Manamcheri, S. Mitra, M. Caccamo, Sandboxing controllers for cyber-physical systems, in: 2011 IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, Chicago, Illinois, USA, 12-14 April, 2011, IEEE Computer Society, 2011, pp. 3–12. doi:10.1109/ICCPS.2011.25.

[31] S. Bak, F. A. T. Abad, Z. Huang, M. Caccamo, Using run-time checking to provide safety and progress for distributed cyber-physical systems, in: 2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2013, Taipei, Taiwan, August 19-21, 2013, IEEE, 2013, pp. 287–296. doi:10.1109/RTCSA.2013.6732229.

[32] S. Mitsch, A. Platzer, Modelplex: Verified runtime validation of verified cyber-physical system models, in: Bonakdarpour and Smolka [55], pp. 199–214. doi:10.1007/978-3-319-11164-3_17.

[33] C. Colombo, G. J. Pace, Fast-forward runtime monitoring - an industrial case study, in: S. Qadeer, S. Tasiran (Eds.), Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, Vol. 7687 of Lecture Notes in Computer Science, Springer, 2012, pp. 214–228. doi:10.1007/978-3-642-35632-2_22.

[34] D. A. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, H. Mantel, Scalable offline monitoring, in: Bonakdarpour and Smolka [55], pp. 31–47. doi:10.1007/978-3-319-11164-3_4.

[35] A. Kassem, Y. Falcone, P. Lafourcade, Monitoring electronic exams, in: Bartocci and Majumdar [56], pp. 118–135. doi:10.1007/978-3-319-23820-3_8.

[36] A. Goodloe, L. Pike, Monitoring distributed real-time systems: A survey and future directions, Tech. Rep. NASA/CR-2010-216724, NASA Langley Research Center, available at http://ntrs.nasa.gov (July 2010).

[37] O. Sokolsky, U. Sammapun, I. Lee, J. Kim, Run-time checking of dynamic properties, Electr. Notes Theor. Comput. Sci. 144 (4) (2006) 91–108. doi:10.1016/j.entcs.2006.02.006.

[38] A. Bauer, M. Leucker, C. Schallhart, Runtime verification for LTL and TLTL, ACM Trans. Softw. Eng. Methodol. 20 (4) (2011) 14:1–14:64. doi:10.1145/2000799.2000800.

[39] P. Thati, G. Rosu, Monitoring algorithms for metric temporal logic specifications, Electronic Notes in Theoretical Computer Science 113 (2005) 145–162. doi:10.1016/j.entcs.2004.01.029.

[40] D. A. Basin, F. Klaedtke, E. Zalinescu, Algorithms for monitoring real-time properties, in: Khurshid and Sen [57], pp. 260–275. doi:10.1007/978-3-642-29860-8_20.

[41] U. Sammapun, I. Lee, O. Sokolsky, RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties, 2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications 0 (2005) 147–153. doi:http://doi.ieeecomputersociety.org/10.1109/RTCSA.2005.84.

[42] D. Nickovic, O. Maler, AMT: a property-based monitoring tool for analog systems, in: J.-F. Raskin, P. S. Thiagarajan (Eds.), Proceedings of the 5th International Conference on Formal modeling and analysis of timed systems (FORMATS 2007), Vol. 4763 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 304–319. doi:10.1007/978-3-540-73368-3_12.

[43] C. Colombo, G. J. Pace, G. Schneider, Dynamic event-based runtime monitoring of real-time and contextual properties, in: D. D. Cofer, A. Fantechi (Eds.), Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers, Vol. 5596 of Lecture Notes in Computer Science, Springer, 2008, pp. 135–149. doi:10.1007/978-3-642-03240-0_13.

[44] C. Colombo, G. J. Pace, G. Schneider, LARVA — safer monitoring of real-time Java programs (tool paper), in: D. V. Hung, P. Krishnan (Eds.), Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009), IEEE Computer Society, 2009, pp. 33–37. doi:10.1109/SEFM.2009.13.

[45] Y. Falcone, You should better enforce than verify, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Proceedings of the First International Conference on Runtime Verification (RV 2010), Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 89–105. doi:10.1007/978-3-642-16612-9_9.

[46] N. Bielova, F. Massacci, Do you really mean what you actually enforced? - edit automata revisited, Int. J. Inf. Sec. 10 (4) (2011) 239–254. doi:10.1007/s10207-011-0137-2.

[47] P. O. Meredith, G. Rosu, Runtime verification with the RV system, in: H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, N. Tillmann (Eds.), Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, Vol. 6418 of Lecture Notes in Computer Science, Springer, 2010, pp. 136–152. doi:10.1007/978-3-642-16612-9_12.

[48] C. Colombo, G. J. Pace, P. Abela, Safer asynchronous runtime monitoring using compensations, Formal Methods in System Design 41 (3) (2012) 269–294. doi:10.1007/s10703-012-0142-8.

[49] D. A. Basin, V. Jugé, F. Klaedtke, E. Zalinescu, Enforceable security policies revisited, ACM Trans. Inf. Syst. Secur. 16 (1) (2013) 3. doi:10.1145/2487222.2487225.

[50] M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, Computational analysis of run-time monitoring - fundamentals of Java-MaC, Electr. Notes Theor. Comput. Sci. 70 (4) (2002) 80–94. doi:10.1016/S1571-0661(04)80578-4.

[51] A. Pnueli, A. Zaks, PSL model checking and run-time verification via testers, in: J. Misra, T. Nipkow, E. Sekerinski (Eds.), FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings, Vol. 4085 of Lecture Notes in Computer Science, Springer, 2006, pp. 573–586. doi:10.1007/11813040_38.

[52] A. P. Sistla, M. Zefran, Y. Feng, Runtime monitoring of stochastic cyber-physical systems with hybrid state, in: Khurshid and Sen [57], pp. 276–293. doi:10.1007/978-3-642-29860-8_21.

[53] G. Rosu, On safety properties and their monitoring, Sci. Ann. Comp. Sci. 22 (2) (2012) 327–365.

[54] M. Viswanathan, M. Kim, Foundations for the run-time monitoring of reactive systems - Fundamentals of the MaC language, in: Z. Liu, K. Araki (Eds.), Proceedings of the First International Colloquium on Theoretical Aspects of Computing (ICTAC 2004), Vol. 3407 of Lecture Notes in Computer Science, Springer, 2004, pp. 543–556. doi:10.1007/978-3-540-31862-0_38.

[55] B. Bonakdarpour, S. A. Smolka (Eds.), Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, Vol. 8734 of Lecture Notes in Computer Science, Springer, 2014. doi:10.1007/978-3-319-11164-3.

[56] E. Bartocci, R. Majumdar (Eds.), Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings, Vol. 9333 of Lecture Notes in Computer Science, Springer, 2015. doi:10.1007/978-3-319-23820-3.

[57] S. Khurshid, K. Sen (Eds.), Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers, Vol. 7186 of Lecture Notes in Computer Science, Springer, 2012. doi:10.1007/978-3-642-29860-8.

Appendix A. Proofs

Recall that $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ is defined as:

$$E_\varphi(\sigma) = \ \Pi_1\big(\mathrm{store}_\varphi(\sigma)\big).$$

where $\mathrm{store}_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma) \times \mathrm{tw}(\Sigma)$ is defined as

$$\begin{array}{rcl} \mathrm{store}_\varphi(\epsilon) & = & (\epsilon, \epsilon) \\ \mathrm{store}_\varphi(\sigma \cdot (t,a)) & = & \begin{cases} (\sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c), \epsilon), & \text{if } \kappa_\varphi(\sigma_s,\sigma'_c) \neq \emptyset, \\ (\sigma_s, \sigma_c) & \text{if } \kappa_{\mathrm{pref}(\varphi)}(\sigma_s,\sigma'_c) = \emptyset \\ (\sigma_s, \sigma'_c) & \text{otherwise,} \end{cases} \\ & & \text{with } \sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}, a \in \Sigma, \\ & & (\sigma_s, \sigma_c) = \mathrm{store}_\varphi(\sigma), \text{ and } \sigma'_c = \sigma_c \cdot (t,a) \end{array}$$

where:

$$\kappa_\varphi(\sigma_s, \sigma'_c) \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma'_c) \cap \sigma_s^{-1} \cdot \varphi,$$

as defined in Section 6.2, with:

$$\mathrm{CanD}(\sigma) = \{w \in \mathrm{tw}(\Sigma) \mid w \succcurlyeq_d \sigma \wedge \mathrm{start}(w) \geq \mathrm{end}(\sigma)\},$$

as defined in Section 6.1.

Appendix A.1. Proof of Proposition 1 (p. 18)

We shall prove that, given a property $\varphi \subseteq \mathrm{tw}(\Sigma)$, the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, defined as per Definition 8 (p. 16), satisfies the physical constraint, is sound and transparent. These constraints are recalled below:

- **Physical constraint:**

$$\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies E_\varphi(\sigma) \preccurlyeq E_\varphi(\sigma') \qquad (\mathbf{Phy}).$$

- **Soundness:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon \qquad (\mathbf{Snd}).$$

- **Transparency:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) \lhd_d \sigma \qquad (\mathbf{Tr}).$$

The proof of **(Phy)** is straightforward by noticing that function store_φ is monotonic on its first output ($\forall \sigma, \sigma' \in \mathrm{tw}(\Sigma) : \sigma \preccurlyeq \sigma' \implies \Pi_1(\mathrm{store}_\varphi(\sigma)) \preccurlyeq \Pi_1(\mathrm{store}_\varphi(\sigma'))$).

We now prove both **(Snd)** and **(Tr)** by an induction on the length of the input timed word σ. For this purpose, we actually prove a slightly stronger property of E_φ: for any $\sigma \in \mathrm{tw}(\Sigma)$, (i) E_φ satisfies $(\mathbf{Snd})_\sigma \stackrel{\mathrm{def}}{=} E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon$ and $(\mathbf{Tr})_\sigma \stackrel{\mathrm{def}}{=} E_\varphi(\sigma) \lhd_d \sigma$, and (ii) $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$, where σ_s and σ_c are as in the definition of $\mathrm{store}_\varphi()$, recalled above.

Induction basis ($\sigma = \epsilon$). The proof of the induction basis is immediate from the definitions of E_φ, $\mathrm{store}_\varphi(\epsilon)$, $\lhd$, and $\lhd_d$.

Induction step. Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, $E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon$ $(\mathbf{Snd})_\sigma$, and $E_\varphi(\sigma) \lhd_d \sigma$ $(\mathbf{Tr})_\sigma$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t, a)$, with $t \in \mathbb{R}_{\geq 0}$, $t \geq \mathrm{end}(\sigma)$, and $a \in \Sigma$. Suppose that $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma'_c = \sigma_c \cdot (t, a)$, where $\mathrm{end}(\sigma_c) \leq t$. We distinguish two cases:

- Case $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$. In this case, we have $E_\varphi(\sigma \cdot (t,a))$ = $\Pi_1\left(\mathrm{store}_\varphi\left(\sigma \cdot (t,a)\right)\right)$ = $\sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c)$. From the definition of function κ_φ, we have $\kappa_\varphi(\sigma_s, \sigma'_c) \subseteq \sigma_s^{-1} \cdot \varphi$, and thus $E_\varphi(\sigma \cdot (t,a)) \in \varphi$. Thus E_φ satisfies $(\mathbf{Snd})_{\sigma'}$.

  From the induction hypothesis, we know that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$. We deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$ which shows that (ii) holds again for σ'.

  Let $w \in \kappa_\varphi(\sigma_s, \sigma'_c)$. From the definition of $\kappa_\varphi()$, since $w \in \sigma_s^{-1} \cdot \varphi$, we have $\mathrm{start}(w) \geq \mathrm{end}(\sigma_s)$, which implies that $\sigma_s \cdot w \in \mathrm{tw}(\Sigma)$. Since $w \in \mathrm{CanD}(\sigma'_c)$, we have $\mathrm{start}(w) \geq t$ and $w \succcurlyeq_d \sigma'_c$, which entails that $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma'_c)$. Moreover, from $\mathrm{start}(w) \geq t$, we know that all dates of the events in w are greater than or equal to those of the events in $\sigma \cdot (t,a)$. Since i) $\sigma_c \cdot (t,a) = \sigma'_c$, ii) w and σ'_c have the same untimed projection (i.e., $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma'_c)$), and iii) the concatenated untimed projections of σ_s and σ'_c form a subword of the untimed projection of $\sigma \cdot (t,a)$ (i.e., $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma'_c) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$), and hence we deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(w) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$. Thus, using $\sigma_s \lhd_d \sigma$ (from induction hypothesis), we obtain $\sigma_s \cdot w \lhd_d \sigma \cdot (t,a)$ = $E_\varphi(\sigma') \lhd_d \sigma'$, i.e., E_φ satisfies $(\mathbf{Tr})_{\sigma'}$.

- Case $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$. Note, this case encompasses the two last cases in function store_φ. From the definition of E_φ, in both cases we have $E_\varphi(\sigma \cdot (t,a))$ = $\Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t,a))\big)$ = σ_s. Since $E_\varphi(\sigma)$ = $\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big)$ = σ_s, and using the induction hypothesis $E_\varphi(\sigma) \models \varphi$, we deduce that $E_\varphi(\sigma') \models \varphi$ $(\mathbf{Snd})_{\sigma'}$.

  Moreover, $E_\varphi(\sigma \cdot (t,a)) \lhd_d \sigma$ and thus $E_\varphi(\sigma \cdot (t,a)) \lhd_d \sigma \cdot (t,a)$. We deduce $(\mathbf{Tr})_{\sigma'}$.

  Finally, from the induction hypothesis $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$, we can conclude that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$, proving (ii) for σ'.

□

Appendix A.2. Proof of Proposition 2 (p. 18)

The proof of Proposition 2 requires the following lemma related to store_φ which says that, when $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and σ_c is not the empty timed word, there is no sequence delaying a prefix of σ_c, starting after the ending date of σ, and allowing to correct σ.

Lemma 1 *Let us consider $\sigma \in \mathrm{tw}(\Sigma)$, if $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c \neq \epsilon$, then*

$$\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi.$$

PROOF. The proof is done by induction on $\sigma \in \mathrm{tw}(\Sigma)$.

Induction basis. For $\sigma = \epsilon$, we have $\sigma_c = \epsilon$ by definition of store_φ, and the induction basis holds.

Induction step. Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, if $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma_c \neq \epsilon$, then $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$ (induction hypothesis). Let us consider $\sigma \cdot (t, a) \in \mathrm{tw}(\Sigma)$, and let $(\sigma'_s, \sigma'_c) = \mathrm{store}_\varphi(\sigma \cdot (t, a))$. Following the definition of function store_φ, we distinguish three cases:

- If $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) \neq \emptyset$, then $\sigma'_c = \epsilon$, and the result holds.
- If $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$, we have $\sigma'_c = \sigma_c \neq \epsilon$. Using the induction hypothesis, if $\sigma'_c = \sigma_c \neq \epsilon$, we have: $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which implies $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma \cdot (t, a)) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which shows that the property holds again for $\sigma \cdot (t, a)$ since $\sigma'_c = \sigma_c$.
- Otherwise ($\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$ and $\kappa_{\mathrm{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t, a)) \neq \emptyset$), we have $\sigma'_c = \sigma_c \cdot (t, a)$. Using the induction hypothesis, we have: $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$, which implies $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma \cdot (t, a)) \wedge \exists v \in \mathrm{pref}(\sigma_c) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$. Since $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$, by definition we have $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma \cdot (t, a)) \wedge w \succcurlyeq_d \sigma_c \cdot (t, a)) \implies \sigma_s \cdot w \notin \varphi$. Combining both predicates, we obtain $\forall w \in \mathrm{tw}(\Sigma) : (\mathrm{start}(w) \geq \mathrm{end}(\sigma \cdot (t, a)) \wedge \exists v \in \mathrm{pref}(\sigma_c \cdot (t, a)) : w \succcurlyeq_d v) \implies \sigma_s \cdot w \notin \varphi$.

□

Let us now return to the proof of Proposition 2. We shall prove that, given a property φ, the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ as per Definition 8 (p. 16) satisfies the optimality constraint $(\mathbf{Op})$ (from Proposition 2, p. 18). That is, we shall prove that $\forall \sigma \in \mathrm{tw}(\Sigma) : (\mathbf{Op})_\sigma$, where:

$$
\begin{aligned}
(\mathbf{Op})_\sigma \stackrel{\text{def}}{=} \quad & E_\varphi(\sigma) = \epsilon \vee \exists m, w \in \mathrm{tw}(\Sigma) : E_\varphi(\sigma) = m \cdot w (\models \varphi), \text{ with} \\
& m_\sigma = \max^{\varphi}_{\prec, \epsilon}(E_\varphi(\sigma)), \text{ and} \\
& w_\sigma = \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \{ w' \in m_\sigma^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_\sigma^{-1} \cdot E_\varphi(\sigma)) \\
& \qquad \wedge m_\sigma \cdot w' \lhd_d \sigma \wedge \mathrm{start}(w') \geq \mathrm{end}(\sigma) \}
\end{aligned}
$$

The proof is done by induction on $\sigma \in \mathrm{tw}(\Sigma)$.

Induction basis. Since $\mathrm{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$ we get $E_\varphi(\epsilon) = \epsilon$.

Induction step. Let us suppose that $(\mathbf{Op})_\sigma$ holds for some $\sigma \in \mathrm{tw}(\Sigma)$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t, a)$ with $t \in \mathbb{R}_{\geq 0}$, $t \geq \mathrm{end}(\sigma)$, and $a \in \Sigma$. Let us prove that $(\mathbf{Op})_{\sigma'}$ holds. Suppose $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma'_c = \sigma_c \cdot (t, a)$. We distinguish two cases depending on whether $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$ or not:

- Case $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$. We have $E_\varphi(\sigma \cdot (t, a)) = \Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t, a))\big) = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c)$. By definition of $\kappa_\varphi(\sigma_s, \sigma'_c)$ we know that $\sigma_s \cdot \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c) \in \varphi$. From the definition of function store_φ and the induction hypothesis, we know that σ_s corresponds to $m_{\sigma'}$ in the definition of $(\mathbf{Op})_{\sigma'}$: it is the maximal strict prefix of $E_\varphi(\sigma') = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c)$ that satisfies φ. Indeed, $\mathrm{store}_\varphi(\sigma) = (\sigma_s, \sigma_c)$ and, either $\sigma_c = \epsilon$, then $E_\varphi(\sigma') = \sigma_s.(t', a)$ for some t' and σ_s is the maximal strict prefix of $E_\varphi(\sigma')$ satisfying φ; or $\sigma_c \neq \epsilon$ and using Lemma 1, we know that none of the prefixes of σ_c can be delayed in such a way that, when appended to σ_s, the concatenation forms a correct sequence.

It follows that $E_\varphi(\sigma \cdot (t,a)) = m_{\sigma'} \cdot w_{\sigma'}$ with $m_{\sigma'} = \sigma_s$ and

$$\begin{aligned} w_{\sigma'} &= \sigma_s^{-1} \cdot E_\varphi(\sigma \cdot (t,a)), \\ &= \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma'_c) \\ &= \min_{\preceq_{\text{lex}},\text{end}} \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \underbrace{\sigma_c \cdot (t,a)}_{\sigma'_c} \wedge \text{start}(w') \geq \text{end}(\sigma'_c) \right\}. \end{aligned}$$

Since $\text{end}(\sigma'_c) = t$, then

$$w_{\sigma'} = \min_{\preceq_{\text{lex}},\text{end}} \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\}.$$

We shall prove that

$$\begin{aligned} &\left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\} \\ &= \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a) \right. \\ &\qquad \left. \wedge \text{start}(w') \geq \text{end}(\sigma \cdot (t,a)) \right\}, \end{aligned}$$

that is (since $\text{end}(\sigma \cdot (t,a)) = t$):

$$\begin{aligned} &\left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid w' \succcurlyeq_d \sigma_c \cdot (t,a) \wedge \text{start}(w') \geq t \right\} \\ &= \left\{ w' \in m_{\sigma'}^{-1} \cdot \varphi \mid \Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a) \right. \\ &\qquad \left. \wedge \text{start}(w') \geq t \right\}. \end{aligned}$$

This amounts to prove that:

$$\begin{aligned} &\forall w' \in m_{\sigma'}^{-1} \cdot \varphi : \text{start}(w') \geq t \\ &\quad \Longrightarrow \big(w' \succcurlyeq_d \sigma_c \cdot (t,a) \\ &\qquad \Longleftrightarrow \big(\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a))) \wedge m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a) \big) \big). \end{aligned}$$

($\Rightarrow$) Since $\Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) = \Pi_\Sigma(\sigma_c \cdot (t,a))$, by definition of $\succcurlyeq_d$, we have $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)))$. From transparency, we know that $\sigma_s \lhd_d \sigma$ and $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$. Then, from $\text{start}(w') \geq t$, we deduce $m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a)$.

($\Leftarrow$) From $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a))$, w' and $m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a) = m_{\sigma'}^{-1} \cdot \sigma_c \cdot (t,a)$ have the same events. Moreover, since $\text{start}(w') \geq t$, all events in w' have greater dates than t (and hence, greater than those of all events in $\sigma_c \cdot (t,a)$). Thus $w' \succcurlyeq_d \sigma_c \cdot (t,a)$.

Thus, we conclude that E_φ satisfies $(\mathbf{Op})_{\sigma'}$.

- Case $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$. We have $E_\varphi(\sigma \cdot (t,a)) = \Pi_1\left(\text{store}_\varphi(\sigma \cdot (t,a))\right) = \Pi_1\big(\text{store}_\varphi(\sigma)\big) = \sigma_s = E_\varphi(\sigma)$. Thus, from the induction hypothesis, we deduce that $(\mathbf{Op})_{\sigma'}$ holds.

□

Appendix A.3. Preliminaries to the proof of Proposition 3 (p. 31): characterising the configurations of enforcement monitors

We define some notions and lemmas related to the configurations of any enforcement monitor $\mathcal{E}$.

Remark 9. In the following proofs, without loss of generality, we assume that at any date, in addition to rule ***idle***, at most one of the ***store*** and ***release*** rules of the enforcement monitor applies. This simplification does not come at the price of reducing the generality nor the validity of the proofs because i) rules ***store*** and ***release*** of the enforcement monitor do not rely on the same conditions, and ii) the ***store*** and ***release*** operations of enforcement monitors are assumed to be executed in zero time. The considered simplification however reduces the number of (equivalent) cases in the following proofs.

Remark 10. Between the occurrences of two (input or output) events, the configuration of the enforcement monitor evolves according to rule ***idle*** (since it is the rule with lowest priority). Moreover, from any configuration, applying ***idle*** twice consecutively each delaying for δ_1 and δ_2, or applying ***idle*** once from the same configuration, with delay $\delta_1 + \delta_2$ will result in the same configuration. To simplify notations we will use a rule to simplify the representation of $\mathcal{E}^{\mathrm{ioo}} \in \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ stating that

$$\sigma \cdot (\epsilon, \mathrm{idle}(\delta_1), \epsilon) \cdot (\epsilon, \mathrm{idle}(\delta_2), \epsilon) \cdot \sigma' \text{ is equivalent to } \sigma \cdot (\epsilon, \mathrm{idle}(\delta_1 + \delta_2), \epsilon) \cdot \sigma',$$

for any $\sigma, \sigma' \in \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ and $\delta_1, \delta_2 \in \mathbb{R}_{\geq 0}$. Thus, for $\mathcal{E}^{\mathrm{ioo}}$, we will only consider sequences of $\big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ where delays appearing in operation idle are maximal (i.e., there is no sequence of two consecutive events with an idle operation).

Appendix A.3.1. Some notations

Based on the assumption stated in Remark 9, there are at most two configurations for each date. Let us define the two functions $\mathrm{config}_{\mathrm{in}}, \mathrm{config}_{\mathrm{out}} : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to C^{\varepsilon}$ that give respectively the first and last configurations of an enforcement monitor at some time instant, reading an input sequence. More formally, given some $\sigma \in \mathrm{tw}(\Sigma), t \in \mathbb{R}_{\geq 0}$:

- $\mathrm{config}_{\mathrm{in}}(\sigma, t) = c_\sigma^t$ such that $c_0^{\mathcal{E}} \overset{w(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}} c_\sigma^t$ where $w(\sigma, t) \overset{\mathrm{def}}{=} \min_{\preceq}\{w \preceq \mathcal{E}^{\mathrm{ioo}}(\sigma, t) \mid \mathrm{timeop}(w) = t\}$;
- $\mathrm{config}_{\mathrm{out}}(\sigma, t) = c_\sigma^t$ such that $c_0^{\mathcal{E}} \overset{\mathcal{E}^{\mathrm{ioo}}(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}} c_\sigma^t$.

Observe that, when at some date, only rule ***idle*** applies, $\mathrm{config}_{\mathrm{in}}(\sigma, t) = \mathrm{config}_{\mathrm{out}}(\sigma, t)$ holds, because there is only one configuration at this date. Moreover, when at some date, other rules apply (rules ***release*** or ***store***), $\mathrm{config}_{\mathrm{in}}(\sigma, t)$ and $\mathrm{config}_{\mathrm{out}}(\sigma, t)$ differ. Note, in all cases, from $\mathrm{config}_{\mathrm{out}}(\sigma, t)$ only rule ***idle*** applies (which increases time).

Moreover, for any $\sigma \in \mathrm{tw}(\Sigma)$, for any two $t, t' \in \mathbb{R}_{\geq 0}$ such that $t \leq t'$, we note $\mathcal{E}(\sigma, t, t')$ for $\mathcal{E}(\sigma, t)^{-1} \cdot \mathcal{E}(\sigma, t')$, i.e., the output sequence of an enforcement monitor between t and t'. Note that, when $t = t'$, we have $\mathcal{E}(\sigma, t, t') = \epsilon$, for any $\sigma \in \mathrm{tw}(\Sigma)$.

The following remark states that configurations keep track of global time, and is a direct consequence of the rules of enforcement monitors in Definition 10 (p. 26).

Remark 11 (Value of the third component of configurations.). Only rule ***idle*** modifies the value of the third component of configurations: it increments the third component as time elapses. That is:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \Pi_3\left(\mathrm{config}_{\mathrm{in}}\left(\sigma, t\right)\right) = \Pi_3(\mathrm{config}_{\mathrm{out}}(\sigma, t)) = t.$$

Appendix A.3.2. Some intermediate lemmas

Before tackling the proof of Proposition 3, we give a list of lemmas that describe the behaviour of an enforcement monitor, describing the configurations or the output at some particular date for some input and memory content.

Similarly to the first physical constraint, the following lemma states that the enforcement monitor cannot change what it has output. More precisely, when the enforcement monitor is seen as function $\mathcal{E}$, the output is monotonic w.r.t. $\preceq$.

Lemma 2 (Monotonicity of enforcement monitors) *Function* $\mathcal{E} : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \mathrm{tw}(\Sigma)$ *is monotonic in its second parameter:*

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t, t' \in \mathbb{R}_{\geq 0} : t \leq t' \implies \mathcal{E}(\sigma, t) \preceq \mathcal{E}(\sigma, t').$$

The lemma states that for any input sequence σ, if we consider two dates t, t' such that $t \leq t'$, then the output of the enforcement monitor at date t is a prefix of the output at date t'.

PROOF (OF LEMMA 2). The proof directly follows from the definitions of the function $\mathcal{E}$ associated to an enforcement monitor (see Section 7.4, p. 28) which directly depends on $\mathcal{E}^{\mathrm{ioo}}$, which is itself monotonic over time (because of the definition of enforcement monitors). □

As a consequence, one can naturally split the output of the enforcement monitor over time, as it is stated by the following corollary.

Lemma 3 (Separation of the output of the enforcement monitor over time)

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t_1, t_2, t_3 \in \mathbb{R}_{\geq 0} : t_1 \leq t_2 \leq t_3 \implies \mathcal{E}(\sigma, t_1, t_3) = \mathcal{E}(\sigma, t_1, t_2) \cdot \mathcal{E}(\sigma, t_2, t_3).$$

The lemma states that for any sequence σ input to $\mathcal{E}$, if we consider three dates $t_1, t_2, t_3 \in \mathbb{R}_{\geq 0}$ such that $t_1 \leq t_2 \leq t_3$, the output of $\mathcal{E}$ between t_1 and t_3 is the concatenation of the output between t_1 and t_2 and the output between t_2 and t_3.

PROOF (OF LEMMA 3). Recall that for any $t, t' \in \mathbb{R}_{\geq 0}$ such that $t \leq t'$, $\mathcal{E}(\sigma, t, t')$ is the output sequence of an enforcement monitor between t and t'. The lemma directly follows from the definition of $\mathcal{E}(\sigma, t, t') = \mathcal{E}(\sigma, t)^{-1} \cdot \mathcal{E}(\sigma, t')$. □

The following lemma states that, at some date t, the output of the enforcement monitor only depends on what has been observed until date t. In other words, the enforcement monitor works in an online fashion.

Lemma 4 (Dependency of the output on the observation only)

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathcal{E}(\sigma, t) = \mathcal{E}(\mathrm{obs}(\sigma, t), t).$$

PROOF (OF LEMMA 4). The proof of the lemma directly follows from the definitions of $\mathcal{E}^{\mathrm{ioo}}$ (Definition 11, p. 30) and obs (in Section 3). Indeed, using $\mathrm{obs}(\sigma, t) = \mathrm{obs}(\mathrm{obs}(\sigma, t), t)$, we deduce that $\mathcal{E}^{\mathrm{ioo}}(\sigma, t) = \mathcal{E}^{\mathrm{ioo}}(\mathrm{obs}(\sigma, t), t)$, for any $\sigma \in \mathrm{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$. Using $\mathcal{E}(\sigma, t) = \Pi_3(\mathcal{E}^{\mathrm{ioo}}(\sigma, t))$, we can deduce the expected result. □

The following lemma states that after reading some input sequence σ entirely, only the memory content σ_{ms} and the value of the clock t influence the output of the enforcement monitor. More specifically, after completely reading some sequence, if an enforcement monitor reaches some configuration containing σ_{ms} in its memory, its future output is fully determined by the memory content σ_{ms} (containing the corrected sequence) and the value of clock variable t, during the total time needed to output it.

Lemma 5 (Values of $\mathrm{config}_{\mathrm{out}}$ when releasing events)

$$\begin{aligned}
&\forall \sigma, \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0}, \forall q \in Q : \\
&\quad t \geq \mathrm{end}(\sigma) \wedge \mathrm{config}_{\mathrm{out}}(\sigma, t) = (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \\
&\quad \implies \forall \sigma'_{\mathrm{ms}} \preceq \sigma_{\mathrm{ms}} : \mathrm{config}_{\mathrm{out}}(\sigma, \mathrm{end}(\sigma'_{\mathrm{ms}})) = (\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'_{\mathrm{ms}}), q, t_F).
\end{aligned}$$

The lemma states that, whatever is the output configuration $(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F)$ reached by reading some input sequence σ at some date $t \geq \mathrm{end}(\sigma)$, then for any prefix σ'_{ms} of σ_{ms}, the output configuration reached at time $\mathrm{end}(\sigma'_{\mathrm{ms}})$ (output date of the last event in σ'_{ms}) is such that σ'_{ms} has been released from the memory (the memory is thus $\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}$) and the clock value in this configuration is $\mathrm{end}(\sigma'_{\mathrm{ms}})$.

PROOF (OF LEMMA 5). The proof is a straightforward induction on the length of σ'_{ms}. It uses the fact that the considered configurations occur at dates greater than $\mathrm{end}(\sigma)$, hence implying that no input event can be read any more. Consequently, following the definition of the enforcement monitor (Definition 10, p. 26), on the configurations of the enforcement monitor, only rules ***idle*** and ***release*** apply. Between $\mathrm{end}(\sigma'_{\mathrm{ms}})$ and $\mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t,a))$ where $\sigma'_{\mathrm{ms}} \preceq \sigma'_{\mathrm{ms}} \cdot (t,a) \preceq \sigma_{\mathrm{ms}}$, the configuration of the enforcement monitor evolves only using rule ***idle*** (no other rule applies) until $\mathrm{config}_{\mathrm{in}}(\sigma, \mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t,a))) = (\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t,a)), q, t_F)$. Rule ***release*** is then applied to get the following derivation $(\sigma'^{-1}_{\mathrm{ms}} \cdot \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t,a)), q) \overset{\epsilon/\mathrm{release}(t,a)/\epsilon}{\hookrightarrow} ((\sigma'_{\mathrm{ms}} \cdot (t,a))^{-1} \cdot \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t,a)), q, t_F)$.

The following lemma relates the date of the last event of the corrected sequence and the value of the last variable stored in the configuration of an enforcement monitor.

Lemma 6 (Relation between some elements in a configuration)

$$\forall \sigma, \sigma_{\mathrm{ms}} \in \mathrm{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0} : \mathrm{config}_{\mathrm{out}}(\sigma, t) = (\sigma_{\mathrm{ms}}, \_, t, \_, t_F) \wedge \sigma_{\mathrm{ms}} \neq \epsilon \implies \mathrm{end}(\sigma_{\mathrm{ms}}) = t_F.$$

PROOF. The lemma is a straightforward consequence of the definition of enforcement monitors (Definition 10, p. 26). Indeed, only rule *store-*φ modifies these elements of a configuration, and it performs it as expected.

The following lemma states that when an enforcement monitor has nothing to read in input anymore, what it releases as output is the observation of its memory content over time.

Lemma 7 (Output of the enforcement monitor according to memory content)

$$\begin{array}{l} \forall \sigma, \sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma), \forall t, t_F \in \mathbb{R}_{\geq 0}, \forall q \in Q : \\ \quad t \geq \mathrm{end}(\sigma) \wedge \mathrm{config}_{\mathrm{out}}(\sigma, t) = (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t, q, t_F) \\ \qquad \implies \forall t' \in \mathbb{R}_{\geq 0} : t \leq t' \leq \mathrm{end}(\sigma_{\mathrm{ms}}) \implies \mathcal{E}(\sigma, t, t') = \mathrm{obs}(\sigma_{\mathrm{ms}}, t'). \end{array}$$

The lemma states that, if after some date t, after reading an input sequence σ, the enforcement monitor is in an output configuration that contains σ_{ms} as a memory content, whatever is the date t' between t and $\mathrm{end}(\sigma_{\mathrm{ms}})$, the output of the enforcement monitor between t and t' is the observation of σ_{ms} with t' time units.

PROOF (OF LEMMA 7). The proof is performed by induction on the length of σ_{ms} and uses Lemma 5.

- Case $|\sigma_{\mathrm{ms}}| = 0$. In this case, $\sigma_{\mathrm{ms}} = \epsilon$ and $\mathrm{end}(\epsilon) = 0$. If $t = t' = 0$, we have $\mathcal{E}(\sigma, t, t') = \epsilon = \mathrm{obs}(\sigma_{\mathrm{ms}}, t')$. Otherwise, $t \leq t'$ does not hold, and thus the lemma vacuously holds.
- Induction case. Let us suppose that the lemma holds for all prefixes of σ_{ms} of some maximum length $n \in [0, |\sigma_{\mathrm{ms}}|-1]$ (induction hypothesis). Following Lemma 6, one can consider $\sigma_{\mathrm{ms}} = \sigma' \cdot (t_F, a)$ where σ' is the prefix of σ_{ms} of length n, and $(t_F, a) \in \mathbb{R}_{\geq 0} \times \Sigma$. On the one hand, at date $\mathrm{end}(\sigma')$, according to Lemma 5, we have $\mathrm{config}_{\mathrm{out}}(\sigma, \mathrm{end}(\sigma')) = ((t_F, a), \sigma_{\mathrm{mc}}, \mathrm{end}(\sigma'), q, t_F)$ for some $\sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma)$ and $q \in Q$. For any $t' \leq \mathrm{end}(\sigma')$, the lemma vacuously holds. On the other hand, let us consider some $t' \in [\mathrm{end}(\sigma'), t_F]$, we have:

$$\mathcal{E}(\sigma, t, t') = \mathcal{E}(\sigma, t, \mathrm{end}(\sigma')) \cdot \mathcal{E}(\sigma, \mathrm{end}(\sigma'), t').$$

(Note, when $t = t' = \mathrm{end}(\sigma')$, the above equation reduces to $\epsilon = \epsilon$.) Using the induction hypothesis, we find $\mathcal{E}(\sigma, t, \mathrm{end}(\sigma')) = \mathrm{obs}(\sigma', \mathrm{end}(\sigma')) = \sigma'$. Using the semantics of the enforcement monitor (only rules ***release*** and ***idle*** apply, no new event is received), we obtain $\mathcal{E}(\sigma, \mathrm{end}(\sigma'), t') = \mathrm{obs}((t_F, a), t')$. Thus, $\mathcal{E}(\sigma, t, t') = \sigma' \cdot \mathrm{obs}((t_F, a), t') = \mathrm{obs}(\sigma' \cdot (t_F, a), t')$.

□

The following lemma states that, for any input σ, after observing the entire input (that is, at any date greater than or equal to $\mathrm{end}(\sigma)$), the content of the internal memory (σ_c) of the enforcement function and the enforcement monitor are the same.

Lemma 8 (Content of the internal memory)

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : t \geq \mathrm{end}(\sigma) \implies \Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t)).$$

PROOF (OF LEMMA 8). The proof is performed by induction on the length of σ. Recall that $\mathrm{store}_\varphi(\sigma)$ is defined in Section 6.2, and $\mathrm{config}_{\mathrm{out}}(\sigma, t)$ is defined in Appendix A.3.1.

- Case $|\sigma| = 0$. In this case, from the definition of the enforcement monitor (Definition 10, p. 26), none of the store rules can be applied. Consequently, we have $\Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t)) = \epsilon$. Regarding the enforcement function, as per Definition 8 (p. 16), we have $\Pi_2(\mathrm{store}_\varphi(\epsilon)) = \epsilon$.

- Induction case. Let us suppose that for some $\sigma \in \mathrm{tw}(\Sigma)$, we have $\forall t \in \mathbb{R}_{\geq 0} : t \geq \mathrm{end}(\sigma) \implies \Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$ (induction hypothesis). Let us consider $\sigma' = \sigma \cdot (t_l, a)$, where $(t_l, a) \in \mathbb{R}_{\geq 0} \times \Sigma$.

  From the induction hypothesis, for $t \geq \mathrm{end}(\sigma)$, we have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$, and therefore, for any $t \geq t_l$, we also have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma, t))$. Let $\sigma_c = \Pi_2(\mathrm{store}_\varphi(\sigma))$. Consequently, we also have $\mathrm{config}_{\mathrm{in}}(\sigma, t_l) = (\_, \sigma_c, t_l, \_, \_) = \mathrm{config}_{\mathrm{in}}(\sigma \cdot (t_l, a), t_l)$.

  From the definition of store_φ, we have $\Pi_2(\mathrm{store}_\varphi(\sigma \cdot (t_l, a))) = \sigma'_c$, where σ'_c is either ϵ, $\sigma_c \cdot (t_l, a)$, or σ_c depending on which case of the store_φ function applies.

  Regarding the enforcement monitor, from the update function (since each case in store_φ has a corresponding case in update), we also have $\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t_l) = (\_, \sigma'_c, t_l, \_, \_)$ (which is obtained by applying one of the store rules based on the value returned by function update). For $t > t_l$, since none of the store rules can be applied, we can conclude that $\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t) = (\_, \sigma'_c, t, \_, \_)$.

  Thus, we have $\Pi_2(\mathrm{store}_\varphi(\sigma \cdot (t_l, a))) = \Pi_2(\mathrm{config}_{\mathrm{out}}(\sigma \cdot (t_l, a), t))$.

□

Appendix A.4. Proof of Proposition 3: relation between enforcement function and enforcement monitor

We shall prove that, given a property φ, the associated enforcement monitor $\mathcal{E}_\varphi$ as per Definition 10 (p. 26) implements the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ as per Definition 8 (p. 16). That is:

$$\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\sigma, t).$$

The proof is done by induction on the length of the input timed word σ.

Induction Basis. Let us suppose that $|\sigma| = 0$, thus $\sigma = \epsilon$ in $\mathrm{tw}(\Sigma)$. On the one hand, we have $E_\varphi(\sigma) = \epsilon$, and thus $\forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \epsilon$. On the other hand, the word $\mathcal{E}_\varphi^{\mathrm{ioo}}(\epsilon, t)$ over the input-operation-output alphabet is such that $\forall t \in \mathbb{R}_{\geq 0} : \Pi_1(\mathcal{E}_\varphi^{\mathrm{ioo}}(\epsilon, t)) = \epsilon$. Thus, according to the definition of the enforcement monitor, the rules ***store-***φ, ***store***$_{\sup}$***-***$\overline{\varphi}$, and ***store-***$\overline{\varphi}$ cannot be applied. Consequently, the memory of the enforcement monitor σ_{ms} remains empty as in the initial configuration. It follows that rule ***release*** cannot be applied as well. We have then $\forall t \in \mathbb{R}_{\geq 0} : c_0^{\mathcal{E}_\varphi} \overset{\epsilon/\mathrm{idle}(t)/\epsilon}{\hookrightarrow}_{\mathcal{E}_\varphi} (\epsilon, \epsilon, t, q_0, 0)$, and thus $\mathcal{E}_\varphi(\epsilon, t) = \epsilon$. Thus, $\forall t \in \mathbb{R}_{\geq 0} : \mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\epsilon, t)$.

Induction Step. Let us suppose that $\mathrm{obs}(E_\varphi(\sigma), t) = \mathcal{E}_\varphi(\sigma, t)$ for any timed word $\sigma \in \mathrm{tw}(\Sigma)$ of some length $n \in \mathbb{N}$, at any date $t \in \mathbb{R}_{\geq 0}$ (induction hypothesis). Let us now consider some input timed word $\sigma \cdot (t_{n+1}, a)$ for some $\sigma \in \mathrm{tw}(\Sigma)$ with $|\sigma| = n$, $t_{n+1} \in \mathbb{R}_{\geq 0}$, and $a \in \Sigma$. We want to prove that $\mathrm{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$, at any date $t \in \mathbb{R}_{\geq 0}$.

Let us consider some date $t \in \mathbb{R}_{\geq 0}$. Note that $\mathrm{end}(\sigma \cdot (t_{n+1}, a)) = t_{n+1}$. We distinguish two cases according to whether $t_{n+1} > t$ or not, that is whether $\sigma \cdot (t_{n+1}, a)$ is completely observed or not at date t.

- Case $t_{n+1} > t$. In this case, $\mathrm{obs}\left(\sigma \cdot (t_{n+1}, a), t\right) = \mathrm{obs}(\sigma, t)$, i.e., at date t, the observations of σ and $\sigma \cdot (t_{n+1}, a)$ are identical.

  On the one hand, from the definition of E_φ (since function store_φ and the delayed subsequence are defined such that the date of each event in output is greater than or equal to the date of the corresponding event in the input), we have:

$$\begin{aligned} \mathrm{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) &= \mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a))\right), t\right) \\ &= \mathrm{obs}\left(\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big), t\right) \\ &= \mathrm{obs}\left(E_\varphi\left(\sigma\right), t\right). \end{aligned}$$

  On the other hand, regarding the enforcement monitor, since $\mathrm{obs}\left(\sigma \cdot (t_{n+1}, a), t\right) = \mathrm{obs}(\sigma, t)$, using Lemma 4 (p. 51), we obtain $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}_\varphi(\sigma, t)$. Using the induction hypothesis, we can conclude that $\mathrm{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) = \mathcal{E}_\varphi\left(\sigma \cdot (t_{n+1}, a), t\right)$.

- Case $t_{n+1} \leq t$. In this case, we have $\mathrm{obs}(\sigma \cdot (t_{n+1}, a), t) = \sigma \cdot (t_{n+1}, a)$ (i.e., $\sigma \cdot (t_{n+1}, a)$ is observed entirely at date t). From Remark 11 (p. 50), we know that the configuration of the enforcement monitor at date $\mathrm{end}(\sigma \cdot (t_{n+1}, a))$ is $\mathrm{config}_{\mathrm{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right) = (\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t_{n+1}, q_\sigma, t_F)$ for some $\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}} \in \mathrm{tw}(\Sigma)$, $q_\sigma \in Q$, $t_F \in \mathbb{R}_{\geq 0}$. Using Lemma 8 (p. 53), we also have $\Pi_2(\mathrm{store}_\varphi(\sigma)) = \sigma_c = \Pi_2(\mathrm{config}_{\mathrm{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right)) = \sigma_{\mathrm{mc}}$. Observe that $\mathrm{config}_{\mathrm{in}}\big(\sigma, t_{n+1}\big) = \mathrm{config}_{\mathrm{in}}\big(\sigma \cdot (t_{n+1}, a), t_{n+1}\big)$ because of i) the definition of $\mathrm{config}_{\mathrm{in}}$ using the definition of $\mathcal{E}_\varphi^{\mathrm{ioo}}$ and ii) the event (t_{n+1}, a) has not been yet consumed through any of the ***store*** rules by the enforcement monitor at date t_{n+1}.

  We distinguish two cases according to whether $\sigma_c \cdot (t_{n+1}, a)$ can be delayed into a word satisfying φ or not, i.e., whether $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) = \emptyset$, or not.

  - Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) = \emptyset$. From the definition of function store_φ, we have $\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a)) = (\sigma_s, \sigma_c')$, and $\Pi_1\big(\mathrm{store}_\varphi(\sigma \cdot (t_{n+1}, a))\big) = \sigma_s$. We also have $\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big) = \sigma_s$. From the definition of E_φ and obs, we have $\mathrm{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \mathrm{obs}(E_\varphi(\sigma), t)$. Regarding $\mathcal{E}_\varphi$, according to the definition of function update, we have $\mathrm{update}(q_\sigma, t_F, \sigma_{\mathrm{mc}}, (t_{n+1}, a)) = (q_\sigma, \sigma_{\mathrm{mc}}, \texttt{bad})$ or $(q_\sigma, \sigma_{\mathrm{mc}} \cdot (t_{n+1}, a), \texttt{c\_bad})$. According to the definition of the transition relation, we have:

$$(\sigma_{\mathrm{ms}}, \sigma_{\mathrm{mc}}, t_{n+1}, q_\sigma, t_F) \overset{(t_{n+1}, a)/\mathrm{store}-\overline{\varphi}(t_{n+1}, a)/\epsilon}{\hookrightarrow}_{\mathcal{E}_\varphi} (\sigma_{\mathrm{ms}}, \sigma'_{\mathrm{mc}}, t_{n+1}, q_\sigma, t_F).$$

where, $\sigma'_{\text{mc}} = \sigma_{\text{mc}}$ if $\text{update}(q_\sigma, t_F, \sigma_{\text{mc}}, (t_{n+1}, a)) = (q_\sigma, \sigma_{\text{mc}}, \texttt{bad})$, and $\sigma'_{\text{mc}} = \sigma_{\text{mc}} \cdot (t_{n+1}, a)$ otherwise. Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$.

Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, $\text{config}(t)$ is such that only the rule ***idle*** applies.

Let us examine $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$. We have:

$$\begin{aligned}\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \quad & \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) \\ & \cdot\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) \\ & \cdot\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t).\end{aligned}$$

Let us examine $\mathcal{E}_\varphi(\sigma, t)$. We have:

$$\mathcal{E}_\varphi(\sigma, t) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon, t_{n+1}) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1}, t).$$

Observe that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon)$ because $\text{obs}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \sigma$ according to the definition of obs. Moreover, $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) = \epsilon$ since only rule ***idle*** applies during the considered time interval. Furthermore, according to Lemma 7, since $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$, we get $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$. Moreover, we know that $\text{config}_{\text{in}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F)$. Since the enforcement monitor is deterministic, and from Remark 9 (p. 50), we also get that $\text{config}_{\text{out}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1},\ q_\sigma, t_F)$. Using Lemma 7 (p. 52) again, we get $\mathcal{E}_\varphi(\sigma, t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$.

Consequently we can deduce that $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}_\varphi(\sigma, t) = \text{obs}(E_\varphi(\sigma),\ t) = \text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t)$.

– Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) \neq \emptyset$. Regarding E_φ, from the definition of function store_φ, we have $\text{store}_\varphi(\sigma \cdot (t_{n+1}, a)) = (\sigma_s \cdot \min_{\preceq_{\text{lex}}, \text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)), \epsilon)$, and $\Pi_1\big(\text{store}_\varphi(\sigma \cdot (t_{n+1}, a))\big) = \sigma_s \cdot \min_{\preceq_{\text{lex}}, \text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a))$. Regarding the enforcement monitor, according to the definition of update, we have $\text{update}(q_\sigma, \sigma_{\text{mc}}, (t_{n+1}, a), t_F) = (q', w, \texttt{ok})$ with $w = \min_{\preceq_{\text{lex}}, \text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a))$, since, $\sigma_c = \sigma_{\text{mc}}$ and from the definition of κ_φ and update, the dates computed for $\sigma_c \cdot (t_{n+1}, a)$ by both these functions are equal. From the definition of the transition relation, we have:

$$(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma, t_F) \overset{(t_{n+1}, a)/\text{store} - \varphi(t_{n+1}, a)/\epsilon}{\hookrightarrow_{\mathcal{E}_\varphi}} (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q', \text{end}(w)),$$

Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q', \text{end}(w))$.

Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, $\text{config}(t)$ is such that only rule ***idle*** applies.

Let us examine $\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t)$. We have:

$$\begin{aligned}\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t) = \quad & \mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) \\ & \cdot\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) \\ & \cdot\mathcal{E}_\varphi(\sigma \cdot (t_{n+1}, a), t_{n+1}, t).\end{aligned}$$

Let us examine $\mathcal{E}_\varphi(\sigma, t)$. We have:

$$\mathcal{E}_\varphi(\sigma, t) = \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1} - t_\epsilon, t_{n+1}) \cdot \mathcal{E}_\varphi(\sigma, t_{n+1}, t).$$

Observe that $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t_{n+1}-t_\epsilon)=\mathcal{E}_\varphi(\sigma,t_{n+1}-t_\epsilon)$ because $\mathrm{obs}(\sigma\cdot(t_{n+1},a),t_{n+1}-t_\epsilon)=\sigma$ according to the definition of obs. Moreover, $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t_{n+1}-t_\epsilon,t_{n+1})=\epsilon$ since only rule ***idle*** applies during the considered time interval.

Furthermore, according to Lemma 7 (p. 52), since $\mathrm{config}_{\mathrm{out}}(\sigma\cdot(t_{n+1},a),t_{n+1})=(\sigma_{\mathrm{ms}}\cdot w,\epsilon,t_{n+1},q',\mathrm{end}(w))$, we get $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t_{n+1},t)=\mathrm{obs}(\sigma_{\mathrm{ms}}\cdot w,t)$.

Now we further distinguish two more sub-cases, based on whether $\mathrm{end}(\sigma_{\mathrm{ms}}\cdot w)=\mathrm{end}(w)>t$ or not (whether all the elements in the memory can be released as output by date t or not).

* Case $\mathrm{end}(w)>t$.
  We further distinguish two more sub-cases based on whether $\mathrm{end}(\sigma_{\mathrm{ms}})>t$, or not.
  - Case $\mathrm{end}(\sigma_{\mathrm{ms}})>t$. In this case, we know that $\mathrm{obs}(\sigma_{\mathrm{ms}}\cdot w,t)=\mathrm{obs}(\sigma_{\mathrm{ms}},t)$. Hence, we can derive that $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)=\mathcal{E}_\varphi(\sigma,t)$. Also, from the induction hypothesis, we know that $\mathcal{E}_\varphi(\sigma,t)=\mathrm{obs}(E_\varphi(\sigma),t)$.
    Regarding enforcement function E_φ, we have

    $$\mathrm{store}_\varphi(\sigma\cdot(t_{n+1},a))=\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)).$$

    Moreover,

    $$\begin{aligned}&\mathrm{obs}\left(E_\varphi\left(\sigma\cdot(t_{n+1},a)\right),t\right)\\&\quad=\mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi\left(\sigma\cdot(t_{n+1},a)\right)\right),t\right)\\&\quad=\mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi\left(\sigma\right)\right)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)),t\right).\end{aligned}$$

    One can have

    $$\mathrm{obs}\left(E_\varphi\left(\sigma\cdot(t_{n+1},a)\right),t\right)=\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big)\cdot o,$$

    where $o\preccurlyeq\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))$, which is equal to $\mathrm{obs}(E_\varphi(\sigma),t)\cdot o$, only if the dates computed by the update function are different from the dates computed by E_φ. This would violate the induction hypothesis stating that $\mathcal{E}_\varphi(\sigma,t)=\mathrm{obs}(E_\varphi(\sigma),t)$. Hence, we have $\mathrm{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi\left(\sigma\right)\right),t\right)=\mathrm{obs}(E_\varphi(\sigma),t)$. Thus, $\mathrm{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)$.
  - Case $\mathrm{end}(\sigma_{\mathrm{ms}})\leq t$. In this case, we can follow the same reasoning as in the previous case to obtain the expected result.
* Case $\mathrm{end}(w)\leq t$.
  In this case, similarly following Lemma 7 (p. 52), we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t_{n+1},t)=\mathrm{obs}(\sigma_{\mathrm{ms}}\cdot w,t)=\sigma_{\mathrm{ms}}\cdot w$. We can also derive that $\mathcal{E}_\varphi(\sigma,t_{n+1},t)=\sigma_{\mathrm{ms}}$. Consequently, we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)=\mathcal{E}_\varphi(\sigma,t)\cdot w$. From the induction hypothesis, we know that $\mathrm{obs}(E_\varphi(\sigma),t)=\mathcal{E}_\varphi(\sigma,t)$, and we have $\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)=\mathrm{obs}(E_\varphi(\sigma),t)\cdot w$. Moreover, we have

  $$\mathrm{store}_\varphi(\sigma\cdot(t_{n+1},a))=\Pi_1\big(\mathrm{store}_\varphi(\sigma)\big)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)),$$

  and thus

  $$\begin{aligned}&\mathrm{obs}\left(E_\varphi\left(\sigma\cdot(t_{n+1},a)\right),t\right)\\&\quad=\mathrm{obs}\left(\Pi_1\left(\mathrm{store}_\varphi\left(\sigma\right)\right)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a)),t\right).\end{aligned}$$

  Henceforth, we have $\mathrm{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathrm{store}_\varphi(\sigma)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))=E_\varphi(\sigma)\cdot\min_{\preceq_{\mathrm{lex}},\mathrm{end}}\kappa_\varphi(\sigma_s,\sigma_c\cdot(t_{n+1},a))$, since, $\sigma_c=\sigma_{\mathrm{mc}}$ and from the definition of κ_φ and update, we know the dates computed for the subsequence $\sigma_c\cdot(t_{n+1},a)$ by E_φ and $\mathcal{E}_\varphi$ are equal. Finally, we have $\mathrm{obs}(E_\varphi(\sigma\cdot(t_{n+1},a)),t)=\mathcal{E}_\varphi(\sigma\cdot(t_{n+1},a),t)$.

□