Noname manuscript No.
(will be inserted by the editor)

Predictive Runtime Enforcement

Srinivas Pinisetty · Viorel Preoteasa · Stavros Tripakis · Thierry Jéron · Yliès Falcone · Hervé Marchand

Received: date / Accepted: date

Abstract Runtime enforcement (RE) is a technique to ensure that the (untrustworthy) output of a black-box system satisfies some desired properties. In RE, the output of the running system, modeled as a sequence of events, is fed into an enforcer. The enforcer ensures that the sequence complies with a certain property, by delaying or modifying events if necessary. This paper deals with *predictive* runtime enforcement, where the system is not entirely black-box, but we know something about its behavior. This *a priori* knowledge about the system allows to output some events immediately, instead of delaying them until more events are observed, or even blocking them permanently. This in turn results in better enforcement policies. We also show that if we have no knowledge about the system, then the proposed enforcement mechanism reduces to standard (non-predictive) runtime enforcement. All our results related to predictive RE of untimed properties are also formalized and proved in the Isabelle theorem prover. We also discuss how our predictive runtime enforcement framework can be extended to enforce timed properties.

This work was supported in part by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138). Thierry Jéron and Yliès Falcone acknowledge the support of the COST Action ARVI IC1402, which is supported by COST (European Cooperation in Science and Technology).

S. Pinisetty,
Aalto University, Finland,
Tel.: +358 504412405,
E-mail: srinu85.pinisetty@gmail.com

V. Preoteasa,
Aalto University, Finland,
E-mail: viorel.preoteasa@aalto.fi

S.Tripakis
Aalto University, Finland and University of California, Berkeley,
E-mail: stavros.tripakis@aalto.fi

T. Jéron, H. Marchand
INRIA Rennes - Bretagne Atlantique, France,
E-mail: First.Last@inria.fr

Y. Falcone
Univ. Grenoble Alpes, Inria, Laboratoire d'Informatique de Grenoble, Grenoble, France,
E-mail: ylies.falcone@imag.fr

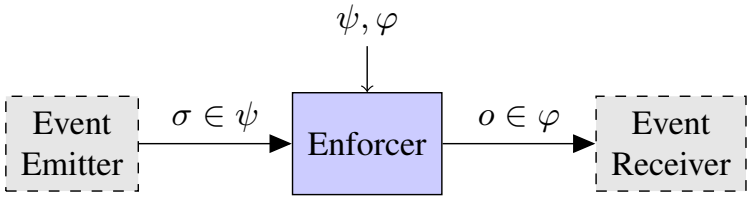


Fig. 1: Predictive runtime enforcement with an enforcer

Keywords runtime monitoring · runtime enforcement · automata · timed automata · monitor synthesis

1 Introduction

Runtime enforcement (RE) is a technique [25, 10, 13] to monitor the execution of a system at runtime and ensure its compliance against a set of formal requirements. Using an enforcer, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies with a property (e.g., a safety requirement). RE aims to ensure the following properties: (i) the output sequence must comply with the property (soundness) and (ii) if the input already complies with the property, it should be left unchanged (*transparency*).

Context and objectives. We focus on *online enforcement of regular properties*. For a given property φ, we synthesize an enforcer that operates at runtime. The general context is depicted in Fig. 1, where an enforcer is placed between an event emitter and an event receiver. The emitter and receiver execute asynchronously. An enforcer takes a sequence of events σ as input and transforms it into a sequence of events o that is correct with respect to property φ. The enforcer is equipped with an internal memory to store some events that are received as input (and to release them as output only when some expected event occurs). For example, consider a property formalizing some desired transactional behavior. Then, the enforcer stores and delays some input events (not releasing them immediately) as long as the transaction is not properly completed.

In existing RE mechanisms (cf. [25, 10, 13, 19]) there is no assumption on the input sequence σ, which can be any sequence of events over some alphabet Σ, i.e., $\sigma \in \Sigma^*$. This can be seen as considering the event emitter to be a *black-box*, i.e., its behavior is completely unknown. In this paper, we study RE for *grey box* systems, i.e., when we know something about the behavior of the event emitter. In particular, instead of letting σ range over Σ^*, we suppose that it ranges over some given property $\psi \subseteq \Sigma^*$.

For example, in the domain of network security, one can use an enforcer as a firewall or a Network Intrusion Detection (NID) system to detect and prevent some attacks. Some network flows may not be interpreted in the same manner at different end-points, and may deceive firewalls and NID's. TCP/IP scrubber eliminates ambiguities from network flows enabling firewall systems to correctly predict end-host response [14]. The knowledge of the system ψ can be considered as a protocol scrubber such as a TCP/IP scrubber [14] that models well-behaved protocol behavior. A model of a system may be already available (e.g., from design models, or as statically verified properties or properties that the system is

already known to enforce). If not available, a model could be built, sometimes automatically, using techniques such as static-analysis [7,8] and automata learning [22].

Motivations. A priori knowledge of the system behavior may help to improve monitoring mechanisms. We study how enforcement mechanisms can benefit from a model of the system. For the enforcement of non-safety properties (i.e., non prefix-closed properties), input events are delayed (stored in the enforcer's memory) until receiving all the events that allow to satisfy the desired property. If we have some knowledge of the system, we can sometimes react quickly (before receiving all the events that allow the input to satisfy the property), if we can predict that the input will inevitably satisfy the property in the future. Prediction thus allows to output events quickly (sometimes soon after they are received), instead of buffering them in the enforcer's memory. Predictive enforcement is hence desirable because it provides better Quality of Service (QoS). Moreover, in some particular scenarios where the actions in the input alphabet are dependent, without prediction, blocking some events (in case of non-safety properties) will lead the system into a deadlock situation, and thus non-safety properties can not be enforced [6]. For example, a requirement such as *"Every request should be followed by an acknowledgement"* can not be enforced in practice, and only having some knowledge of the system will allow to enforce such requirements. Our predictive enforcement framework allows to circumvent such situations and to enforce non-safety properties over dependent actions. In Section 3.1, we elaborate on the motivations through more detailed examples.

Problem definition. Given two regular properties φ and ψ, we aim to study mechanisms that take as input a word $\sigma \in \psi$ and output a sequence o that (i) satisfies φ (soundness), and (ii) is a prefix of input σ (transparency), as in standard RE [25,10,19]. In addition, we require (iii) the notion of *urgency*, which states that the observed input sequence σ must be released immediately (i.e., $o = \sigma$), if either σ already satisfies φ, or every possible extension of σ (that can be obtained from the knowledge of ψ) will result in the input satisfying φ. We refer to this problem as *predictive runtime enforcement*.

Contributions. We introduce a framework for predictive runtime enforcement of regular properties. In our framework, the notions of soundness and transparency are similar to those used in the standard RE frameworks [25,10,19]. However, in addition to soundness and transparency we propose the new notion of urgency, to ensure that input events are released as soon as possible, instead of being *blocked* or delayed until more input is observed. At an abstract level, we model enforcers as functions that transform words. We define the *constraints* that an enforcement function (for some property φ) should satisfy. Given a property φ, we present a functional definition, describing the input-output behavior, and also prove that it satisfies soundness, transparency, and urgency. Finally, we present algorithms describing how the proposed enforcement functions can be implemented. All our results related to predictive RE of untimed properties are formalized and proved in the Isabelle theorem prover [15] [1]. The proposed algorithms are implemented in Python to show the practical feasibility of our approach, and we also discuss examples of practical applications. The performance of the Python implementation has been evaluated using examples based on practical applications (see Section 3.7).

[1] The Isabelle theories are available at:

https://github.com/isabelle-theory/PredictiveRuntimeEnforcement

Moreover, we also discuss how our predictive runtime enforcement framework can be extended to enforce timed properties. Some earlier works [18,17,16,19], show how to synthesize enforcers for timed properties, where the event emitter is considered as a black-box. When we consider timed properties, in addition to the order of events, their occurrence time also affects the satisfaction of the property. For real-time systems, prediction for enforcement of timed properties is certainly advantageous since it allows to release events as output earlier (in some cases). We extend the notion of *urgency* from the untimed setting to the timed setting, and we define the predictive runtime enforcement problem for timed properties. We also present a functional definition, and some issues related to implementability of a timed predictive enforcer.

This paper extends the results of [20], and provides the following additional contributions:

- correctness proofs are included in Appendix A;
- the independence of the constraints of the enforcement mechanism are proved in Section 3.3;
- discussion of some applications is elaborated via examples in Section 3.6;
- proposed algorithms are implemented in Python[2] and discussed in Section 3.7.1;
- this implementation in Python has been evaluated using examples based on practical applications, discussed in Section 3.7.2;
- preliminary results on predictive RE for timed properties are provided in Section 4.

Paper organization. Section 2 presents preliminaries and notations. Section 3 is related to predictive RE of untimed properties. Firstly, we motivate predictive RE via a set of examples in Section 3.1, illustrating how an enforcer with some knowledge about the system behaves, compared to an enforcer without any knowledge of the system. In Section 3.2, we define the predictive RE problem formally, and the independence of the constraints of the enforcement mechanism are proved in Section 3.3. We provide a solution to the predictive RE problem in Section 3.4. Given two properties φ and ψ, we show how an enforcer can be defined as a function that transforms words. In Section 3.5, we present an algorithm which computes the output of the enforcement function incrementally (since we focus on online enforcement), given deterministic automata recognizing φ and ψ. In Section 3.6 some example applications are discussed, and in Section 3.7 we discuss about an implementation of the proposed algorithms and its evaluation using some examples.

Section 4 is related to predictive RE of timed properties. In Section 4.1, we first introduce some results about runtime enforcement for timed properties without prediction. Later in Section 4.2, we discuss some examples before we formally define the predictive RE problem for timed properties in Section 4.3, and in Section 4.4, we define a predictive enforcement mechanism as a function that transforms timed words.

Section 5 discusses related work, and Section 6 presents conclusions and open perspectives.

[2] The Python implementation is available for download at:

https://github.com/SrinivasPinisetty/PredictiveRE

2 Preliminaries and notation

In Section 2.1, we describe preliminaries and notations related to the untimed notions. In Section 2.2, we lift notations to the timed setting.

2.1 Untimed setting

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

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

The set $\mathrm{pref}(w)$ denotes the *set of prefixes* of w and, $\mathrm{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathcal{L}} \mathrm{pref}(w)$ 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 $\mathcal{L} \cdot \Sigma^* = \mathcal{L}$, where the concatenation operator naturally extends to sets of words.

For a word w and $i \in [1, |w|]$, the i-th letter of w is denoted as $w_{[i]}$. 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 denoted as $w_{[i \cdots j]}$, and the suffix of word w starting from index i is denoted as $w_{[i \cdots]}$.

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, i.e., $\Pi_i(e) \stackrel{\text{def}}{=} e_i$.

Deterministic and complete automata. A *deterministic and complete automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ where, Q is the set of *locations* (also called *states*), $q_0 \in Q$ is the initial location, Σ is the finite alphabet, $\delta : Q \times \Sigma \to Q$ is the (total) transition function and $F \subseteq Q$ is the set of accepting locations.

Any incomplete or non-deterministic automaton can be determinized and completed. Hence, in this paper we consider only deterministic and complete automata and the term "automaton" refers to "deterministic and complete automaton".

The transition function δ is extended to words by setting $\delta(q, \epsilon) = q$, and $\delta(q, a \cdot \sigma) = \delta(\delta(q, a), \sigma)$, for any $q \in Q, a \in \Sigma, \sigma \in \Sigma^*$.

Languages of automata. A word σ is *accepted* by $\mathcal{A}$ *starting from location* q if $\delta(q, \sigma) \in F$, and σ is *accepted* by $\mathcal{A}$ if σ is accepted starting from the initial location q_0.

The *language* of $\mathcal{A}$ *starting from location* q is denoted $\mathcal{L}(\mathcal{A}, q)$ and is the set of all accepted words from location q: $\mathcal{L}(\mathcal{A}, q) = \{\sigma \in \Sigma^* \mid \delta(q, \sigma) \in F\}$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is $\mathcal{L}(\mathcal{A}, q_0)$, i.e. the language of $\mathcal{A}$ from the initial location q_0.

The next lemma relates accepted words and the states reached by their prefixes in an automaton.

Lemma 1

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in \mathcal{L}(\mathcal{A}) \iff (\sigma' \in \mathcal{L}(\mathcal{A}, \delta(q_0, \sigma)))$$

Intuitively, Lemma 1 states that given any two words $\sigma, \sigma' \in \Sigma^*$, the word obtained by concatenating them ($\sigma \cdot \sigma'$) belongs to the language of $\mathcal{A}$ if and only if the word σ' belongs to the language accepted by $\mathcal{A}$ starting from the location reached by reading σ in $\mathcal{A}$ (i.e., from $\delta(q_0, \sigma)$).

Product and complementation of automata. Let automata $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ and $\mathcal{A}' = (Q', q'_0, \Sigma, \delta', F')$ be over the same alphabet Σ. The *product* of $\mathcal{A}$ and $\mathcal{A}'$, denoted $\mathcal{A} \times \mathcal{A}'$, is defined as $(Q \times Q', (q_0, q'_0), \Sigma, \delta \times \delta', F \times F')$, where $(\delta \times \delta')((q, q'), a) = (\delta(q, a), \delta'(q', a))$. The *complement* of $\mathcal{A}$ denoted as $\overline{\mathcal{A}}$ is defined as $(Q, q_0, \Sigma, \delta, Q \setminus F)$. For any states $q \in Q$, $q' \in Q'$, we have $\mathcal{L}(\mathcal{A} \times \mathcal{A}', (q, q')) = \mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}', q')$ and $\mathcal{L}(\overline{\mathcal{A}}, q) = \Sigma^* \setminus \mathcal{L}(\mathcal{A}, q)$.

Classification of properties. A *regular property* is a language accepted by an automaton. In the sequel, we consider only regular properties and we refer to them as properties.

Safety (resp. co-safety) properties are sub-classes of regular properties [3]. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). Formally, safety properties are the prefix-closed languages that can be accepted by an automaton. Co-safety properties are the extension-closed languages that can be accepted by an automaton.

Thus, an automaton $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ is:

- a *safety* automaton if $\forall a \in \Sigma, \forall q \in Q \setminus F : \delta(q, a) \notin F$,
- a *co-safety* automaton if $\forall a \in \Sigma, \forall q \in F : \delta(q, a) \in F$.

Note that the complement of a safety automaton is a co-safety automaton and vice-versa. In these definitions we consider automata where Q contains only locations reachable from the initial location q_0.

2.2 Timed setting

Timed words and languages. In a timed setting, the occurrence time of actions is also important. For an enforcer in a timed setting, input (resp. output) sequences are seen as sequences of events composed of a date and an action, where the date is interpreted as the absolute time when the action is received (resp. released) by the enforcer.

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), where $\mathrm{date}((t, a)) \stackrel{\mathrm{def}}{=} t \in \mathbb{R}_{\geq 0}$ is the absolute time at which the action $\mathrm{act}((t, a)) \stackrel{\mathrm{def}}{=} a \in \Sigma$ occurs.

In a timed setting, input and output sequences of enforcers are timed words. A timed word over the finite 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 $\mathrm{start}(\sigma) \stackrel{\mathrm{def}}{=} t_1$ the starting date of σ and $\mathrm{end}(\sigma) \stackrel{\mathrm{def}}{=} t_n$ its ending date (with the convention that the starting and ending dates are null 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, etc) naturally extend to timed words.

When concatenating two timed words, one should ensure that the concatenation results in a timed word, i.e., dates should be non-decreasing. This is guaranteed if the ending date of the first timed word does not exceed the starting date of the second one. Formally, let $\sigma =$

[3] Similarly to some monitoring frameworks [9,24,19], we consider safety and co-safety properties over finite words.

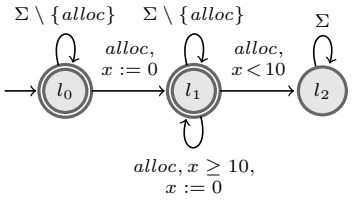


Fig. 2: Timed automaton: example

$(t_1, a_1) \cdots (t_n, a_n)$ 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 when $\mathrm{end}(\sigma) > \mathrm{start}(\sigma')$.

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).

Example 1 Consider a set of actions $\Sigma = \{a, b, c\}$ and $\sigma_1 = (1, a) \cdot (2.3, b) \cdot (3, a) \cdot (4, c)$ a timed word over Σ. Note that the occurrence dates of events in σ_1 are increasing. The starting date of σ_1 is $\mathrm{start}(\sigma_1) = 1$ and the ending date is $\mathrm{end}(\sigma_1) = 4$. The untimed projection of σ_1 is $\Pi_\Sigma(\sigma_1) = a \cdot b \cdot a \cdot c$. Consider two more timed words over Σ, $\sigma_2 = (2, b) \cdot (2.3, b) \cdot (3, a)$, and $\sigma_3 = (10, b) \cdot (12, a)$. The concatenation $\sigma_1 \cdot \sigma_2$ is undefined since $\mathrm{start}(\sigma_2)$ is less than $\mathrm{end}(\sigma_1)$. The concatenation of σ_1 and σ_3 is $\sigma_1 \cdot \sigma_3 = (1, a) \cdot (2.3, b) \cdot (3, a) \cdot (4, c) \cdot (10, b) \cdot (12, a)$.

2.2.1 Timed automata and timed properties

A timed automaton [1] 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 $\chi \in \mathbb{R}^X_{\geq 0}$ and $\tau \in \mathbb{R}_{\geq 0}$, $\chi + \tau$ is the valuation assigning $\chi(x) + \tau$ to each clock x of X. Given a set of clocks $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the clock valuation χ where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of *guards*, i.e., clock constraints defined as conjunctions 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 $\chi \in \mathbb{R}^X_{\geq 0}$, we write $\chi \models g$ when g holds according to χ.

Timed automata syntax and semantics. Before going into the formal definitions, we introduce timed automata on an example. The timed automaton in Fig. 2 defines the requirement *"In every 10 time units, there cannot be more than 1 alloc action"*. The set of locations is $L = \{l_0, l_1, l_2\}$, and l_0 is the initial location [4]. The set of actions is $\Sigma = \{alloc, rel\}$. There are transitions between locations upon actions. A finite set of real-valued clocks is used to model real-time behavior: set $X = \{x\}$ in the example. On the transitions, there are i) guards with constraints on clock values (such as $x < 10$ on the transition between l_1 and l_2 in the considered example), and ii) resets of clocks. Upon the first occurrence of action $alloc$, the automaton moves from l_0 to l_1, and the clock x is reset to 0. In location l_1, if action $alloc$ is received, and if $x \geq 10$, then the automaton remains in l_1, resetting the value of clock x to 0. It moves to location l_2 otherwise.

[4] We denote accepting locations using double circles.

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*, $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times \mathcal{P}(X) \times L$ is the *transition relation*, where $\mathcal{P}(X)$ is the powerset of X (the set of all subsets of X). $F \subseteq L$ is a set of *accepting locations*.

The semantics of a timed automaton is defined as a transition system where each state consists of the current location and the current values of clocks. Since the possible values for a clock are infinite, a timed automaton has infinitely many states. 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 $Q = L \times \mathbb{R}^X_{\geq 0}$ is the (infinite) set of *states*, $q_0 = (l_0, \chi_0)$ is the *initial state* where χ_0 is the valuation that maps every clock in X to 0, $Q_F = F \times \mathbb{R}^X_{\geq 0}$ 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 $(l, \chi) \xrightarrow{(\tau, a)} (l', \chi')$ with $\chi' = (\chi + \tau)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ such that $\chi + \tau \models g$ for $\tau \in \mathbb{R}_{\geq 0}$.

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 in Δ, the conjunction of guards $g_1 \wedge g_2$ is unsatisfiable. $\mathcal{A}$ is said *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 labeled by a evaluates to *true* (i.e., it holds according to any valuation): $(\forall l \in L, \forall a \in \Sigma : \bigvee_{(l,g,a,Y,l')} g) = true$. For example, the TA in Fig. 2 is deterministic and complete.

In the remainder of this paper, we shall consider only deterministic and complete timed automata. Note that not all non-deterministic timed automata are determinizable [12,26,2].

A *run* ρ of $\mathcal{A}$ from a state $q \in Q$ triggered at time $t \in \mathbb{R}_{\geq 0}$ over a timed *trace* $w_t = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ is a sequence of moves in $[\![\mathcal{A}]\!]$ denoted as $\rho_t = q \xrightarrow{(\tau_1, a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\tau_n, a_n)} q_n$, for some $n \in \mathbb{N}$, satisfying condition $t_1 = t + \tau_1$ and $\forall i \in [2, n], t_i = t_{i-1} + \tau_i$. To simplify notations, we note $q \xrightarrow{w_t} q_n$ in this case, and generalize it to $q \xrightarrow{w_t} P$ when $q_n \in P$ for a subset P of Q. The set of runs from the initial state $q_0 \in Q$, starting at $t = 0$ is denoted $\mathrm{Run}(\mathcal{A})$ and $\mathrm{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$. We denote by $\mathcal{L}(\mathcal{A})$ the set of traces 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 of an accepted run.

Timed properties. In the sequel, a timed property is defined by a timed language $\varphi \subseteq \mathrm{tw}(\Sigma)$ that can be recognized by a complete and deterministic timed automaton. That is, we consider the set of regular timed properties that can be defined as deterministic timed automata. Given a timed word $\sigma \in \mathrm{tw}(\Sigma)$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$.

3 Predictive runtime enforcement of untimed properties

This section introduces predictive runtime enforcement of untimed properties. In this section, φ and ψ are properties defined by deterministic and complete automata. First, we motivate predictive RE via examples in Section 3.1. Then, we formally introduce the predictive

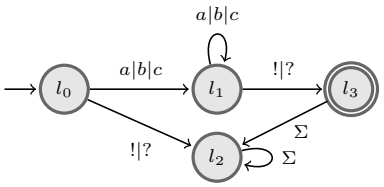


Fig. 3: Property to enforce: φ

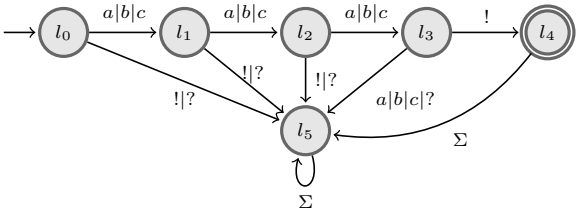


Fig. 4: Possible input sequences: ψ_1

RE problem in Section 3.2 and in Section 3.3 we prove the independence of the constraints of the enforcement mechanism. We later provide a solution to the predictive RE problem in Section 3.4 defining a predictive enforcement mechanism as a function that transforms words. We also present algorithms that implement the proposed enforcement function in Section 3.5. In Section 3.6 and Section 3.7, we briefly discuss some example applications, and an implementation of the proposed algorithms.

3.1 Motivating examples

Before formally defining the problem of predictive RE, let us motivate the usefulness of prediction in runtime enforcement via some examples.

3.1.1 Enforcing file format requirements

Consider a scenario where an application writes to a file, using multiple write operations. At the end of the sequence of writes, the file must obey a required format. The format might not hold in the middle of the sequence of writes (so, the property is not prefix-closed).

Let us now consider a specific requirement. Consider a simple application (say application 1) that allows to write a non-empty string containing characters from the set $\{a, b, c\}$. We also have special end-of-string characters $\{!, ?\}$: the string should end with one of these characters, which cannot occur elsewhere in the string. The string is valid only if this required format condition holds. The automaton in Fig. 3 defines this requirement φ. Its alphabet is $\Sigma = \{a, b, c, !, ?\}$. Location l_0 is initial, and the only accepting location is l_3.

Consider another application (say application 2) that makes use of application 1. Without any knowledge about the sequence of write operations performed by application 2 (where each write operation writes a character), the enforcer for φ must buffer all the writes without saving to disk until one of the special characters is received. Once it receives a special character, it can "flush" its buffer.

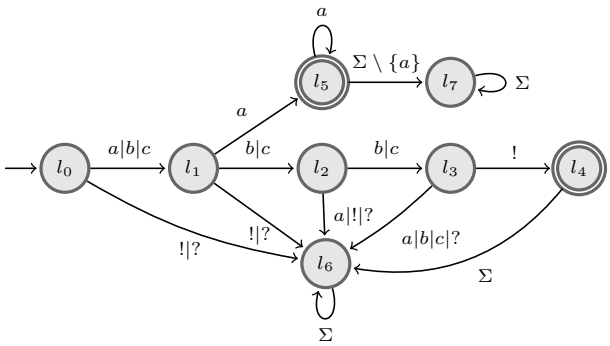


Fig. 5: Possible input sequences: ψ_2

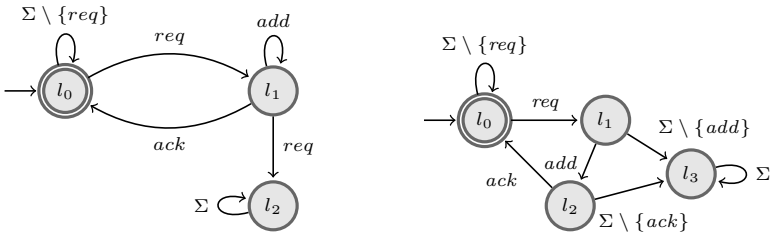
Input ψ_1. Suppose that we have some knowledge of application 2, and we know what strings it can produce. Consider the automaton in Fig. 4 modeling strings that application 2 can output (that will be given as input to the enforcer). So, we now know that the input sequence that the enforcer receives is three characters (each of them belonging to $\{a, b, c\}$) ending with the special character "!". Thus, the input sequences that the enforcer will receive are $\psi_1 = \{abc!,\ aac!,\ \ldots\}$. Suppose that the input is $\sigma = abc!$. Without prediction, the enforcer will buffer events a, b, and c in its memory until it sees an "!" event. But with prediction, each event will be output (written) immediately after it is read.

Input ψ_2. In the predictive setting, it is not always possible to output events immediately, and in some situations we may require to buffer some events in the memory of the enforcer. For example, instead of ψ_1, consider ψ_2 (defined by the automaton in Fig. 5) defining possible input sequences. If the second character is "a", then the third character should also be an "a", and a special character at the end is not necessary for such strings. Consequently, when the enforcer sees the first character, it cannot output it immediately. It has to wait until it receives the second character, and if the second character is not an "a", then it outputs the first character followed by the second character. If the third character is not an "a", it can be output immediately (without waiting for a special character as in the non-predictive case).

3.1.2 Monitoring communication

Let us consider another situation where two applications communicate with one another. Application 1 can request a service from the other application by sending a request (req) message, and application 2 acknowledges any request received from the other application by sending an acknowledgement (ack) message. Message (add) corresponds to adding a request to a processing queue. Consider a requirement that *"Every request should be followed by an acknowledgement"*. The automaton in Fig. 6a formally defines this requirement. The set of actions is $\Sigma = \{req, ack, add\}$, and location l_0 is initial, and accepting.

Without prediction. In approaches without prediction (cf. [13,10,19]), the enforcer outputs events only after observing a sequence of events which satisfies the property. Consequently, when the enforcer receives a request (req) from application 1, it will not release it, and will keep waiting forever for an acknowledgement (ack) from application 2 which will be sent by application 2 only after it receives a request. This will result in a deadlock situation. Thus, in



(a) Property to enforce: φ (b) Input sequences: ψ

Fig. 6: Monitoring communication

this particular scenario, actions (req) and (ack) are dependent, and thus the property cannot be enforced without prediction.

With prediction. Now, consider that we have some knowledge regarding behavior of application 2, defined by the automaton in Fig. 6b. Whenever application 2 receives a request (req) message, it does some processing (such as adding this request to a processing queue (add)), and it will send an acknowledgement (ack) message. If the enforcer has this knowledge from ψ regarding what application 2 does upon receiving a request (req) message, it can learn that it will receive an acknowledgment (ack) in the future. Thus, whenever it receives a request (req) message from application 1, it can release it immediately. Consequently, with prediction (given ψ also as input to the enforcer, that defines all possible input sequences), property φ can be enforced. Moreover, note that in this particular example, $\psi \subseteq \varphi$, and thus any input can be immediately released as output.

3.2 Predictive runtime enforcement

In this section, we formalize the predictive runtime enforcement problem. Roughly speaking, the purpose of enforcement monitoring is to read some (possibly incorrect) sequence produced by a running system (input to the enforcer), and to transform it into an output sequence that is correct w.r.t. a property φ that we want to enforce. At an abstract level, an enforcer can be seen as a function that transforms words. An enforcement function for a given property φ takes as input a word over Σ and outputs a word over Σ that is either empty or belongs to φ.

Now in the predictive case, instead of considering Σ^* as the language of possible inputs, we consider $\psi \subseteq \Sigma^*$, that defines the set of possible sequences that the enforcer receives at runtime as input. As we discussed in the introduction, ψ may be already available (e.g., from design models, or as statically verified properties or properties that the system is already known to enforce). If ψ is not available, it could also be built for instance from knowledge obtained using static-analysis.

Similar to enforcement mechanisms in [25, 10, 13, 19], several constraints are required on how an enforcement function transforms words. Our enforcement mechanism cannot insert (or suppress) events, and cannot change their order, but is allowed to block when a violation is detected. The notions of soundness, transparency and monotonicity are similar to the ones in the non-predictive case [10, 25]. Soundness expresses that the output must satisfy property φ, and transparency generally aims at preventing the input sequence from being modified unnecessarily.

In our predictive setting, we introduce an additional requirement called *urgency*, expressing that if the input sequence received so far does not satisfy the property φ, it is still released as output immediately if all possible input events that the enforcer will receive in the future will allow to satisfy φ.

Definition 3 (Predictive enforcer) Given properties $\psi, \varphi \subseteq \Sigma^*$, a *predictive enforcer* for ψ, φ is a function $E_{\psi,\varphi} : \Sigma^* \to \Sigma^*$ satisfying the following constraints:

Soundness

$$\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \neq \epsilon \implies E_{\psi,\varphi}(\sigma) \in \varphi \quad \textbf{(Snd)}$$

Transparency

$$\forall \sigma \in \Sigma^* : E_{\psi,\varphi}(\sigma) \preccurlyeq \sigma \quad \textbf{(Tr1)}$$

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma \quad \textbf{(Tr2)}$$

Monotonicity

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies E_{\psi,\varphi}(\sigma) \preccurlyeq E_{\psi,\varphi}(\sigma') \quad \textbf{(Mo)}$$

Urgency

$$\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma \quad \textbf{(Ur)}$$

As we shall show later in Section 3.4, for any ψ, φ, a predictive enforcer always exists (Theorem 1), and can be computed using the algorithm described in Section 3.5.

Soundness. (**Snd**) means that for any input word belonging to ψ, if the output of the enforcer is non-empty (ϵ), then it must satisfy φ. Regarding soundness, notice that it is restricted to input words that belong to ψ. When the input word is in ψ, it means that it may be a complete input (with no extension) and we want to be sure that the output satisfies φ. However, when the input is not in ψ, we do not require the output to be in φ, since we allow to predict and output the observed input sequence immediately (irrespective of whether it belongs to φ), if we know that every possible continuation of the observed input satisfies φ (see urgency).

Note that the condition stating that the output be non-empty is unavoidable. The output of the enforcer may be ϵ, and ϵ may not belong to the language accepted by some properties such as some non-safety properties. For example, ϵ does not belong to the language accepted by the automaton in Fig. 3. If instead we had formalized soundness as $\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \in \varphi$, then if the output for some non-safety property is ϵ, then this soundness condition cannot be satisfied. For example, let the automaton in Fig. 3 define the property φ we want to enforce, and the automaton in Fig. 5 define the set of input sequences. The sequence baa is a valid input sequence. But, none of the prefixes of this sequence (including ϵ) satisfies the property φ, and the output of the enforcer will be ϵ in such cases.

Transparency. Transparency is similar to its version in the non-predictive setting [10, 19]. It is expressed as the conjunction of the two constraints (**Tr1**) and (**Tr2**). (**Tr1**) expresses that the output of the enforcer should be a prefix of the input. This constraint corresponds to the fact that the enforcer is allowed only to block events, but it is not allowed to insert (or suppress) events, and also cannot change their order. (**Tr2**) expresses that, if the input word satisfies the property, then the output should be equal to the input.

(**Tr1**) allows to block events. For some properties (e.g., safety), blocking everything (producing ϵ as output for any input sequence) will satisfy both (**Snd**) and (**Tr1**) constraints. (**Tr2**) is added to ensure that the enforcer should alter the input sequence minimally (i.e., if the input sequence satisfies the property φ, then the enforcer should output it completely). As it turns out, (**Tr2**) is a consequence of the urgency constraint, (**Ur**), and is therefore redundant (see Lemma 2 that follows).

Monotonicity. The monotonicity constraint means that the enforcer cannot undo what is already released as output during the incremental computation. (**Mo**) expresses that the output of the enforcer for an extended input word σ' of an input word σ, extends the output produced by the enforcer for σ. This can be seen as a *causality* property.

Urgency. (**Ur**) expresses that, when the enforcer knows that the input σ followed by a prefix (σ') of any continuation σ_{con} (such that $\sigma \cdot \sigma_{\mathrm{con}} \in \psi$) will satisfy the property ($\sigma \cdot \sigma' \in \varphi$), then the output of the enforcer after reading σ (which is $E_{\psi,\varphi}(\sigma)$), should be σ.

The urgency constraint is related to releasing events as output as soon as possible. It expresses that if an input word (which may or may not belong to ψ) satisfies φ, then it should be output immediately without waiting for future input events. In case if the word received so far does not satisfy φ, the enforcer should still output the input word immediately if all possible future continuations of that (we can obtain from ψ) will satisfy φ.

Remark 1 (Online behavior and releasing events as soon as possible.) Note that our enforcer works in an online fashion (running in parallel with the system), and thus it does not have the entire input sequence, and moreover its length is also unknown. For efficiency reasons, the output should be built incrementally in a online fashion. Enforcer should output events as soon as possible. The monotonicity and urgency constraints are related to this online behavior of the enforcer.

Lemma 2 *(***Tr2***) is a consequence of (***Ur***).*

$$(\mathbf{Ur}) \implies (\mathbf{Tr2})$$

Proof When the input word σ belongs to φ, for every possible extension of the input σ_{con}, there is a prefix which is ϵ and $\sigma \cdot \epsilon \in \varphi$. Consequently, whenever $\sigma \in \varphi$ (which is the hypothesis of (**Tr2**)), the hypothesis of (**Ur**) will be true. Thus we can conclude that $(\mathbf{Ur}) \implies (\mathbf{Tr2})$.

To see why (**Mo**) is needed, consider the following example. Let φ be the automaton in Fig. 3, and let $\psi = \Sigma^*$. Consider the input sequence $\sigma_1 = abc!$. Due to (**Ur**), and since the hypothesis of (**Ur**) holds for σ_1, $E_{\psi,\varphi}(\sigma_1)$ should be $abc!$. Suppose that the input is extended with a few more events and let the new input sequence be $\sigma_2 = \sigma_1 \cdot ab = abc!ab$. When we consider σ_2, the hypothesis of (**Ur**) does not hold. Therefore, without (**Mo**), $E_{\psi,\varphi}(\sigma_2)$ could be either ϵ or $abc!$, since both these words satisfy (**Snd**), (**Tr1**), and (**Ur**). Constraint (**Mo**) only allows to modify the output by appending more events to the output. Thus, together with (**Mo**), the only possible output for the input word $abc!ab$ is $abc!$, which is the maximal prefix of $abc!ab$ satisfying the hypothesis of (**Ur**).

For any φ, ψ, for any input word $\sigma \in \Sigma^*$, the output of any enforcer that satisfies the constraints (**Snd**), (**Tr1**), (**Ur**) and (**Mo**) will be the maximal prefix of the input satisfying the hypothesis of the (**Ur**) constraint.

Remark 2 (Alternative urgency constraint) A weaker definition of urgency could be:

$$\forall \sigma \in \Sigma^* : (\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \implies \sigma \cdot \sigma_{\mathrm{con}} \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma. \quad \textbf{(Ur')}$$

The (**Ur'**) constraint expresses that, when the input σ followed by any continuation σ_{con} s.t. $\sigma \cdot \sigma_{\mathrm{con}} \in \psi$ satisfies property φ, then the output of the enforcer after reading σ (which is $E_{\psi,\varphi}(\sigma)$), should be σ.

Urgency constraint (**Ur'**) is weaker than urgency constraint (**Ur**). If $\sigma \cdot \sigma_{\mathrm{con}} \in \psi$ then there is certainly a prefix of σ_{con} (which is σ_{con} in this case) such that σ followed by that prefix satisfies φ. From transparency constraints, notice that the output can be a prefix of the input sequence. Thus, our urgency constraint (**Ur**) is stronger, since we consider all the prefixes of σ_{con} instead of σ_{con} only. Therefore, (**Ur**) results in avoiding unnecessary delaying of the input which (**Ur'**) might have allowed (in some cases).

Lemma 3 *The alternative urgency constraint (***Ur'***) is weaker than the urgency constraint (***Ur***), i.e.,* (**Ur**) $\implies$ (**Ur'**).

The proof of Lemma 3 is given in Appendix A.1 in page 39.

Remark 3 When we have no knowledge about the system (i.e., $\psi = \Sigma^*$), soundness and transparency constraints reduces to the non-predictive case by replacing $\sigma \in \Sigma^*$ instead of $\sigma \in \psi$. The notion of urgency in this case can be simplified as follows:

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma.$$

Lemma 4 *When $\psi = \Sigma^*$, the constraint (***Ur***) is equivalent to the following condition (which is (***Tr2***) constraint):*

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma.$$

The proof of Lemma 4 is given in Appendix A.1 in page 39.

Note, when $\psi \subseteq \varphi$, then the enforcer immediately outputs any word received as input. The output of the enforcement function is always equal to the input.

Lemma 5 $\psi \subseteq \varphi \implies (\forall \sigma \in \Sigma^* : E_{\psi,\varphi}(\sigma) = \sigma)$.

The proof of Lemma 5 is given in Appendix A.1 in page 40.

3.3 Independence of the constraints

We prove that the constraints (**Snd**), (**Tr1**), (**Ur**) and (**Mo**) are independent. We prove their independence by showing that all combinations of three of these constraints together with the negation of the fourth have models. If $\Sigma = \{\bullet\}$, where $\bullet$ is a letter, then the following four lemmas show the independence of the constraints.

Lemma 6 *If $\psi = \{\bullet\}^*$, $\varphi = \{\bullet\}$, and $(\forall \sigma : E_{\psi,\varphi}(\sigma) = \sigma)$ then*

$$\neg(\mathbf{Snd}) \wedge (\mathbf{Tr1}) \wedge (\mathbf{Ur}) \wedge (\mathbf{Mo})$$

Lemma 7 *If $\psi = \{\bullet\}^*$, $\varphi = \{\bullet\}$, and $(\forall \sigma : E_{\psi,\varphi}(\sigma) = \bullet)$ then*

$$(\mathbf{Snd}) \wedge \neg(\mathbf{Tr1}) \wedge (\mathbf{Ur}) \wedge (\mathbf{Mo})$$

Lemma 8 *If* $\psi = \{\bullet\}^*$, $\varphi = \{\bullet\}$, $(\forall \sigma \neq \bullet : E_{\psi,\varphi}(\sigma) = \epsilon)$, *and* $E_{\psi,\varphi}(\bullet) = \bullet$ *then*

$$(\mathbf{Snd}) \wedge (\mathbf{Tr1}) \wedge (\mathbf{Ur}) \wedge \neg(\mathbf{Mo})$$

Lemma 9 *If* $\psi = \{\bullet\}^*$, $\varphi = \{\bullet\}^*$, *and* $(\forall \sigma : E_{\psi,\varphi}(\sigma) = \epsilon)$ *then*

$$(\mathbf{Snd}) \wedge (\mathbf{Tr1}) \wedge \neg(\mathbf{Ur}) \wedge (\mathbf{Mo})$$

The proofs of these lemmas follow easily using Lemma 4 ($\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma$).

3.4 Functional definition

In this section, we provide a definition of a predictive enforcer as a function that incrementally builds the output. This functional definition provides an abstract view, describing how to transform an input word according to the property φ. We also prove that this functional definition satisfies all the constraints of a predictive enforcer defined in Section 3.2.

Definition 4 (Enforcement function) Given properties $\psi, \varphi \subseteq \Sigma^*$, where ψ is the property of possible input sequences, and φ is the property that we want to enforce, the enforcement function $E_{\psi,\varphi} : \Sigma^* \to \Sigma^*$ is defined as:

$$E_{\psi,\varphi}(\sigma) = \Pi_1\Big(\mathrm{store}_{\psi,\varphi}(\sigma)\Big).$$

where:

- $\mathrm{store}_{\psi,\varphi} : \Sigma^* \to \Sigma^* \times \Sigma^*$ is defined as:

$$\begin{aligned} \mathrm{store}_{\psi,\varphi}(\epsilon) &= (\epsilon, \epsilon) \\ \mathrm{store}_{\psi,\varphi}(\sigma \cdot a) &= \begin{cases} (\sigma_s \cdot \sigma_c \cdot a, \epsilon) & \text{if } \kappa_{\psi,\varphi}(\sigma_s \cdot \sigma_c \cdot a), \\ (\sigma_s, \sigma_c \cdot a) & \text{otherwise} \end{cases} \end{aligned}$$

  with $(\sigma_s, \sigma_c) = \mathrm{store}_{\psi,\varphi}(\sigma)$.

- $\kappa_{\psi,\varphi}(\sigma)$ is defined as:

$$\kappa_{\psi,\varphi}(\sigma) = (\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\mathrm{con}} \wedge \sigma \cdot \sigma' \in \varphi).$$

The enforcement function takes a word over Σ as input, and produces a word over Σ as output. Function store takes a word over Σ as input, and computes a pair of words over Σ. The first component of the output of function store (extracted by Π_1) is the output of the enforcement function.

The first element of the output of function store is a prefix of the input that will be the output of the enforcement function (the property φ is satisfied by this prefix followed by any continuation of the input including ϵ); the second element is the suffix of the input which the enforcer cannot output yet. Function store is defined inductively: initially, for an empty input, both elements are empty; if σ is read, $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$, and another new event $a \in \Sigma$ is observed, there are two possible cases based on whether $\kappa_{\psi,\varphi}$ holds or not.

$\kappa_{\psi,\varphi}$ is a Boolean function (predicate) that takes a word over Σ as input and returns a Boolean as output. This function tests the hypothesis of the urgency constraint, (**Ur**). $\kappa_{\psi,\varphi}$ returns `true` if for every continuation σ_{con} such that $\sigma \cdot \sigma_{\mathrm{con}} \in \psi$, there is a prefix σ' of σ_{con}

such that $\sigma \cdot \sigma' \in \varphi$. Thus, if the sequence σ provided as input to this function satisfies φ, then this condition will be satisfied since for every continuation σ_{con}, ϵ is a prefix of σ_{con} such that $\sigma \cdot \epsilon \in \varphi$. The function $\kappa_{\psi,\varphi}$ returns false if the input sequence σ does not satisfy φ, and there is a continuation of σ that will not allow to satisfy φ.

Remark 4 Note that when $\psi = \Sigma^*$, following Lemma 4, the function $\kappa_{\psi,\varphi}(\sigma)$ can be simplified, and defined as follows:

$$\kappa_{\psi,\varphi}(\sigma) = (\sigma \in \varphi)$$

Lemma 10 introduces some properties of the enforcement function, and auxiliary function $\kappa_{\psi,\varphi}$ that will be used in proving that the enforcement function satisfies the soundness, transparency, urgency, and monotonicity constraints.

Lemma 10 *For all $\sigma, \sigma', \sigma_s, \sigma_c \in \Sigma^*$ we have*

1. $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c) \implies \sigma = \sigma_s \cdot \sigma_c$
2. $E_{\psi,\varphi}(\sigma) \neq \epsilon \implies \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$
3. $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma')$
4. $\sigma \in \varphi \implies \kappa_{\psi,\varphi}(\sigma)$

The proof of Lemma 10 is given in Appendix A.1 in page 40. These properties are also proved in Isabelle. They are proved by induction on σ or the length of σ.

Property 1 of Lemma 10 states that for any input sequence σ, if $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$, the concatenation of the two output words of $\mathrm{store}_{\psi,\varphi}$ which is $\sigma_s \cdot \sigma_c$ is equal to the input word σ.

Property 2 of Lemma 10 states that for any input sequence σ, if the output of the enforcement function $E_{\psi,\varphi}(\sigma)$ is not ϵ, then $\kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$ holds. This means that the sequence that is released as output $E_{\psi,\varphi}(\sigma)$, will certainly be extended in the future to satisfy φ.

Property 3 of Lemma 10 states that for any two sequences σ and σ', if $\kappa_{\psi,\varphi}(\sigma)$ holds and if σ is a prefix of σ', then σ is also a prefix of the output of the enforcement function for σ'. This means that if the input σ allows to satisfy φ (for every possible future extension of it), then σ should be a prefix of the output of the enforcement function for a longer input sequence σ'.

Finally, property 4 of Lemma 10 states that for any input sequence σ, if σ belongs to property φ, then the function $\kappa_{\psi,\varphi}$ for input σ returns true.

Theorem 1 (Soundness, transparency, monotonicity, and urgency) *Given two properties ψ, and φ, the enforcement function $E_{\psi,\varphi}$ as per Definition 4 is a predictive enforcer satisfying constraints (***Snd***), (***Tr1***), (***Tr2***), (***Mo***), and (***Ur***).*

According to Theorem 1, for any two properties ψ and φ, a predictive enforcer always exists.

Proof This theorem can be proved using Definitions 3 and 4 and Lemma 10. The properties (**Tr1**), (**Tr2**), (**Ur**), and (**Mo**) are immediate consequences of the definitions and Lemma 10.

We show here the proof of the soundness (**Snd**) constraint:

$$\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \neq \epsilon \implies E_{\psi,\varphi}(\sigma) \in \varphi.$$

Let us consider $\sigma \in \psi$ and $E_{\psi,\varphi}(\sigma) \neq \epsilon$, and prove that $E_{\psi,\varphi}(\sigma) \in \varphi$.

From (**Tr1**), we have $E_{\psi,\varphi}(\sigma) \preccurlyeq \sigma$, implying that there is a σ' such that $\sigma = E_{\psi,\varphi}(\sigma) \cdot \sigma'$. From Lemma 10.2 we have also $\kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$, and by expanding the definition of $\kappa_{\psi,\varphi}$, we obtain:

$$\kappa_{\psi,\varphi}(\sigma) = (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$$

Consequently, for $\sigma_{\text{con}} = \sigma'$, we obtain $\sigma'' \preccurlyeq \sigma'$ such that $E_{\psi,\varphi}(\sigma) \cdot \sigma'' \in \varphi$. From $E_{\psi,\varphi}(\sigma) \cdot \sigma'' \in \varphi$, using Lemma 10.4, we have $\kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma) \cdot \sigma'')$. We prove now that $\sigma'' = \epsilon$:

$$\begin{array}{ll} & true \\ \Leftrightarrow & \{\text{proved property}\} \\ & \sigma'' \preccurlyeq \sigma' \\ \Rightarrow & \{\text{property of word concatenation}\} \\ & E_{\psi,\varphi}(\sigma) \cdot \sigma'' \preccurlyeq E_{\psi,\varphi}(\sigma) \cdot \sigma' \\ \Leftrightarrow & \{\text{property } \sigma = E_{\psi,\varphi}(\sigma) \cdot \sigma'\} \\ & E_{\psi,\varphi}(\sigma) \cdot \sigma'' \preccurlyeq \sigma \\ \Rightarrow & \{\text{Lemma 10.3 and } \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma) \cdot \sigma'')\} \\ & E_{\psi,\varphi}(\sigma) \cdot \sigma'' \preccurlyeq E_{\psi,\varphi}(\sigma) \\ \Rightarrow & \{\text{properties of concatenation and prefix}\} \\ & \sigma'' = \epsilon \end{array}$$

Because $\sigma'' = \epsilon$, from $E_{\psi,\varphi}(\sigma) \cdot \sigma'' \in \varphi$ we obtain the conclusion $E_{\psi,\varphi}(\sigma) \in \varphi$.

Theorem 1 is also proved in Isabelle.

Maximal output. For any input word $\sigma \in \Sigma^*$, the output of the enforcement function $E_{\psi,\varphi}$ as per Definition 4 is the maximal prefix of the input word σ that satisfies the hypothesis of the urgency constraint (**Ur**). In fact any other prefix σ' of σ that satisfies $\kappa_{\psi,\varphi}(\sigma')$ is a prefix of $E_{\psi,\varphi}(\sigma)$, and this property is expressed in Lemma 10.3.

Remark 5 (Safety properties) Using ψ and predicting future extensions of the input is useful when φ is not prefix closed. In case if the property φ is a safety property (i.e. prefix closed), every prefix of the output sequence should satisfy the property φ. Thus, once φ is violated, no continuation will allow to satisfy the property in the future. Decision to output an event should be taken immediately after the event is received (also when $\psi = \Sigma^*$).

Example 2 (Incremental computation by the enforcement function) Let the property to enforce φ be represented by the automaton in Fig. 3, and let the input property ψ be represented by the automaton in Fig. 5. Table 1 illustrates how the enforcement function incrementally builds the output when the input word is abc!.

3.5 Enforcement algorithm

In Section 3.4, we provided an abstract view of our predictive enforcement monitoring mechanism, defining it as a function that transforms words. However, it is not immediate to see how this function can be implemented. In particular, how the component function $\kappa_{\psi,\varphi}$ can be implemented is not straightforward. In this section, we provide algorithms that implement $\kappa_{\psi,\varphi}$, as well as the overall enforcement function.

Let automaton $\mathcal{A}_\psi = (Q_\psi, q_\psi, \Sigma, \delta_\psi, F_\psi)$ define property $\psi = \mathcal{L}(\mathcal{A}_\psi, q_\psi)$, and automaton $\mathcal{A}_\varphi = (Q_\varphi, q_\varphi, \Sigma, \delta_\varphi, F_\varphi)$ define property $\varphi = \mathcal{L}(\mathcal{A}_\varphi, q_\varphi)$. We recall that ψ

Table 1: Example illustrating incremental computation by the enforcement function.

σ	σ_s	σ_c	$E_{\psi,\varphi}(\sigma)$
$\epsilon \notin \psi$	ϵ	ϵ	$\epsilon \notin \varphi$
a $\notin \psi$	ϵ	a	$\epsilon \notin \varphi$
ab $\notin \psi$	ab	ϵ	ab $\notin \varphi$
abc $\notin \psi$	abc	ϵ	abc $\notin \varphi$
abc! $\in \psi$	abc!	ϵ	abc! $\in \varphi$

models the property of possible input sequences, and φ models the property that we want to enforce.

We devise an on-line algorithm, with input $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$, which is an infinite loop that waits for input events (letters of the alphabet). We know that any input sequence that we get satisfies ψ eventually. An iteration of the algorithm is triggered by an input event. If the sequence of events obtained already followed by the current event does not satisfy $\kappa_{\psi,\varphi}$ then we hold this event. Otherwise we output all events held by earlier iterations.

We start by implementing function $\kappa_{\psi,\varphi}$.

Implementation of $\kappa_{\psi,\varphi}$ We first introduce an automaton $\mathcal{B}_\varphi$ based on $\mathcal{A}_\varphi$. Let $\mathcal{B}_\varphi = (Q_\varphi, q_\varphi, \Sigma, \delta'_\varphi, F_\varphi)$, where δ'_φ is defined as

$$\delta'_\varphi(q,a) = \begin{cases} \delta_\varphi(q,a) & \text{if } q \notin F_\varphi, \\ q & \text{otherwise .} \end{cases}$$

In $\mathcal{B}_\varphi$, we retain all the transitions in $\mathcal{A}_\varphi$ that are from non-accepting locations. Any transition from an accepting location in $\mathcal{A}_\varphi$ is directed to the same accepting location. Thus, in automaton $\mathcal{B}_\varphi$, we will not have transitions from accepting to non accepting locations.

Intuitively, a word σ is accepted by $\mathcal{B}_\varphi$ starting from $q \in Q_\varphi$ if it is an extension of a word accepted by $\mathcal{A}_\varphi$ starting also from q. We can see also that the property $\mathcal{L}(\mathcal{B}_\varphi, q)$ is a co-safety property [5].

Lemma 11 *If $q \in Q_\varphi$ and $\sigma \in \Sigma^*$ then*

$$\sigma \in \mathcal{L}(\mathcal{B}_\varphi, q) \iff (\exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma \wedge \sigma' \in \mathcal{L}(\mathcal{A}_\varphi, q)).$$

Example 3 (An example automaton $\mathcal{B}_\varphi$) Let us now further understand how $\mathcal{B}_\varphi$ is constructed for any given $\mathcal{A}_\varphi$ via a simple example. Consider the automaton in Fig. 3 as $\mathcal{A}_\varphi$. In Fig. 7, we can see the automaton $\mathcal{B}_\varphi$ that we obtain from $\mathcal{A}_\varphi$. The only transition in $\mathcal{A}_\varphi$ from an accepting to a non-accepting location is the transition from location l_3 to location l_2. In automaton $\mathcal{B}_\varphi$ this transition is replaced with a self-loop in location l_3. All the other transitions that are in $\mathcal{A}_\varphi$ remain in $\mathcal{B}_\varphi$.

The implementation of function $\kappa_{\psi,\varphi}$ is given by the next theorem.

Theorem 2 *If $\sigma \in \Sigma^*$, $p = \delta_\psi(q_\psi, \sigma)$, and $q = \delta_\varphi(q_\varphi, \sigma)$ then*

$$\kappa_{\psi,\varphi}(\sigma) \iff \mathcal{L}(\mathcal{A}_\psi \times \overline{\mathcal{B}_\varphi}, (p,q)) = \emptyset.$$

[5] Co-safety properties are extension-closed languages by definition.

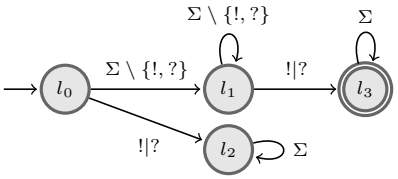


Fig. 7: Automaton $\mathcal{B}_\varphi$

Proof We have:

$\kappa_{\psi,\varphi}(\sigma)$
$\Leftrightarrow$ {Definition of $\kappa_{\psi,\varphi}$, $\psi = \mathcal{L}(\mathcal{A}_\psi, q_\psi)$, and $\varphi = \mathcal{L}(\mathcal{A}_\varphi, q_\varphi)$}
$\forall \sigma'' \in \Sigma^*$: $\sigma \cdot \sigma'' \in \mathcal{L}(\mathcal{A}_\psi, q_\psi) \implies (\exists \sigma' \preccurlyeq \sigma'' : \sigma \cdot \sigma' \in \mathcal{L}(\mathcal{A}_\varphi, q_\varphi))$
$\Leftrightarrow$ {Lemma 1, $p = \delta_\psi(q_\psi, \sigma)$, and $q = \delta_\varphi(q_\varphi, \sigma)$ }
$\forall \sigma'' \in \Sigma^* : \sigma'' \in \mathcal{L}(\mathcal{A}_\psi, p) \implies (\exists \sigma' \preccurlyeq \sigma'' : \sigma' \in \mathcal{L}(\mathcal{A}_\varphi, q))$
$\Leftrightarrow$ {Lemma 11}
$\forall \sigma'' \in \Sigma^* : \sigma'' \in \mathcal{L}(\mathcal{A}_\psi, p) \implies \sigma'' \in \mathcal{L}(\mathcal{B}_\varphi, q)$
$\Leftrightarrow$ {Properties of sets and automata}
$\mathcal{L}(\mathcal{A}_\psi \times \overline{\mathcal{B}_\varphi}, (p, q)) = \emptyset$

Theorem 2 shows that testing $\kappa_{\psi,\varphi}(\sigma)$ reduces to checking emptiness of a regular language.

Enforcement algorithm. Let us now see the algorithm in detail, that requires automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ as input. Algorithm Enforcer (see Algorithm 1) is an infinite loop that scrutinizes the system for input events. In the algorithm, p holds the current state of automaton $\mathcal{A}_\psi$ and q holds the current state of $\mathcal{A}_\varphi$. Initially p, q are assigned the initial states of automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$, respectively. Sequence σ_c corresponds to σ_c in the functional definition, and contains the sequence of events that are already received, and not released as output yet. Automaton $\mathcal{C}$ is the product of the automata $\mathcal{A}_\psi$, and $\overline{\mathcal{B}_\varphi}$. Primitive await_event is used to wait for a new input event. Function release takes a sequence of events, and releases them as output of the enforcer. The algorithm proceeds as follows. The memory σ_c is initialized to ϵ, and the current state information of $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ are initialized with their initial states. It then enters an infinite loop where it waits for an input event. Upon receiving an event a, the current states of $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ are updated, with the state that we reach in these automata, from their current state, reading event a. Later, the algorithm checks whether the language accepted by the automaton $\mathcal{C}$ from state (p, q) is empty [6]. If the language accepted by the automaton $\mathcal{C}$ from state (p, q) is empty, then all the events that are in the memory of the enforcer σ_c followed by the received event a are released as output, and the memory of the enforcer is emptied (σ_c is set to ϵ). Otherwise the event a is added to the memory of the enforcer.

The next lemma shows that this algorithm implements an on-line version of the function $E_{\psi,\varphi}$.

[6] The language accepted by the automaton $\mathcal{C}$ from state (p, q) is empty if no accepting state is reachable from state (p, q) in $\mathcal{C}$.

Algorithm 1 Enforcer

1: $\sigma_c \leftarrow \epsilon$
2: $p,\ q \leftarrow q_\psi,\ q_\varphi$
3: $\mathcal{C} \leftarrow \mathcal{A}_\psi \times \overline{\mathcal{B}_\varphi}$
4: **while** true **do**
5: $\quad a \leftarrow \mathsf{await\_event}()$
6: $\quad p,\ q \leftarrow \delta_\psi(p,a),\ \delta_\varphi(q,a)$
7: $\quad$ **if** $\mathcal{L}(C,(p,q)) = \emptyset$ **then**
8: $\quad\quad \mathsf{release}(\sigma_c \cdot a)$
9: $\quad\quad \sigma_c \leftarrow \epsilon$
10: $\quad$ **else**
11: $\quad\quad \sigma_c \leftarrow \sigma_c \cdot a$

Lemma 12 *If σ is the sequence of events received so far by the enforcement algorithm, and if $\sigma_1, \cdots, \sigma_k$ are the sequences released by the algorithm for σ, then*

$$E_{\psi,\varphi}(\sigma) = \sigma_1 \cdot \ldots \cdot \sigma_k \text{ and } \sigma = E_{\psi,\varphi}(\sigma) \cdot \sigma_c$$

where σ_c corresponds to σ_c in the algorithm, equivalent to σ_c in the definition of $E_{\psi,\varphi}$.

Lemma 12 states that for input σ, if we concatenate the output sequences released by the enforcement algorithm, it will be equal to the output of the enforcement function $E_{\psi,\varphi}(\sigma)$. The input sequence σ is equal to the output of the enforcement function $E_{\psi,\varphi}(\sigma)$ followed by the sequence in the memory of the enforcer σ_c.

Proof of Lemma 12 is given in Appendix A.1 in page 42.

Remark 6 (Complexity) The predictive runtime enforcement method has an off-line and an on-line component. In particular, the product automaton $\mathcal{C}$ computed in line 3 of Algorithm 1 can be computed off-line, before the actual on-line monitoring starts. In fact, the test for emptiness in line 7 of the algorithm can also be computed off-line, for every possible pair of states (p, q) in the product (how to check emptiness is well-known in automata theory). Then the results can be stored in a table with size the number of states in the product state space, i.e., the product of the states in $\mathcal{A}_\psi$ and in $\mathcal{B}_\varphi$. This results in quadratic space complexity, but constant time complexity for the on-line emptiness check. The 1-step reaction implemented in line 6 can also be done in constant time, by storing the transition tables of the two automata (these can be stored separately for each automaton, therefore requiring less space than the product). Overall, this gives a constant time on-line complexity for Algorithm 1.

3.6 Applications of predictive RE

In this section, we will discuss another application of predictive RE. In Section 3.1, we already discussed few abstract examples related to enforcing file format requirements, and monitoring communication. Related to monitoring communication, let us now consider a specific communication protocol.

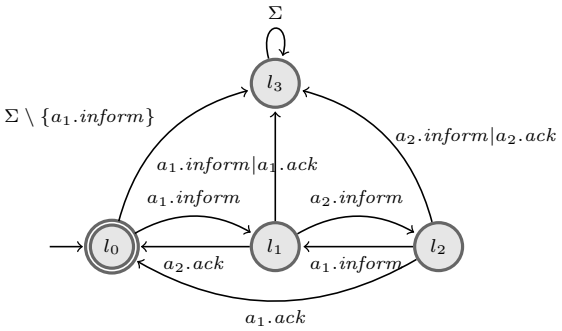
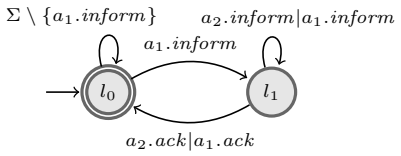


Fig. 8: Datasync protocol model: ψ

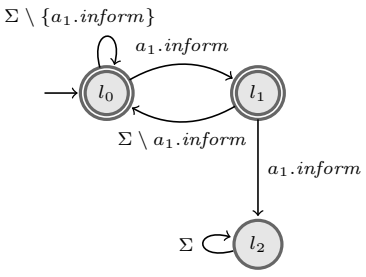
Communication Protocol in Multi-agent Systems. A Multi-agent system (MAS) [28] is a system composed of multiple intelligent agents that interact with each other. Each agent is considered to be an autonomous entity such as a software program. Agents are asynchronous, and communicate via message passing.

Consider some agents working together in an environment. Whenever an agent makes some change to the environment, it informs the other agents to have a common perception of the environment, and the agents use the *datasync* protocol [21] for this purpose. In the area of multi-agent systems, automata have been used widely to represent communication protocols such as the *continuous update* protocol and the *datasync* protocol [21].

The *datasync* protocol as an automaton is illustrated in Fig. 8. In this example, we consider two agents a_1 and a_2. Agent a_1 informs the other agent of changes to the environment by sending an $a_1.inform$ action. Agent a_2 can respond by sending back an acknowledgement $a_2.ack$ if it accepts the change or correct it by responding with an $a_2.inform$ message. The set of actions is $\Sigma = \{a_1.inform, a_2.inform, a_1.ack, a_2.ack\}$.



(a) Property φ_1



(b) Property φ_2

Fig. 9: Properties φ_1 and φ_2

Consider that the enforcement mechanism on agent a_1 has some knowledge of the *datasync* protocol (ψ defined by the automaton in Fig. 8). We can synthesize enforcers for properties such as:

φ_1 Every $inform$ action from agent a_1 (i.e., every $a_1.inform$ action) should eventually end with an ack action from agent a_1 or a_2 (i.e., an action from $\{a_1.ack, a_2.ack\}$).

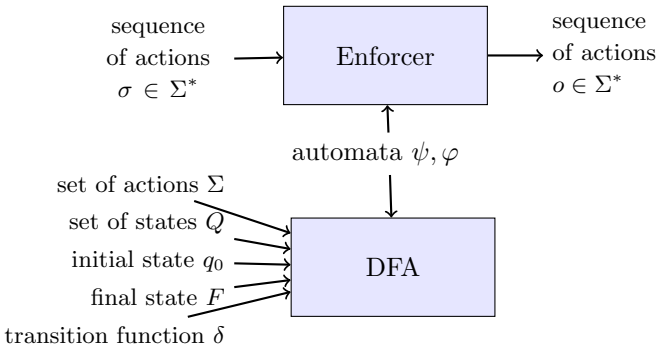


Fig. 10: Implementation overview

In between $inform$ and ack there can be some $inform$ actions. This property is defined by the automaton in Fig. 9a.

φ_2 Agent a_1 cannot send two consecutive $inform$ messages. This property is defined by the automaton in Fig. 9b.

Without knowledge of ψ and without prediction, when the enforcer for property φ_1 receives an $inform$ action it will not release it and it will keep waiting for an ack, resulting in a deadlock situation. Thus, without prediction, properties such as φ_1 cannot be enforced in practice. Using predictive RE, from the provided knowledge of the datasync protocol, the enforcer learns that it will eventually receive an ack action upon receiving an $inform$ action from agent a_1.

3.7 Implementation and evaluation

In this section, we will discuss about an implementation of Algorithm 1 in Section 3.7.1, and the evaluation of its performance using examples from different applications in Section 3.7.2.

3.7.1 Implementation

The predictive enforcement monitoring algorithm described in Section 3.5 is implemented in 500 lines of code in Python. The functionality is divided into two modules as shown in Fig. 10. The *DFA* module contains all the functionality related to defining automata, operations on automata such as negation and product, and checking emptiness. The *Enforcer* module implements the predictive enforcement algorithm described in Section 3.5. The implementation with documentation and some examples are available for download at: https://github.com/SrinivasPinisetty/PredictiveRE.

The *enforcer* method in the module *Enforcer* is invoked with two automata defining ψ and φ, and a sequence of events. In order to use the *enforcer* method, the properties ψ and φ should be described in the intended format using the *DFA* module. Fig. 11 presents the automaton in Fig. 3 described in the intended format using the *DFA* module. A complete example illustrating how the *enforcer* method is invoked is presented in Appendix B.

```
phi = Automata.DFA(
['a', 'b', 'c' ,'?','!'],
['l0','l1', 'l2', 'l3' ],
'l0',
lambda q: q in ['l3'],
lambda q, x: {
        ('l0', 'a') : 'l1',
        ('l0', 'b') : 'l1',
        ('l0', 'c') : 'l1',
        ('l0', '?') : 'l2',
        ('l0', '!') : 'l2',
        ('l1', 'a') : 'l1',
        ('l1', 'b') : 'l1',
        ('l1', 'c') : 'l1',
        ('l1', '?') : 'l3',
        ('l1', '!') : 'l3',
        ('l2', 'a') : 'l2',
        ('l2', 'b') : 'l2',
        ('l2', 'c') : 'l2',
        ('l2', '?') : 'l2',
        ('l2', '!') : 'l2',
        ('l3', 'a') : 'l2',
        ('l3', 'b') : 'l2',
        ('l3', 'c') : 'l2',
        ('l3', '?') : 'l2',
        ('l3', '!') : 'l2',
        }[(q, x)]
)
```

Fig. 11: Example: definition of automaton in Fig. 3

3.7.2 Evaluation

Using some example properties based on real applications, we evaluated the performance of the Python implementation. As we discussed in Remark 6, the product automaton $\mathcal{C}$ in line 3 of Algorithm 1 is computed off-line. Moreover, the emptiness check for every state (p, q) in the product automaton $\mathcal{C}$ is also computed off-line (i.e., before entering the infinite loop (line 4 in Algorithm 1)) and the results are stored in a table.

We will focus on benchmarking the total off-line time required (i.e., the time taken for computing the automaton $\mathcal{C}$ from $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ plus the time taken for computing the table containing the results of the emptiness checks for each state in $\mathcal{C}$). We will also measure the space used for storing the emptiness check table. We will consider examples varying in the number of states in the automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ and measure the performance of the Python implementation. Experiments were conducted on an Intel Core i5-4210U at 1.70GHz CPU, with 4 GB RAM, and running on Windows 7. The reported numbers are mean values over 1000 runs.

Examples. Results of the performance analysis for examples considered from different application domains are presented in Table 2.

Table 2: Performance analysis.

Example	*States ($\mathcal{A}_\psi$)*	*States ($\mathcal{A}_\varphi$)*	*Time (Sec.)*	*Size (Entries)*	*Size (Bytes)*
`FileFormat1`	1	4	0.000139	4	115
`FileFormat2`	5	4	0.000616	20	397
`DataSync1`	4	3	0.000473	12	253
`DataSync2`	4	9	0.001445	36	806
`TCP1`	7	6	0.003868	42	868
`TCP2`	7	18	0.014691	126	2678
`TCP3`	7	54	0.055115	378	8386
`TrafficLight1`	11	3	0.001077	33	702
`TrafficLight2`	11	9	0.003653	99	2049
`TrafficLight3`	11	27	0.012255	297	6260
`TrafficLight4`	11	81	0.049394	891	19430
`TrafficLight5`	11	243	0.224003	2673	61521

- Examples `FileFormat1` and `FileFormat1` are related to enforcing file format requirements described in Section 3.1.
- Examples `DataSync1` and `DataSync1` are related to monitoring communication, in particular the datasync protocol example described in Section 3.6, where the automaton $\mathcal{A}_\psi$ in both these examples is the property ψ (described by the automaton in Figure 8). The automaton $\mathcal{A}_\varphi$ in `DataSync1` is the property φ_1 (defined by the automaton in Figure 9a), and in `DataSync2` the automaton $\mathcal{A}_\varphi$ is the property obtained by taking the conjunction of properties φ_1 and φ_2 (i.e., the automaton obtained by taking the conjunction of the automata in Figure 9a and 9b).
- Examples `TCP1`, `TCP2` and `TCP3` are related to using the enforcer as a firewall or a NID to detect and prevent some attacks. The automaton defining the input property ψ in these examples models the TCP connection status, described in [27], and different properties to enforce are considered such as *"Each connection should start with S. At most 4 consecutive S actions are allowed."*, and *"Data transfer can occur only after connection is established."*. In each example, the number of properties enforced is incremented, resulting in an increase in the number of states of the automaton defining φ.
- Examples `TrafficLight1` to `TrafficLight5` are related to the traffic light controller application. In this example, the input property ψ defines a simple traffic light controller model, and different properties to enforce are considered such as *"No two consecutive red signals."* and *"A red signal should be immediately followed by a green signal."*. In each example, the number of properties to enforce is incremented.

Results. In Table 2, entry *Example* is the name of the example, *States ($\mathcal{A}_\psi$)* is the number of states in the automaton $\mathcal{A}_\psi$ and *States ($\mathcal{A}_\varphi$)* is the number of states in the automaton $\mathcal{A}_\varphi$. The entry *Time (Sec.)* presents the total (off-line) time (time required to compute the automaton $\mathcal{C}$ from $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$, and the table containing results of emptiness check for every state in $\mathcal{C}$). The number of entries in the emptiness check table (which is equal to the number of state in $\mathcal{C}$) is presented in the entry *Size (Entries)*, and the entry *Size (Bytes)* shows the size of this table in bytes.

From Table 2, we can notice that the number of entries in the emptiness check table (which is equal to the number of states in the automaton $\mathcal{C} = \mathcal{A}_\psi \times \overline{\mathcal{B}_\varphi}$) increases with increase in the number of states in $\mathcal{A}_\psi$ and (or) $\mathcal{A}_\varphi$. The size of the table in bytes increases linearly with increase in the number of entries in the emptiness check table. Regarding the off-line time, as expected we can notice that it increases with increase in the number of states in $\mathcal{A}_\psi$ and (or) $\mathcal{A}_\varphi$ (resulting in increase in the number of states in the automaton $\mathcal{C}$). We can notice that the time increase is not linear because the time required to compute emptiness check is not linear on the number of states of $\mathcal{C}$.

4 Predictive runtime enforcement of timed properties

We extend the predictive RE framework presented in Section 3 to enforce timed properties. In this section, φ and ψ are timed properties defined as deterministic and complete timed automata $\mathcal{A}_\varphi$ (with semantics $[\![\mathcal{A}_\varphi]\!]$) and $\mathcal{A}_\psi$ (with semantics $[\![\mathcal{A}_\psi]\!]$) respectively. Inputs and outputs of an enforcer are now timed words.

We first recall some results on runtime enforcement for timed properties without prediction in Section 4.1, where the system being monitored is considered as a black-box. Later in Section 4.2, we motivate via examples the interests and advantages of having prediction for runtime enforcement in the timed setting. The soundness, monotonicity [7] and transparency constraints from the non-predictive case can be adapted in a straightforward manner. For predictive runtime enforcement, similar to the untimed case, we need an additional constraint, namely *urgency* which is not straightforward to adapt from the untimed setting to the timed setting. We formally describe the predictive runtime enforcement problem of timed properties in Section 4.3. Later in Section 4.4, we provide a solution to the predictive RE problem, defining an enforcement mechanism as a function that transforms timed words, and finally in Section 4.5 we discuss about an implementation of the predictive RE problem for timed properties.

4.1 Runtime enforcement of timed properties without prediction

Timed properties are more precise to specify desired behaviors of systems since they allow to explicitly state how time should elapse between events. When we consider timed properties (over finite sequences), in addition to the order of events, their occurrence time also affects the satisfaction of the property. Enforcement monitors for timed properties can be useful in various application domains [16]. For instance, in the context of security monitoring, enforcers can be used as firewalls to prevent denial of service attacks by ensuring a minimal delay between input events (carrying some request for a protected server).

Some earlier endeavors [18,17,16,19,11], describe how to synthesize enforcers for timed properties. Timed automaton is the model used to formally define a property from which an enforcer is synthesized. All regular timed properties specified by deterministic timed automata are supported in the framework proposed in [17,19]. The considered enforcement mechanisms are *time retardants*, i.e., their main enforcement primitive consists in delaying the received events.

In the timed setting [19], an enforcement mechanism should be *sound*, which means that it should correct input words according to the property φ if possible, and otherwise produce

[7] In some earlier works [17,19], constraint monotonicity is called a *physical* constraint.

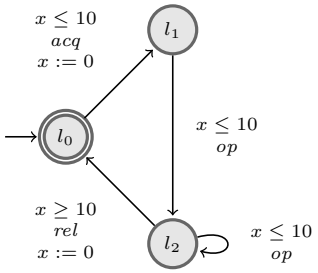


Fig. 12: TA defining property to enforce: φ

an empty output. Second, it should be *transparent*, which means that it is only allowed to shift events in time while keeping their order (such behavior is referred to as time retardants). Third, it should satisfy the *monotonicity* constraint reflecting the streaming of events: the output sequence can only be modified by appending new events to its tail.

4.2 Motivating examples

Let us now see how prediction (using some knowledge of the system behavior) will help in the timed setting. For real-time systems, it is certainly advantageous if it is possible to release events as output earlier. Moreover, in the timed setting some properties that become non-enforceable because of holding some events in the buffer (and delaying them) can be enforced by predicting future input events and releasing events as output earlier.

Remark 7 Completeness. Regarding completeness of timed automata examples, for readability we omit a *trap* location (i.e, a non-accepting location with a self-loop over all actions) and its incoming transitions. If no transition can be triggered upon receiving an event, a TA implicitly moves to a non-accepting trap location.

4.2.1 Example1: Reduce output dates with prediction

Let us consider the situation where a process accesses a resource via three interactions with the resource: acquisition (acq), release (rel), and a specific operation (op). Consider the following requirement with timing constraints (extracted from examples described in [11]), defined by the automaton in Fig. 12. "*The process should behave in a transactional manner, where each transaction consists of an acquisition of the resource, at least one operation on it, and then a release of it. After the resource is acquired by a process, the operations on the resource by that process should be done within 10 time units. After the resource is acquired, it should not be released by the process before 10 time units. There should be no more than 10 time units without any ongoing transaction.*"

Without prediction. Let us consider the input sequence $\sigma_1 = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel)$ (where each event is composed of an action, and a date indicating the time instant at which the action is received as input). The enforcer receives the first action acq at $t = 1$, followed by op at $t = 2$, etc.

According to the enforcement mechanism for timed properties proposed in [19], at $t = 1, 2, 2.4$, when the enforcer receives the actions, it cannot release them as output

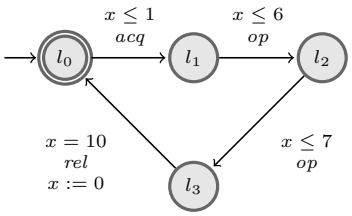


Fig. 13: TA defining possible input sequences: ψ_1

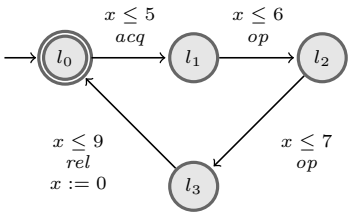


Fig. 14: TA defining possible input sequences: ψ_2

but memorizes them since, upon each reception, the sequence of actions it received so far cannot be delayed to satisfy the property φ. At $t = 3$, upon the reception of action rel, the sequence received so far can be delayed to satisfy the property φ. Thus, the date associated with the first action acq is set to 3. The output of the enforcer for σ_1 is $(3, acq) \cdot (3, op) \cdot (3, op) \cdot (13, rel)$. To satisfy the timing constraint on release actions after acquisitions, the date associated to the last event rel is set to 13.

Consider another input sequence $\sigma_2 = (0, acq) \cdot (2, op) \cdot (2.4, op) \cdot (10, rel)$. The enforcer observes the required sequence of actions only at $t = 10$. Thus, the date associated with the first action acq is set to 10. The next two op actions are also released as output at $t = 10$. The date associated to the last event rel is set to 20 (to satisfy the timing constraint that there should be a delay of at least 10 time units between acquisition and release of a resource). Thus, the output for σ_2 will be $(10, acq) \cdot (10, op) \cdot (10, op) \cdot (20, rel)$.

With prediction (property φ, input property ψ_1). Now, assume that the enforcer has some knowledge of the behavior of the processes defined by the automaton in Fig. 13. The enforcer knows that all the input sequences that it receives satisfy ψ_1. Consider again the input sequence $\sigma_2 = (0, acq) \cdot (2, op) \cdot (2.4, op) \cdot (10, rel)$. Notice that $\sigma_2 \in \psi_1$. Now, instead of waiting until 10 time units to start releasing events, after observing the first event acq at 0 time units, from ψ_1, the enforcer knows for sure that within 10 time units, it will receive two op actions followed by a rel action. Thus, the enforcer can release acq as output immediately at 0 time units. The next two op actions can be also released at time instants when they are received (2 and 2.4 respectively). The enforcer also knows that rel will be received exactly at 10 time units. Thus, the last action rel also can be released as output at the same time instant. The output of the enforcer will be $(0, acq) \cdot (2, op) \cdot (2.4, op) \cdot (10, rel)$. Since $\psi_1 \subseteq \varphi$, given any input sequence that belongs to ψ_1, predictive enforcer for φ can release any event as output immediately after receiving it.

With prediction (property φ, input property ψ_2). Now, consider ψ_2 instead of ψ_1. The enforcer knows that all the input sequences that it receives satisfy ψ_2. Consider the input

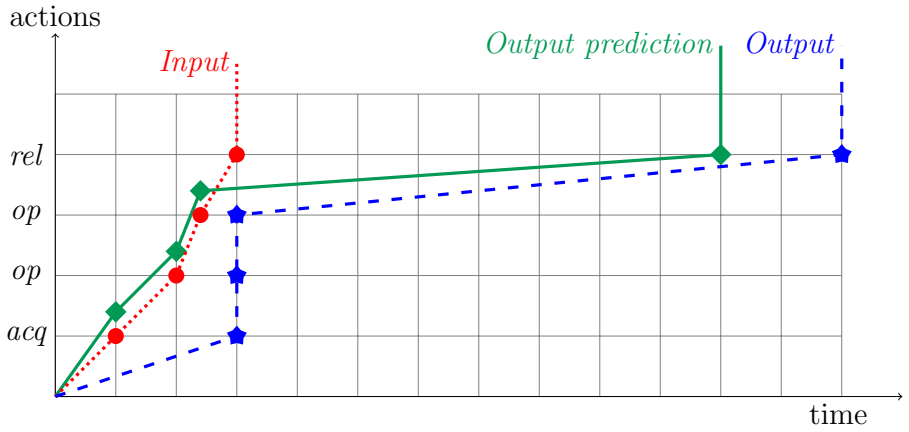


Fig. 15: Property φ, input property ψ_2: input word σ_1 (red, circle), non-predictive enforcer output (blue, star), predictive enforcer output (green, diamond). The diamonds are shifted upwards not to overlap with the circles.

sequence $\sigma_1 = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel)$. Now, instead of waiting until 3 time units to start releasing events, after observing the first event acq at 1 time units, from ψ_2, the enforcer knows for sure that within 9 time units, it will receive two op actions followed by a rel action. Thus, the enforcer can release acq as output immediately at 1 time units. The next two op actions can be also released at time instants when they are received (2 and 2.4). Only the last event rel needs to be delayed for some additional time (in order to satisfy the timing constraint). The output of the enforcer will be $(1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (11, rel)$. Notice that the output sequence $(1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (11, rel)$ belongs to φ but it does not belong to ψ_2. This is because of introducing additional delays between some actions. Such a situation can never occur in the untimed setting. We never change the input sequence and only block when it is not possible to output (thus the output is a prefix of the input). In the untimed case, if all the events are released as output, the input and output sequences will be equal and it belongs to both ψ and φ. In the timed case, blocking affects the absolute time (modifying the date at which the event is released as output).

Fig. 15 illustrates the enforcement mechanism behavior without and with prediction (given ψ_1) when correcting the input sequence σ_1 (dates in abscissa and actions in ordinate). The dashed curve in red color represents the input sequence, the solid curve in blue color represents the output sequence without prediction, and the solid curve in green color represents the output sequence with prediction.

Remark 8 Notice that the untimed projections of the output (considering only the sequence of actions, ignoring dates) are identical with and without prediction in this example. But prediction allows to output some events earlier (output dates of some events can be less). In this particular example, notice that the dates of all the events in the output are less with prediction, compared to the output without prediction.

4.2.2 Example2: Enforce more properties with prediction

Consider again the property φ defined by the automaton in Fig. 12.

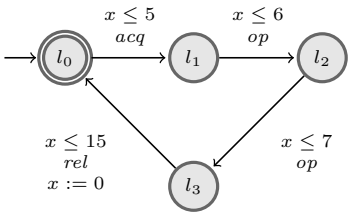


Fig. 16: TA defining possible input sequences: ψ_3

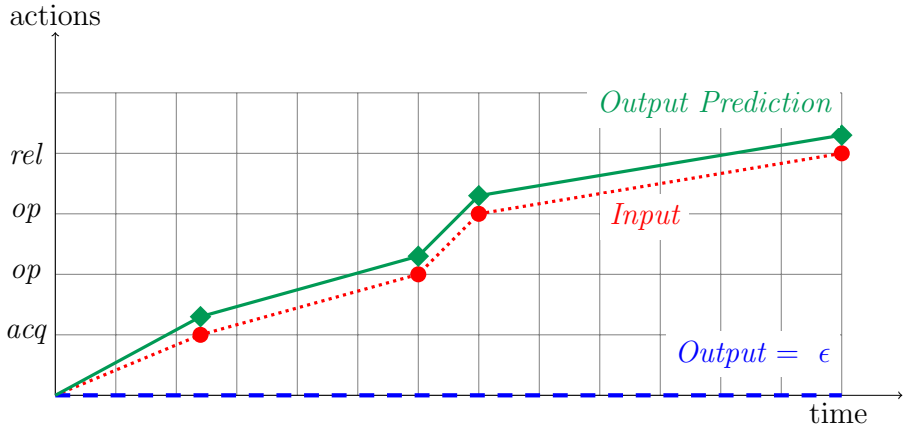


Fig. 17: Property φ, input property ψ_3: input word σ_3 (red, circle), non-predictive enforcer output (blue, star), predictive enforcer output (green, diamond). The diamonds are shifted upwards not to overlap with the circles.

Without prediction. For some non-safety timed properties such as the property defined by the automaton in Fig. 12, the enforcement mechanism in [19] fails from preserving correct sequences. Such properties are described as non-enforceable properties in [19][8].

For example, consider the input sequence $\sigma_3 = (2.4, acq) \cdot (6, op) \cdot (7, op) \cdot (13, rel)$. We shall see that though σ_3 satisfies φ, the output of the enforcer will be ϵ. The enforcer observes actions acq followed by two op actions and a rel action only at 13 time units. Thus, without prediction, the date associated with the first action acq should be at least 13 time units. However, if the enforcer chooses a date greater than 10 for the first action acq, the timing constraint cannot be satisfied. Consequently, the output of the enforcer will be ϵ for the considered input sequence. The enforcer without prediction cannot release any event as output, since it has to wait until it receives action rel to start releasing the previous received actions (acq and op), that affects the date of the first action, falsifying the timing constraint.

With prediction. Let ψ_3 (defined by the TA in Fig. 16) define the set of input sequences. The enforcer knows that all the input sequences that it receives satisfy ψ_3. Consider again the same input sequence $\sigma_3 = (2.4, acq) \cdot (6, op) \cdot (7, op) \cdot (13, rel)$. Now, instead of

[8] Remark 3 in [19] describes non-enforceable timed properties.

waiting until 13 time units to start releasing events, after observing the first event acq at 2.4 time units, from ψ_3, the enforcer knows for sure that within 7 time units, it will receive two op, and within 15 time units the resource will be released (the enforcer will receive a rel action). The enforcer can thus release acq as output immediately at 2.4 time units. The next two op actions and rel action can be also released at time instants when they are received (6, 7 and 13 respectively). The output of the enforcer will be equal to the input $(2.4, acq) \cdot (6, op) \cdot (7, op) \cdot (13, rel)$. Thus the property becomes enforceable given that the input word of the enforcer for φ belongs to ψ_3. Fig. 17 illustrates the enforcement mechanism behavior without and with prediction (given ψ_3) when correcting the input sequence σ_3 to enforce property φ.

4.2.3 Preliminaries to RE of timed properties

In addition to the prefix order $\preccurlyeq$, we will use the following partial orders on timed words.

Delaying order $\geq_d$: For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, $\sigma' \geq_d \sigma$ iff they have the same untimed projection but the dates of events in σ' are greater than or equal to the dates of corresponding events in σ.

In other words, sequence σ' is obtained from σ by keeping all actions, but with a potential increase in dates. Formally:

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

For example, $(4, a) \cdot (7, b) \cdot (9, c) \geq_d (3, a) \cdot (5, b) \cdot (8, c)$.

Delaying prefix order $\preccurlyeq_d$: For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, σ' *delays* σ (denoted as $\sigma' \preccurlyeq_d \sigma$) iff the untimed projection of σ' is a prefix of the untimed projection of σ, but the dates of events in σ' may exceed the dates of corresponding events in σ.

In other words, sequence σ' is obtained from σ by keeping a prefix of actions, but with a potential increase in dates of the considered prefix. Formally:

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

For example, $(4, a) \cdot (7, b) \preccurlyeq_d (3, a) \cdot (5, b) \cdot (8, c)$.

We have that $\sigma' \geq_d \sigma$ if and only if $\sigma' \preccurlyeq_d \sigma \wedge |\sigma'| = |\sigma|$.

4.3 Predictive enforcement monitoring of timed properties

Similar to the untimed setting in Section 3.2, several constraints are required on how an enforcement function transforms words. Input and output are timed words over Σ. The input timed word σ belongs to the input property $\psi \subseteq \mathrm{tw}(\Sigma)$ and the property to enforce is $\varphi \subseteq \mathrm{tw}(\Sigma)$. Our enforcer can only introduce some delays between events if necessary, and block when a violation is detected. Similar to the untimed setting in Section 3.2, it cannot insert (or suppress) events, and cannot change their order.

Definition 5 (Constraints on an enforcement mechanism) Given properties $\psi, \varphi \subseteq \mathrm{tw}(\Sigma)$, an enforcement function $E_{\psi,\varphi} : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, should satisfy the following constraints:
- **Soundness:**

$$\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \neq \epsilon \implies E_{\psi,\varphi}(\sigma) \in \varphi. \qquad \textbf{(SndT)}$$

- **Transparency:**

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_{\psi,\varphi}(\sigma) \preccurlyeq_d \sigma. \qquad \textbf{(TrT)}$$

- **Monotonicity:**

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

– Soundness (**SndT**) means that for any input word belonging to ψ, if the output of the enforcer is not ϵ, then it must satisfy φ. This constraint is similar to the soundness constraint in the untimed setting.
– Transparency (**TrT**) expresses that for any input timed word, the output timed word is a delayed prefix of the input. Note that in the untimed setting, the output word is a prefix of the input word. But, in the timed setting, delaying some events affects the output dates of those events as we saw in some examples in Section 4.2. Thus, in the timed setting, the enforcement mechanism is allowed to increase dates of events while preserving their order.
– The monotonicity constraint is based on the fact that, over time, the enforcement function outputs a continuously-growing sequence of events. The constraint (**MoT**) means that the output produced for an extension σ' of an input timed word σ extends the output produced for σ. The output for a given input can be modified by only appending new events (with greater dates).

The soundness, transparency, and monotonicity constraints are similar to non-predictive enforcement of timed properties [19]. For prediction, similar to the untimed setting in Section 3.2, we should introduce another additional constraint, namely *urgency*. However, it is not straightforward to adapt urgency from the untimed setting to the timed setting. Let us now see some alternatives for urgency starting from a straightforward adaptation of the urgency constraint from the untimed setting.

Remark 9 (UrgencyT1)

$$\begin{aligned}\forall \sigma \in \mathrm{tw}(\Sigma) : (\forall \sigma_{\mathrm{con}} \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \implies \\ \exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma' \preccurlyeq \sigma_{\mathrm{con}} \wedge \sigma \cdot \sigma' \in \varphi) \\ \implies E_{\psi,\varphi}(\sigma) = \sigma\end{aligned}$$

The constraint UrgencyT1 is a straightforward adaptation of the urgency constraint **Ur** in the untimed setting, where we consider timed words over $\mathrm{tw}(\Sigma)$ instead of words over Σ.

This straightforward adaptation works for some simple cases. For example, let the TA in Fig. 12 define φ, and the TA in Fig. 13 define ψ. Consider again the input sequence $\sigma_2 = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (10, rel)$. For input σ_2, UrgencyT1 allows to output each event immediately after it is received as described in Section 4.2.2.

However, it is easy to see that UrgencyT1 does not take into account all cases. It is not always possible to release each event immediately after it is received, and we may have to delay some events (for example to satisfy the timing constraints). For example, let the property to enforce φ be the TA in Fig. 12, and let ψ be the TA in Fig. 14. If we consider the input sequence $\sigma_1 = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel)$, also with prediction, as discussed in Section 4.2, we should delay the last action rel. Upon each event, not all continuations σ_{con} according to ψ may allow to satisfy φ. But, if we consider delaying some events in σ_{con}, it may allow to satisfy φ. In this particular example, at $t = 1$, when we receive acq,

from ψ we know that we will receive two op actions and a rel action before $t = 9$. Thus at $t = 1$, not all continuations allow to satisfy φ, but if we consider delaying some events of all continuations (delaying the last event rel in this example), we know at $t = 1$ that every continuation allows to satisfy φ.

Remark 10 (UrgencyT2) In this alternative urgency constraint, instead of prefixes of all σ_{con}, we consider delayed prefixes of all σ_{con}.

$$\begin{aligned}\forall \sigma \in \text{tw}(\Sigma) : (\forall \sigma_{\text{con}} \in \text{tw}(\Sigma) : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \\ \exists \sigma' \in \text{tw}(\Sigma) : \sigma' \preccurlyeq_d \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \\ \implies E_{\psi,\varphi}(\sigma) = \sigma\end{aligned}$$

For all the examples with prediction that we discussed in Section 4.2, UrgencyT2 constraint allows to release events as expected.

For example, let us again consider the situation that did not work with UrgencyT1, where the property to enforce φ be the TA in Fig. 12, and ψ be the TA in Fig. 14. Consider again the input sequence $\sigma_1 = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel)$. Upon each event, all continuations σ_{con} according to ψ, have a delayed prefix that allows to satisfy φ. Thus, the UrgencyT2 constraint is satisfactory here.

However, UrgencyT2 is also not sufficient since the transparency constraint also allows to increase the dates of events, and the enforcer should build the output incrementally. Thus the input sequence that is already corrected (and released as output) may not be a prefix of the entire observed input sequence.

For example let us again consider the property to enforce φ be the TA in Fig. 12, and ψ be the TA in Fig. 14. Consider the input sequence $\sigma = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel) \cdot (6, acq) \cdot (7, op) \cdot (8, op) \cdot (9.5, rel)$. As we discussed in Section 4.2, the date associated to the first rel in the output is increased to 11 to satisfy the property φ. At $t = 6$, the input sequence observed $(1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel) \cdot (6, acq)$ (followed by a delayed prefix of every continuation) does not belong to φ, but a delayed version of the input sequence (for example $(1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (11, rel) \cdot (11, acq)$), followed by a delayed prefix of any continuation, satisfies φ.

We are interested in online enforcement mechanism, and thus the output should be built incrementally in a streaming fashion. An enforcement function does not have the entire input sequence, and also the length of the input is unknown. Enforcement mechanisms should take decision to release input events as soon as possible. Only when there is no possibility to correct the input (also using the knowledge of all possible input continuations), the enforcement mechanism should wait to receive more events.

We now formulate Urgency focusing on deciding to release events as output as soon as possible using knowledge from ψ, instead of waiting to observe more events.

Definition 6 (Urgency) Given properties $\psi, \varphi \subseteq \text{tw}(\Sigma)$, an enforcement function $E_{\psi,\varphi} : \text{tw}(\Sigma) \to \text{tw}(\Sigma)$, that satisfies soundness, transparency and monotonicity constraints should also satisfy the following urgency constraint:

$$\begin{aligned}&\forall \sigma \in \text{tw}(\Sigma), \forall t \geq \text{end}(\sigma), \forall a \in \Sigma : \\ &(\exists w \in \text{tw}(\Sigma) : E_{\psi,\varphi}(\sigma) \cdot w \geq_d \sigma \cdot (t, a) \ \wedge \ \text{start}(w) \geq t \wedge \\ &\quad \forall \sigma_{\text{con}} \in \text{tw}(\Sigma) : \sigma \cdot (t, a) \cdot \sigma_{\text{con}} \in \psi \implies \\ &\qquad \exists \sigma' \in \text{tw}(\Sigma) : \sigma' \preccurlyeq_d \sigma_{\text{con}} \ \wedge \ E_{\psi,\varphi}(\sigma) \cdot w \cdot \sigma' \in \varphi) \\ &\implies \Pi_\Sigma(E_{\psi,\varphi}(\sigma \cdot (t, a))) = \Pi_\Sigma(\sigma \cdot (t, a)).\end{aligned} \tag{UrT}$$

The urgency constraint (**UrT**) expresses that the enforcer should decide to output events as soon as possible without waiting to observe more events. For any input timed word σ, the output of the enforcer $E_{\psi,\varphi}(\sigma)$ is a delayed prefix of σ. The suffix of σ that the enforcer is unable to decide to output after reading σ is $\sigma_{[|E_{\psi,\varphi}(\sigma)|+1\cdots]}$. Consider a timed word of length $|\sigma+1|$ (i.e., σ is extended with one more additional input event (t,a) at time t). The urgency constraint expresses whether the enforcer decides to output the events that remained from σ followed by the event (t,a) (i.e., $\sigma_{[|E_{\psi,\varphi}(\sigma)|+1\cdots]} \cdot (t,a)$) at time t.

The constraint (**UrT**) means that when we consider a timed word of length $|\sigma+1|$ with one more additional event (t,a), the enforcer *decides* to output all the remaining events from $\sigma \cdot (t,a)$ (which is $\sigma_{[|E_{\psi,\varphi}(\sigma)|+1\cdots]} \cdot (t,a)$) immediately at appropriate minimal dates if

- there exists a timed word w which is an extension of $E_{\psi,\varphi}(\sigma)$ starting at or after time t such that $E_{\psi,\varphi}(\sigma) \cdot w$ is a delayed word of the input $\sigma \cdot (t,a)$, and
- for every continuation σ_{con} of $\sigma \cdot (t,a)$, if there is a delayed prefix σ' of σ_{con} such that $E_{\psi,\varphi}(\sigma) \cdot w \cdot \sigma' \in \varphi$.

Thus if the hypothesis of this urgency condition holds, we know for sure that $\mathit{\Pi}_{\Sigma}(E_{\psi,\varphi}(\sigma \cdot (t,a))) = \mathit{\Pi}_{\Sigma}(\sigma \cdot (t,a))$. The enforcer will not wait to receive more events to decide to output events in $\sigma \cdot (t,a)$.

Definition 7 (Timed predictive enforcer) Given properties $\psi, \varphi \subseteq \mathrm{tw}(\Sigma)$, a timed predictive enforcer for ψ, φ is a function $E_{\psi,\varphi} : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, satisfying the constraints **(SndT)**, **(TrT)**, **(MoT)**, and **(UrT)**.

As we shall show later in Section 4.4, for any ψ, φ, a timed predictive enforcer always exists (Theorem 3).

Remark 11 (Uniqueness) The urgency constraint $(\mathbf{UrT})$ is related to deciding to output events as soon as possible, and if the hypothesis of this constraint holds, we know for sure that all the events will be released as output (i.e., the untimed projection of the input and output of the enforcer will be equal). The timed predictive enforcer for some given properties ψ, φ is generally not unique since there can be different choices regarding the exact dates at which the events will be released as output. For example, let the property ψ be the TA in Fig. 14 and the property φ be the TA in Fig. 12. Let the input sequence be $\sigma = (1, acq) \cdot (2, op) \cdot (2.4, op) \cdot (3, rel)$. After observing the first event acq at 1 time units, the hypothesis of the urgency constraint $(\mathbf{UrT})$ will be satisfied, and we know that the event acq can be released as output without waiting to observe more events. However, there are several possible choices regarding the exact date at which the event acq will be released as output. It can be released at the time instant when the event is observed (1 time units) or later (for example at 1.15 time units). As described in the following remark, our enforcement function defined later in Section 4.4 always chooses minimal possible output dates.

Remark 12 (Optimal output dates) Whenever the enforcer decides to output some events, it should choose optimal (minimal) possible output dates for those events with respect to the current situation, releasing events as output as soon as possible. Once the enforcer decides to output some events, the output dates for those events cannot be modified in the future.

4.4 Functional definition

In this section, we provide a definition of a timed predictive enforcer as a function that builds the output incrementally, taking decisions to output events as soon as possible. Whenever

the enforcement function decides to release some events as output, it computes minimal possible output dates for those events.

The definition of the enforcement function shall use the set $\mathrm{CanD}(\sigma)$ of candidate delayed sequences of σ, independently of the properties ψ and φ.

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

$\mathrm{CanD}(\sigma)$ is the set of timed words w that delay σ, and start at or after the ending date of σ (which is the date of the last event of σ).

Given ψ, φ, the enforcement function $E_{\psi,\varphi}$ defines how an input sequence σ is transformed, such that the output of $E_{\psi,\varphi}$ satisfies φ. The enforcement function $E_{\psi,\varphi} : \mathrm{tw}(\Sigma) \rightarrow \mathrm{tw}(\Sigma)$ is defined as $E_{\psi,\varphi}(\sigma) = \Pi_1\left(\mathrm{store}_{\psi,\varphi}(\sigma)\right)$ where $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$. Sequence σ_s is a delayed prefix of the input that is to be released as output. σ_c is a suffix of the input sequence for which the dates at which these events can be released as output cannot be computed yet (i.e., at date $\mathrm{end}(\sigma)$).

Let us now see the definition of the enforcement function $E_{\psi,\varphi}$ in detail.

Definition 8 (Enforcement function) Given ψ, φ, the enforcement function $E_{\psi,\varphi} : \mathrm{tw}(\Sigma) \rightarrow \mathrm{tw}(\Sigma)$ is defined as:

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

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

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

where

$$\kappa_{\psi,\varphi}(\sigma_n, \sigma_s, \sigma'_c) \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma'_c) \cap \mathrm{Sure}_{\psi,\varphi}(\sigma_n, \sigma_s)$$

and

$$\begin{aligned}
\mathrm{Sure}_{\psi,\varphi}(\sigma_n, \sigma_s) \stackrel{\mathrm{def}}{=} \{w \in tw(\Sigma) \mid \forall \sigma_{\mathrm{con}} \in \mathrm{tw}(\Sigma) :\\
\sigma_n \cdot \sigma_{\mathrm{con}} \in \psi \implies \exists \sigma' \in \mathrm{tw}(\Sigma) :\\
\sigma' \preccurlyeq_d \sigma_{\mathrm{con}} \wedge \sigma_s \cdot w \cdot \sigma' \in \varphi\}
\end{aligned}$$

Function $\mathrm{Sure}_{\psi,\varphi}$ takes two timed words as input. Input timed word $\sigma_n = \sigma \cdot (t,a)$, corresponds to the entire input sequence, and σ_s which is a delayed prefix of σ_n is the output of the enforcement function after reading σ. Function $\mathrm{Sure}_{\psi,\varphi}$ returns all the timed words w, such that there is a delayed prefix σ' for every future extension of σ_n, such that what is already decided to be released as output by the enforcer σ_s concatenated with $w \cdot \sigma'$ satisfies the property φ.

Note that $\mathrm{CanD}(\sigma'_c)$ computes all the delayed timed words of σ'_c that start at or after $\mathrm{end}(\sigma'_c)$. Thus the set $\kappa_{\psi,\varphi}(\sigma_n, \sigma_s, \sigma'_c)$ which is $\mathrm{CanD}(\sigma'_c) \cap \mathrm{Sure}_{\psi,\varphi}(\sigma_n, \sigma_s)$ is non-empty if and only if the hypothesis of the *UrT* constraint is satisfied.

Theorem 3 (Soundness, transparency, urgency, and monotonicity constraint) *Given two properties ψ, and φ, the enforcement function $E_{\psi,\varphi}$ as per Definition 8 is a timed predictive enforcer satisfying constraints* **(SndT)**, **(TrT)**, **(MoT)**, *and* **(UrT)**.

According to Theorem 3, for any two properties ψ, and φ, a timed predictive enforcer always exists. Proof of Theorem 3 is given in Appendix A.2.

Remark 13 In the untimed case, for a given input $\sigma \in \psi$, the final output o after reading σ completely will be the same with or without prediction. But in the timed setting, the output sequences may not be equal (since we may output some events earlier with prediction). Moreover, even if we restrict our attention to untimed projections of outputs, those may differ with or without prediction (see examples in Section 4.2).

4.5 Algorithm

In the untimed case, the output word is a prefix of the input word, and if the input word satisfies the property φ, the output will be equal to the input. But, in the timed case, delaying events affects the output dates, thus modifying the output words (though all the events are released as output). Thus, in the timed setting, the output timed word may not be a prefix of the input timed word belonging to ψ (as illustrated via examples in Section 4.2).

There are several aspects that need to be investigated regarding if and how the enforcement function (specifically the function $\kappa_{\psi,\varphi}$) can be computed in the timed setting.

For example, in the untimed setting, from the automaton $\mathcal{A}_\psi = (Q_\psi, q_\psi, \Sigma, \delta_\psi, F_\psi)$ (that defines the input property ψ) and the observed input word σ, the state reached upon reading σ is $q = \delta(\mathcal{A}_\psi, q_\psi)$. The automaton that recognizes all the continuations of the observed input σ, is the automaton $\mathcal{A}_\psi$ starting from the state q.

In the timed setting, from the current observed input σ and the TA defining ψ, we need to first build a finite construction that accepts all the delayed timed words of the timed words w that are continuations of σ such that $\sigma \cdot w \in \psi$. To obtain the TA that recognizes all continuations of σ, similar to the untimed setting, in the TA $\mathcal{A}_\psi$ we treat the state reached upon reading σ as the initial state. Now, we need to find some construction that accepts all the delayed words of this timed automaton. One possible construction is discussed in Remark 14. However, this construction is not finite. How to obtain such a finite construction for a given TA is indeed an interesting problem on itself, and we leave the algorithm for computing the enforcement function in the timed case as future work.

Remark 14 (Delayed version of a timed automaton) Given a timed automaton $\mathcal{A}$, we want to obtain a construction C, that accepts all the delayed timed words of timed words belonging to $\mathcal{L}(\mathcal{A})$. Consider a FIFO queue F of infinite capacity. Whenever a transition is triggered in $\mathcal{A}$, the action triggering this transition is sent to F instead of writing it to the output. Consider another $untimed$ automaton B, where its only possible action is to dequeue actions from the queue F at arbitrary times. $C = A \times F \times B$ recognizes all the delayed timed words of timed words belonging to $\mathcal{L}(\mathcal{A})$.

5 Related work

Runtime enforcement was initiated by the work of Schneider [25] and has been extensively studied from then onwards. According to how a monitor is allowed to correct the input sequence, several models of enforcers (enforcement monitors) have been proposed. Security

automata [25] focused on safety properties, and blocked the execution as soon as an illegal sequence of actions (not compliant with the property) is recognized. Later, several refinements have been proposed such as suppression automata [13] that allowed to suppress events from the input sequence, insertion automata [13] that allowed to insert events to the input sequence, and edit-automata [13] or so-called generalized enforcement monitors [10] that allowed to perform any of these primitives. In all these approaches, the system is considered as a black-box. In our approach, we make use of (any) available knowledge of the system, and we do not allow to suppress or insert events.

Recently, Bloem et al. [3] presented a framework to synthesize enforcement monitors for reactive systems, called *shields*, from a set of safety properties. This work focuses on reactive systems, and it is not possible to block actions and to release them later (or to halt the system). The shield must always act instantaneously (upon erroneous input, some output must be produced instantaneously). In some cases, when a property violation is unavoidable, the shield allows deviation for k consecutive steps. In case if a second violation occurs within k steps, then the shield enters a *fail-safe* mode, where it ensures only correctness, and no longer minimizes deviation. In our approach, when it is not possible to act instantaneously, we allow to buffer input events. Moreover, we release some events as output only after being sure that the property will be satisfied eventually with subsequent output events.

Another recent approach by Dolzehnko et al in [6] introduces Mandatory Result Automata (MRAs). MRAs extend edit-automata [13] by refining the input-output relationship of an enforcement mechanism and thus allow a more precise description of the enforcement abilities of an enforcement mechanism in concrete application scenarios such as the scenario described in Section 3.1.2. In order to handle such scenarios, their approach makes use of knowledge about the actions and their effect on the monitored system (i.e., the input alphabet is split into actions and results). Moreover, the MRA model assumes synchronizable actions (i.e., after receiving an action another action cannot be received until the previous action returned a result). In our approach, we consider actions that are transmitted between (asynchronous) event emitter and receiver and hence do not consider the effect of actions.

All the previously mentioned approaches do not consider any model of the system (the system is considered as a black-box). In [29], Zhang et al. propose predictive semantics for runtime verification, enabling the verification monitor to foresee property satisfaction or violation before the observed execution satisfies or violates it.

Some recent work by Chabot et al. [5] uses knowledge of the program to extend enforcement. In their approach, the monitor's enforcement power is extended by giving it access to statically gathered information about the program's possible behavior. The approach of [5] works for safety properties, but, as the authors explicitly state, there is no guarantee that it would work for non-safety properties. Our approach works for any regular property. Moreover, as discussed in Remark 5, having knowledge of the input has no advantage for safety properties. Furthermore, we also make use of knowledge of the system to also predict possible futures, and to output events earlier whenever possible.

Our work is related to supervisory control [4], where a new "controlled" system is obtained by composing a system with a controller in closed-loop. The controlled system must meet a given specification and does not produce illegal actions as output. In supervisory control the controller controls the system, because it feeds-back into the system, in closed loop. But in our case, the loop remains open. The enforcer does not feed-back into the system. Moreover, it is not mandatory to have a model of the system in our approach. Remark 3 discusses that our constraints reduce to non-predictive case when $\psi = \Sigma^*$.

In some earlier works [18, 17, 19, 16], we presented *runtime enforcement for timed properties*. In [18] we introduced runtime enforcement for timed properties, proposing enforce-

ment monitor synthesis for timed safety and co-safety properties. We later generalized our framework to synthesize an enforcement monitor for any regular timed property [17, 19]. We also extended our work to parametric timed properties, where events are parameterized, allowing events to carry some data values from the monitored system [16]. In all these works, enforcement monitors work as *delayers*, i.e., their main enforcement primitive consists in delaying the received events, to correct the input sequence of timed events according to the property. Whenever the input sequence received cannot be corrected, the monitor blocks those events until it receives more events. In all these works, the system is considered as a black-box. Recently in [23] we considered runtime enforcement for timed properties with some uncontrollable events. An uncontrollable event cannot be blocked/delayed by an enforcement monitor. The system is still considered as a black-box, but we assume here that the enforcement monitor has knowledge about actions that are not controllable.

6 Conclusion and future work

This paper extends existing work in runtime enforcement by proposing a *predictive* RE framework. The framework generalizes RE from black-box to grey-box systems, that is, systems for which some a-priori knowledge is available. We show how knowledge about a system's behavior can benefit enforcement, by allowing a enforcer to anticipate ("predict") future input events and as a result become more responsive at its output. Compared to earlier works on enforcement, this is achieved by introducing an additional constraint called *urgency*. Urgency ensures that enforcers react as soon as possible, often outputting events immediately after they are received, instead of buffering them indefinitely. The property to be enforced as well as the knowledge about the system are modeled as deterministic automata (i.e., regular languages). We show how to synthesize enforcement mechanisms for any regular property and provide algorithms implementing these mechanisms in polynomial memory and constant online time. We also implemented the proposed algorithms in Python. For real-time systems, if it is possible to output earlier (using prediction), it is certainly beneficial. We extended the predictive RE problem to real-time properties.

Several interesting extensions and alternatives remain to be explored in the future. As we discussed in Section 4.5, how to compute the enforcement function in the timed setting remains to be explored. In the untimed case, we implemented the proposed algorithms, and we also briefly discussed some example applications. In the future, we also plan to extend our work experimentally, and further study the feasibility of applying our approach in some particular scenarios.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994), http://dx.doi.org/10.1016/0304-3975(94)90010-8
2. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S.E., Thomas, W. (eds.) Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II. Lecture Notes in Computer Science, vol. 5556, pp. 43–54. Springer (2009), http://dx.doi.org/10.1007/978-3-642-02930-1_4
3. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: Runtime enforcement for reactive systems. In: TACAS. LNCS, vol. 9035. Springer (2015)
4. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)

5. Chabot, H., Khoury, R., Tawbi, N.: Extending the enforcement power of truncation monitors using static analysis. Computers & Security 30(4), 194–207 (2011)
6. Dolzhenko, E., Ligatti, J., Reddy, S.: Modeling runtime enforcement with mandatory results automata. Int. J. Inf. Sec. 14(1), 47–60 (2015), http://dx.doi.org/10.1007/s10207-014-0239-8
7. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems 27(7), 1165–1178 (2008), http://dx.doi.org/10.1109/TCAD.2008.923410
8. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Softw. 19(1), 42–51 (Jan 2002), http://dx.doi.org/10.1109/52.976940
9. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? STTT 14(3), 349–382 (2012)
10. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
11. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Science of Computer Programming 123, 2 – 41 (2016)
12. Finkel, O.: Undecidable problems about timed automata. In: Formal Modeling and Analysis of Timed Systems, pp. 187–199. Springer (2006)
13. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12(3), 19:1–19:41 (Jan 2009)
14. Malan, G.R., Watson, D., Jahanian, F., Howell, P.: Transport and application protocol scrubbing. In: Proceedings IEEE INFOCOM 2000, Israel. pp. 1381–1390 (2000)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
16. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Runtime enforcement of parametric timed properties with practical applications. In: Lesage, J., Faure, J., Cury, J.E.R., Lennartson, B. (eds.) 12th International Workshop on Discrete Event Systems, WODES 2014, 2014. pp. 420–427. International Federation of Automatic Control (2014), http://www.ifac-papersonline.net/Discrete_Event_Systems/12th_International_Workshop_on_Discrete_Event_Systems__2014_/index.html
17. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Runtime enforcement of regular timed properties. In: Proceedings of the ACM Symposium on Applied Computing (SAC-SVT). pp. 1279–1286. ACM (2014)
18. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Timo, O.L.N.: Runtime enforcement of timed properties. In: Qadeer, S., Tasiran, S. (eds.) Proceedings of the Third International Conference on Runtime Verification (RV 2012). Lecture Notes in Computer Science, vol. 7687, pp. 229–244. Springer (2012)
19. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods in System Design 45(3), 381–422 (2014)
20. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. In: Proceedings of the ACM Symposium on Applied Computing (SAC-SVT). p. to appear. ACM (2016)
21. Pitt, J., Mamdani, E.H.: A protocol-based semantics for an agent communication language. In: Dean, T. (ed.) Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages. pp. 486–491. Morgan Kaufmann (1999)
22. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: Learnlib: a framework for extrapolating behavioral models. International Journal on Software Tools for Technology Transfer 11(5), 393–407 (2009), http://dx.doi.org/10.1007/s10009-009-0111-8
23. Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H.: Enforcement of (timed) properties with uncontrollable events. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 542–560. Springer (2015)
24. Rosu, G.: On safety properties and their monitoring. Sci. Ann. Comp. Sci. 22(2), 327–365 (2012)
25. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000), http://doi.acm.org/10.1145/353323.353382
26. Tripakis, S.: Folk theorems on the determinization and minimization of timed automata. Inf. Process. Lett. 99(6), 222–226 (2006), http://dx.doi.org/10.1016/j.ipl.2006.04.015
27. Tuglular, T., Belli, F.: Protocol-based testing of firewalls. In: Formal Methods (SEEFM), 2009 Fourth South-East European Workshop on. pp. 53–59 (Dec 2009)
28. Wooldridge, M.: An Introduction to MultiAgent Systems. Wiley Publishing, 2nd edn. (2009)
29. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: NASA Formal Methods - 4th International Symposium. LNCS, vol. 7226, pp. 418–432. Springer (2012)

A Proofs

A.1 Proofs: untimed setting

In this section, we will discuss proofs of lemmas and theorems in Section 3. ψ and φ in this section are regular properties that are defined by automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$. To ease understanding, for some Lemmas, we provide manual proofs with some explanations. A document with all Isabelle proofs can be accessed from: https://github.com/isabelle-theory/PredictiveRuntimeEnforcement

Proof (of Lemma 3) We shall prove that the urgency constraint (**Ur'**) is weaker than the urgency constraint (**Ur**), i.e.,

$$\begin{aligned}&\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \\ &\quad \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma \\ \implies \\ &\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma \cdot \sigma_{\text{con}} \in \varphi) \\ &\quad \implies E_{\psi,\varphi}(\sigma) = \sigma.\end{aligned}$$

Assume (**Ur**) and for $\sigma \in \Sigma^*$ assume that $(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma \cdot \sigma_{\text{con}} \in \varphi)$ holds. We need to show that $E_{\psi,\varphi}(\sigma) = \sigma$. We have

$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma \cdot \sigma_{\text{con}} \in \varphi)$
$\Leftrightarrow$ $\{\sigma_{\text{con}} \preccurlyeq \sigma_{\text{con}}$ is true$\}$
$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma_{\text{con}} \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma_{\text{con}} \in \varphi)$
$\Rightarrow$ {Existential quantifier introduction}
$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$

Therefore the hypothesis of (**Ur**) is true, so we obtain $E_{\psi,\varphi}(\sigma) = \sigma$.

Proof (of Lemma 4) We shall prove that given properties $\psi, \varphi \subseteq \Sigma^*$, when $\psi = \Sigma^*$, the constraint (**Ur**) is equivalent to the following:

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma.$$

(**Ur**)
$\Leftrightarrow$ {Definition of **Ur**}
$\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma$
$\Leftrightarrow$ $\{\psi = \Sigma^*\}$
$\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma$
$\Leftrightarrow$ {Sub-derivation: $(\forall \sigma_{\text{con}} \in \Sigma^* : \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \Leftrightarrow \sigma \in \varphi$}

- $(\forall \sigma_{\text{con}} \in \Sigma^* : \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$
  $\Rightarrow$ {Universal quantifier elimination}
  $(\exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \epsilon \wedge \sigma \cdot \sigma' \in \varphi)$
  $\Leftrightarrow$ {σ' can only be ϵ}
  $\sigma \in \varphi$
- $\sigma \in \varphi$
  $\Leftrightarrow$ {Properties of words}
  $(\forall \sigma_{\text{con}} \in \Sigma^* : \epsilon \preccurlyeq \sigma_{\text{con}}) \wedge \sigma \cdot \epsilon \in \varphi$
  $\Leftrightarrow$ {Logic}
  $(\forall \sigma_{\text{con}} \in \Sigma^* : \epsilon \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \epsilon \in \varphi)$
  $\Rightarrow$ {Existential quantifier introduction}
  $(\forall \sigma_{\text{con}} \in \Sigma^* : \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$

... $\forall \sigma \in \Sigma^* : \sigma \in \varphi \Rightarrow E_{\psi,\varphi}(\sigma) = \sigma$
$\Leftrightarrow$ {Definition}

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma$$

Proof (of Lemma 5) We shall prove that give properties $\psi, \varphi \subseteq \Sigma^*$, when $\psi \subseteq \varphi$, for any word $\sigma \in \Sigma^*$, the output of the enforcement function is σ ($E_{\psi,\varphi}(\sigma) = \sigma$). Assume $\psi \subseteq \varphi$, then we have:

$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$

$\Leftarrow$ {Existential quantifier introduction}

$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma_{\text{con}} \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma_{\text{con}} \in \varphi)$

$\Leftrightarrow$ {Properties of words}

$(\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \sigma \cdot \sigma_{\text{con}} \in \varphi)$

$\Leftrightarrow$ {Assumption $\psi \subseteq \varphi$}

true

From this, using urgency (**Ur**), we obtain $E_{\psi,\varphi}(\sigma) = \sigma$.

Proof (of Lemma 10) We shall prove the following properties of the enforcement function, and auxiliary functions $\text{store}_{\psi,\varphi}$ and $\kappa_{\psi,\varphi}$.

For all $\sigma, \sigma' \in \Sigma^*$ we have

1. $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c) \implies \sigma = \sigma_s \cdot \sigma_c$
2. $E_{\psi,\varphi}(\sigma) \neq \epsilon \implies \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$
3. $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma')$
4. $\sigma \in \varphi \implies \kappa_{\psi,\varphi}(\sigma)$

Proof (of property 1 of Lemma 10) Let us prove Property 1 using induction on the input sequence σ.

Induction basis. If $\sigma = \epsilon$, from the definition of the enforcement function (Definition 4), $E_{\psi,\varphi}(\sigma) = \epsilon$. From the definition of $\text{store}_{\psi,\varphi}$, $\text{store}_{\psi,\varphi}(\epsilon) = (\epsilon, \epsilon)$. Since $\epsilon = \epsilon \cdot \epsilon$, this property holds for $\sigma = \epsilon$.

Induction step. Assume that for every $\sigma \in \Sigma^*$ of some length $n \in \mathbb{N}$, $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c) \implies \sigma = \sigma_s \cdot \sigma_c$.

We now prove that for any $a \in \Sigma$, property 1 holds for $\sigma \cdot a$. We have the following two possible cases:

- Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = \mathsf{true}$. Since $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is true, according to the definition of $\text{store}_{\psi,\varphi}$, we will have $\text{store}_{\psi,\varphi}(\sigma \cdot a) = (\sigma_s \cdot \sigma_c \cdot a, \epsilon)$. From the induction hypothesis, we have $\sigma_s \cdot \sigma_c = \sigma$. So, we have $\sigma_s \cdot \sigma_c \cdot a \cdot \epsilon = \sigma \cdot a$. Thus, the property holds.
- Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = \mathsf{false}$. Since $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is false, according to the definition of $\text{store}_{\psi,\varphi}$, we will have $\text{store}_{\psi,\varphi}(\sigma \cdot a) = (\sigma_s, \sigma_c \cdot a)$. Using induction hypothesis, we will have $\sigma_s \cdot \sigma_c \cdot a \cdot \epsilon = \sigma \cdot a$. Thus, the property holds.

Proof (of property 2 of Lemma 10) Let us prove Property 2 using induction on the input sequence σ.

Induction basis. If $\sigma = \epsilon$, from the definition of the enforcement function (Definition 4), $E_{\psi,\varphi}(\epsilon) = \epsilon$. Since $E_{\psi,\varphi}(\epsilon) = \epsilon$, property $E_{\psi,\varphi}(\sigma) \neq \epsilon \implies \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$ trivially holds for $\sigma = \epsilon$.

Induction step. Assume that for every σ of some length $n \in \mathbb{N}$, $E_{\psi,\varphi}(\sigma) \neq \epsilon \implies \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$. Let $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$. According to Definition 4, we know that $E_{\psi,\varphi}(\sigma) = \mathit{\Pi}_1\left(\text{store}_{\psi,\varphi}(\sigma)\right) = \sigma_s$.

We now prove that for any $a \in \Sigma$, Property 2 also holds for $\sigma \cdot a$. We have the following two possible cases:

- Case $E_{\psi,\varphi}(\sigma \cdot a) = \epsilon$. The property trivially holds in this case.
- Case $E_{\psi,\varphi}(\sigma \cdot a) \neq \epsilon$. We can also notice from Definition 4 that there are two possible cases based on whether $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is either true or false.
  - Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = true$. In this case, $E_{\psi,\varphi}(\sigma \cdot a) = \sigma_s \cdot \sigma_c \cdot a = \sigma \cdot a$ (from Property 1). Also, $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is true in this case. Thus the property holds for $\sigma \cdot a$.
  - Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = false$. In this case, $E_{\psi,\varphi}(\sigma \cdot a) = E_{\psi,\varphi}(\sigma)$, and using the induction hypothesis, we can conclude that the property holds also for $\sigma \cdot a$.

Proof (of property 3 of Lemma 10)

Let us prove Property 3 using induction on the length of σ'.

Induction basis. If $\sigma' = \epsilon$, then $\sigma = \epsilon$. From the definition of the enforcement function (Definition 4), $E_{\psi,\varphi}(\sigma) = \epsilon$. Since $\epsilon \preccurlyeq \epsilon$, property 3 trivially holds for $\sigma = \epsilon$.

Induction step. Assume that for every $\sigma' \in \Sigma^*$ of some length $n \in \mathbb{N}$, $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma')$ holds.

Let $\text{store}_{\psi,\varphi}(\sigma') = (\sigma'_s, \sigma'_c)$. According to Definition 4, we know that $E_{\psi,\varphi}(\sigma') = \mathit{\Pi}_1\left(\text{store}_{\psi,\varphi}(\sigma)\right) = \sigma'_s$. From property 1, we have $\sigma' = \sigma'_s \cdot \sigma'_c$.

We now prove that for any $a \in \Sigma$, Property 3 holds for $\sigma' \cdot a$. We have the following two possible cases:

- Case $\kappa_{\psi,\varphi}(\sigma' \cdot a) = \mathsf{true}$. Since $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is true, according to the definition of $\mathrm{store}_{\psi,\varphi}$, we will have $\mathrm{store}_{\psi,\varphi}(\sigma' \cdot a) = (\sigma'_s \cdot \sigma'_c \cdot a, \epsilon)$. Consequently, $E_{\psi,\varphi}(\sigma' \cdot a) = \sigma'_s \cdot \sigma'_c \cdot a = \sigma' \cdot a$.
  We now have two subcases based on whether $\sigma = \sigma' \cdot a$ or not.
  - Case $\sigma = \sigma' \cdot a$. We already saw that $\kappa_{\psi,\varphi}(\sigma' \cdot a)$ is true, and $E_{\psi,\varphi}(\sigma' \cdot a) = \sigma' \cdot a = \sigma$. Consequently, we have $\sigma \preccurlyeq E_{\psi,\varphi}(\sigma' \cdot a)$. Thus the property $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \cdot a \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma' \cdot a)$ holds in this case.
  - Case $\sigma \preccurlyeq \sigma'$. From the induction hypothesis, we have $\sigma \preccurlyeq E_{\psi,\varphi}(\sigma') = \sigma'_s$. In this case, we already showed that $E_{\psi,\varphi}(\sigma' \cdot a) = \sigma'_s \cdot \sigma'_c \cdot a$. Since $\sigma \preccurlyeq \sigma'_s$, we also have $\sigma \preccurlyeq \sigma'_s \cdot \sigma'_c \cdot a = E_{\psi,\varphi}(\sigma' \cdot a)$. Thus the property $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \cdot a \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma' \cdot a)$ holds in this case.
- Case $\kappa_{\psi,\varphi}(\sigma' \cdot a) = \mathsf{false}$. Since $\kappa_{\psi,\varphi}(\sigma' \cdot a)$ is false, according to the definition of $\mathrm{store}_{\psi,\varphi}$, we will have $\mathrm{store}_{\psi,\varphi}(\sigma' \cdot a) = (\sigma'_s, \sigma'_c \cdot a)$. In this case, $E_{\psi,\varphi}(\sigma' \cdot a) = \sigma'_s = E_{\psi,\varphi}(\sigma')$.
  We again have two subcases here based on whether $\sigma = \sigma' \cdot a$ holds or not.
  - Case $\sigma = \sigma' \cdot a$. Since $\kappa_{\psi,\varphi}(\sigma' \cdot a)$ is false, the property trivially holds in this case.
  - Case $\sigma \preccurlyeq \sigma'$. From the induction hypothesis, we have $\sigma \preccurlyeq E_{\psi,\varphi}(\sigma') = \sigma'_s$. Since $E_{\psi,\varphi}(\sigma' \cdot a) = E_{\psi,\varphi}(\sigma')$, consequently we have $\sigma \preccurlyeq E_{\psi,\varphi}(\sigma' \cdot a)$. Thus the property $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \cdot a \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma' \cdot a)$ holds in this case.

Proof (of property 4 of Lemma 10) We shall prove that given properties $\varphi, \psi \subseteq \Sigma^*$, and for any word $\sigma \in \Sigma^*$, the following property holds:
$\sigma \in \varphi \implies \kappa_{\psi,\varphi}(\sigma)$.
Assume that $\sigma \in \varphi$. Function $\kappa_{\psi,\varphi}(\sigma)$ is defined as follows:

$$\kappa_{\psi,\varphi}(\sigma) = (\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\mathrm{con}} \wedge \sigma \cdot \sigma' \in \varphi)$$

- $\sigma \in \varphi$
- $\Rightarrow$ {Propositional calculus}
  $(\exists \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi) \Rightarrow \sigma \in \varphi$
- $\Leftrightarrow$ {Predicate calculus}
  $\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \Rightarrow \sigma \in \varphi$
- $\Leftrightarrow$ {Properties of words}
  $\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \Rightarrow \epsilon \preccurlyeq \sigma_{\mathrm{con}} \wedge \sigma \cdot \epsilon \in \varphi$
- $\Rightarrow$ {Existential quantifier introduction}
  $\forall \sigma_{\mathrm{con}} \in \Sigma^* : \sigma \cdot \sigma_{\mathrm{con}} \in \psi \Rightarrow (\exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\mathrm{con}} \wedge \sigma \cdot \sigma' \in \varphi)$
- $\Leftrightarrow$ {Definition}
  $\kappa_{\psi,\varphi}(\sigma)$

Proof (of Lemma 12) We shall prove that the algorithm 1 implements the enforcement function $E_{\psi,\varphi}$ (Definition 4).

That is, we shall prove that if $\sigma = a_1 \cdots a_n$ is the sequence of events received so far by the enforcement algorithm, and if $\sigma_1, \cdots, \sigma_k$ are the sequences released by the algorithm for σ, then

$$E_{\psi,\varphi}(\sigma) = \sigma_1 \cdot \ldots \cdot \sigma_k \text{ and } \sigma = E_{\psi,\varphi}(\sigma) \cdot \sigma_c$$

where σ_c corresponds to σ_c in the algorithm, equivalent to σ_c in the definition of $E_{\psi,\varphi}$.

Let us prove this lemma using induction on the length of the input sequence σ.

Induction basis. If $\sigma = \epsilon$, from the definition of the enforcement function (Definition 4), $\mathrm{store}_{\psi,\varphi}(\epsilon) = (\sigma_s, \sigma_c) = (\epsilon, \epsilon)$, and $E_{\psi,\varphi}(\sigma) = \epsilon$.

Regarding the enforcement algorithm, σ_c in the algorithm is initialized to ϵ, since no event is received yet, this corresponds to the first iteration of the *while* loop, where the process is waiting for an event in line 5. Thus statement *release* in the algorithm is never executed yet. Concatenation of the output sequences released by the algorithm is thus ϵ. Consequently, Lemma 12 holds for $\sigma = \epsilon$.

Induction step. Assume that for every $\sigma \in \Sigma^*$ of some length $n \in \mathbb{N}$, Lemma 12 holds.

Regarding the enforcement function $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$, and $E_{\psi,\varphi}(\sigma) = \mathit{\Pi}_1\left(\mathrm{store}_{\psi,\varphi}(\sigma)\right) = \sigma_s$. After receiving σ, $p = \delta_\psi(q_\psi, \sigma)$, and $q = \delta_\varphi(q_\varphi, \sigma)$. Let σ_c be equal to σ_c in the algorithm, and let the concatenation of output sequences released by the algorithm be equal to $E_{\psi,\varphi}(\sigma) = \sigma_s$.

We now prove that for any $a \in \Sigma$, Lemma 12 holds for $\sigma \cdot a$. We have the following two possible cases:

- Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = \mathsf{true}$. Regarding the enforcement function, since $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is true, according to the definition of $\mathrm{store}_{\psi,\varphi}$, we will have $\mathrm{store}_{\psi,\varphi}(\sigma{\cdot}a) = (\sigma_s{\cdot}\sigma_c{\cdot}a, \epsilon)$. Consequently, $E_{\psi,\varphi}(\sigma{\cdot}a) = \sigma_s \cdot \sigma_c \cdot a = \sigma \cdot a$.
  Regarding the enforcement algorithm, upon receiving event a, after executing step 6 of the algorithm, $p = \delta_\psi(q_\psi, \sigma{\cdot}a)$, and $q = \delta_\varphi(q_\varphi, \sigma{\cdot}a)$. Since $\kappa_{\psi,\varphi}(\sigma{\cdot}a) = \mathsf{true}$, using Theorem 2, $\mathcal{L}(C,(p,q)) = \emptyset$ and statement 8 ($release(\sigma_c{\cdot}a)$) will be executed. Statement 9 will also be executed in this case resetting σ_c in the algorithm to ϵ. Using the induction hypothesis, we know that the concatenation of output sequences released by the algorithm is equal to σ_s. The new sequence released is $\sigma_c \cdot a = E_{\psi,\varphi}(\sigma \cdot a)$. Thus, Lemma 12 holds in this case.
- Case $\kappa_{\psi,\varphi}(\sigma \cdot a) = \mathsf{false}$. Regarding the enforcement function, since $\kappa_{\psi,\varphi}(\sigma \cdot a)$ is false, according to the definition of $\mathrm{store}_{\psi,\varphi}$, we will have $\mathrm{store}_{\psi,\varphi}(\sigma \cdot a) = (\sigma_s, \sigma_c \cdot a)$. Consequently, $E_{\psi,\varphi}(\sigma \cdot a) = \sigma_s = E_{\psi,\varphi}(\sigma)$.
  Regarding the enforcement algorithm, upon receiving event a, after executing step 6 of the algorithm, $p = \delta_\psi(q_\psi, \sigma{\cdot}a)$, and $q = \delta_\varphi(q_\varphi, \sigma{\cdot}a)$. Since $\kappa_{\psi,\varphi}(\sigma{\cdot}a)$ is false, using Theorem 2, $\mathcal{L}(C,(p,q)) \neq \emptyset$. Consequently, statements 8, and 9 of the algorithm will not be executed, and statement 11 will be executed before going back to statement 3 (next iteration of the $while$ loop). Using the induction hypothesis, Lemma 12 holds for $\sigma \cdot a$ in this case.

A.2 Proofs: timed setting

In this section, we will discuss the proof of Theorem 3. φ and ψ in this section are regular timed properties that are defined by deterministic TA $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$.

Proof (of Theorem 3) We shall prove that given two properties ψ, and φ, the enforcement function $E_{\psi,\varphi}$ as per Definition 8 is a timed predictive enforcer satisfying constraints **(SndT)**, **(TrT)**, **(MoT)**, and **(UrT)**.

Let us recall the definition of the enforcement function.

$$E_{\psi,\varphi}(\sigma) = \mathit{\Pi}_1\left(\mathrm{store}_{\psi,\varphi}(\sigma)\right),$$

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

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

where

$$\kappa_{\psi,\varphi}(\sigma_n, \sigma_s, \sigma_c') \stackrel{\mathrm{def}}{=} \mathrm{CanD}(\sigma_c') \cap \mathrm{Sure}_{\psi,\varphi}(\sigma_n, \sigma_s)$$

and

$$\begin{aligned}
\mathrm{Sure}_{\psi,\varphi}(\sigma_n, \sigma_s) \stackrel{\mathrm{def}}{=} \{w \in tw(\Sigma) \mid \forall \sigma_{\mathrm{con}} \in \mathrm{tw}(\Sigma) :\\
\sigma_n \cdot \sigma_{\mathrm{con}} \in \psi \implies \exists \sigma' \in \mathrm{tw}(\Sigma) :\\
\sigma' \preccurlyeq_d \sigma_{\mathrm{con}} \wedge \sigma_s \cdot w \cdot \sigma' \in \varphi\}
\end{aligned}$$

The proof of constraint **(MoT)** follows the same reasoning of proof of Property 3 of Lemma 10 in the untimed setting.

Regarding constraints **(SndT)**, we prove that the enforcement function $E_{\psi,\varphi}$ satisfies the following condition:

$$\forall \sigma \in \mathrm{tw}(\sigma) : \sigma \in \psi \implies (E_{\psi,\varphi}(\sigma) = \epsilon \vee E_{\psi,\varphi}(\sigma) \models \varphi)$$

Regarding constraint **(TrT)**, we prove a slightly stronger property:

$$\forall \sigma \in \mathrm{tw}(\Sigma) : E_{\psi,\varphi}(\sigma) \preccurlyeq \sigma$$

Let us also recall the constraint **(UrT)**:

$$\begin{aligned}
&\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \geq \mathrm{end}(\sigma), \forall a \in \Sigma : \\
&(\exists w \in \mathrm{tw}(\Sigma) : E_{\psi,\varphi}(\sigma) \cdot w \geq_d \sigma \cdot (t,a) \ \wedge \ \mathrm{start}(w) \geq t \wedge \\
&\quad \forall \sigma_{\mathrm{con}} \in \mathrm{tw}(\Sigma) : \sigma \cdot (t,a) \cdot \sigma_{\mathrm{con}} \in \psi \implies \\
&\qquad \exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma' \preccurlyeq_d \sigma_{\mathrm{con}} \ \wedge \ E_{\psi,\varphi}(\sigma) \cdot w \cdot \sigma' \in \varphi) \\
&\implies \Pi_\Sigma(E_{\psi,\varphi}(\sigma \cdot (t,a))) = \Pi_\Sigma(\sigma \cdot (t,a)).
\end{aligned}$$

We also prove that for any timed word $\sigma \in \mathrm{tw}(\sigma)$, the function $\mathrm{store}_{\psi,\varphi}$ satisfies the following:

$$\Pi_\Sigma(\sigma_s \cdot \sigma_c) = \Pi_\Sigma(\sigma)$$

We shall prove it by an induction on the length of the input timed word σ.

Induction basis. If $\sigma = \epsilon$, from the definition of the enforcement function (Definition 8), $E_{\psi,\varphi}(\sigma) = \epsilon$. Since $E_{\psi,\varphi}(\sigma) = \sigma = \epsilon$, constraints **(SndT)** and **(UrT)** trivially holds. Since $\epsilon \preccurlyeq \epsilon$, $E_{\psi,\varphi}(\sigma) \preccurlyeq \sigma$ holds for $\sigma = \epsilon$, and thus constraint **(TrT)** holds. Since $\mathrm{store}_{\psi,\varphi}(\epsilon) = (\epsilon, \epsilon)$, for $\sigma = \epsilon$ the condition $\Pi_\Sigma(\sigma_s \cdot \sigma_c) = \Pi_\Sigma(\sigma)$ holds.

Induction step. Assume that for every $\sigma \in \mathrm{tw}(\sigma)$ of some length $n \in \mathbb{N}$, $E_{\psi,\varphi}$ satisfies constraints **(SndT)**, **(TrT)**, and **(UrT)**.

Let $\mathrm{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$. According to Definition 8, we know that $E_{\psi,\varphi}(\sigma) = \Pi_1\Big(\mathrm{store}_{\psi,\varphi}(\sigma)\Big) = \sigma_s$. Let $\Pi_\Sigma(\sigma_s \cdot \sigma_c) = \Pi_\Sigma(\sigma)$.

We now prove that for any $a \in \Sigma$, and $t \geq \mathrm{end}(\sigma)$, $E_{\psi,\varphi}$ satisfies constraints **(SndT)**, **(TrT)**, and **(UrT)** for $\sigma \cdot (t,a)$. We have the following two possible cases:

- Case $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a)) \neq \emptyset$.
  From the definition of $\mathrm{store}_{\psi,\varphi}$, we have $\mathrm{store}_{\psi,\varphi}(\sigma \cdot (t,a)) = (\sigma'_s, \epsilon)$ where $\sigma'_s = E_{\psi,\varphi}(\sigma \cdot (t,a)) = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} \kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a))$. We also know that $\Pi_\Sigma(\sigma_s \cdot \sigma_c) = \Pi_\Sigma(\sigma)$, and thus we have $\Pi_\Sigma(E_{\psi,\varphi}(\sigma \cdot (t,a))) = \Pi_\Sigma(\sigma'_s \cdot \epsilon) = \Pi_\Sigma(\sigma \cdot (t,a))$.
  Note that $\mathrm{CanD}(\sigma_c \cdot (t,a))$ computes all the delayed timed words of $\sigma_c \cdot (t,a)$ that start at or after t. Thus we have $E_{\psi,\varphi}(\sigma \cdot (t,a)) \preccurlyeq_d \sigma \cdot (t,a)$, and consequently we have $E_{\psi,\varphi}(\sigma \cdot (t,a)) \preccurlyeq_d \sigma \cdot (t,a)$. Thus **(TrT)** holds in this case.
  According to the definition of $\mathrm{Sure}_{\psi,\varphi}()$, the set $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a))$ which is $\mathrm{CanD}(\sigma_c \cdot (t,a)) \cap \mathrm{Sure}_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s)$ is non-empty if and only if the hypothesis of the **(UrT)** constraint is satisfied. We already showed that $\Pi_\Sigma(E_{\psi,\varphi}(\sigma \cdot (t,a))) = \Pi_\Sigma(\sigma \cdot (t,a))$ in this case. Thus the constraint **(UrT)** holds.
  Regarding constraint **(SndT)**, we have the following two cases based on whether $\sigma \cdot (t,a)$ belongs to ψ or not.
  - Case $\sigma \cdot (t,a) \notin \psi$. Constraint **(SndT)** trivially holds in this case.
  - Case $\sigma \cdot (t,a) \in \psi$. Since $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a))$ is non-empty, we know that $\mathrm{Sure}_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s)$ is non-empty. In this case, since $\sigma \cdot (t,a) \in \psi$, one possible continuation σ_{con} is ϵ, and thus all the timed words w that belong to the set $\mathrm{Sure}_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s)$ are the timed words satisfying the condition $\sigma_s \cdot w \in \varphi$. Since $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a))$ is $\mathrm{CanD}(\sigma_c \cdot (t,a)) \cap \mathrm{Sure}_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s)$, and $E_{\psi,\varphi}(\sigma \cdot (t,a)) = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}},\mathrm{end}} \kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a))$, we can conclude that $E_{\psi,\varphi}(\sigma \cdot (t,a)) \in \varphi$ satisfying constraint **(SndT)**.
- Case $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a)) = \emptyset$. From the definition of $\mathrm{store}_{\psi,\varphi}$, we have $\mathrm{store}_{\psi,\varphi}(\sigma \cdot (t,a)) = (\sigma_s, \sigma_c \cdot (t,a))$, and consequently we have $E_{\psi,\varphi}(\sigma \cdot (t,a)) = \sigma_s = E_{\psi,\varphi}(\sigma)$. Using the induction hypothesis, we can conclude that the constraints **(SndT)** and **(TrT)** also hold for $\sigma \cdot (t,a)$. Also $\Pi_\Sigma(\sigma_s \cdot \sigma_c \cdot (t,a)) = \Pi_\Sigma(\sigma \cdot (t,a))$ holds. Regarding constraint **(UrT)**, from the definition of $\kappa_{\psi,\varphi}()$ and $\mathrm{Sure}_{\psi,\varphi}()$ since $\kappa_{\psi,\varphi}(\sigma \cdot (t,a), \sigma_s, \sigma_c \cdot (t,a)) = \emptyset$, we can conclude that the hypothesis of the constraint **(UrT)** does not hold in this case. Thus, the constraint **(UrT)** trivially holds in this case.

B Implementation details

The predictive enforcement monitoring algorithm described in Section 3.5 is implemented in Python. Source files and examples can be downloaded from:

https://github.com/SrinivasPinisetty/PredictiveRE.

Module `Automata.py` contains all the functionality related to defining automata and operations on automata. Module `Enforcer.py` contains an implementation of the predictive enforcement algorithm described in Section 3.5. Using these modules is simple, and only requires Python. Let us now see an example illustrating how to describe the automata ψ, φ, and how to invoke the `enforcer` method.

In addition to the Python `system` module, import both the modules `Automata` and `Enforcer`.

```
import sys
sys.path.append("../")
import copy
import Enforcer
import Automata
```

Describe automata ψ, and φ. In this example, ψ is the automaton in Figure 4, and φ is the automaton in Figure 3.

```
psi = Automata.DFA(
['a', 'b', 'c' ,'?','!'],
['l0','l1', 'l2', 'l3', 'l4', 'l5'],
'l0',
lambda q: q in ['l4'],
lambda q, x: {
        ('l0', 'a') : 'l1',
        ('l0', 'b') : 'l1',
        ('l0', 'c') : 'l1',
        ('l0', '?') : 'l5',
        ('l0', '!') : 'l5',
        ('l1', 'a') : 'l2',
        ('l1', 'b') : 'l2',
        ('l1', 'c') : 'l2',
        ('l1', '?') : 'l5',
        ('l1', '!') : 'l5',
        ('l2', 'a') : 'l3',
        ('l2', 'b') : 'l3',
        ('l2', 'c') : 'l3',
        ('l2', '?') : 'l5',
        ('l2', '!') : 'l5',
        ('l3', 'a') : 'l5',
        ('l3', 'b') : 'l5',
        ('l3', 'c') : 'l5',
        ('l3', '?') : 'l5',
        ('l3', '!') : 'L4',
        ('l4', 'a') : 'l5',
        ('l4', 'b') : 'l5',
        ('l4', 'c') : 'l5',
        ('l4', '?') : 'l5',
        ('l4', '!') : 'l5',
        ('l5', 'a') : 'l5',
        ('l5', 'b') : 'l5',
        ('l5', 'c') : 'l5',
        ('l5', '?') : 'l5',
        ('l5', '!') : 'l5',
        }[(q, x)]
)

phi = Automata.DFA(
['a', 'b', 'c' ,'?','!'],
```

```
['l0','l1', 'l2', 'l3' ],
'l0',
lambda q: q in ['l3'],
lambda q, x: {
        ('l0', 'a') : 'l1',
        ('l0', 'b') : 'l1',
        ('l0', 'c') : 'l1',
        ('l0', '?') : 'l2',
        ('l0', '!') : 'l2',
        ('l1', 'a') : 'l1',
        ('l1', 'b') : 'l1',
        ('l1', 'c') : 'l1',
        ('l1', '?') : 'l3',
        ('l1', '!') : 'l3',
        ('l2', 'a') : 'l2',
        ('l2', 'b') : 'l2',
        ('l2', 'c') : 'l2',
        ('l2', '?') : 'l2',
        ('l2', '!') : 'l2',
        ('l3', 'a') : 'l2',
        ('l3', 'b') : 'l2',
        ('l3', 'c') : 'l2',
        ('l3', '?') : 'l2',
        ('l3', '!') : 'l2',
        }[(q, x)]
)
```

The `enforcer` method with automata ψ, φ, and some test input sequence can be invoked as follows:

```
Enforcer.enforcer(psi,phi,
         ['a', 'a', 'b', '!' ])
```

We provide other examples in the directories `examplesUsage` and `examplesAdditional`. All the examples used for evaluation (see Section 3.7) are available in the directory `examplesAdditional`. To run the examples in the directories `examplesUsage` and `examplesAdditional` via command prompt, navigate to the directory, and invoke each script in the directory. For example if `test1.py` is under the `examplesUsage` directory, you can invoke it using the command `python test1.py`.