# Predictive Runtime Verification of Timed Properties

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

# Predictive Runtime Verification of Timed Properties

Srinivas Pinisetty[a,*], Thierry Jéron[b], Stavros Tripakis[c], Yliès Falcone[d], Hervé Marchand[b], Viorel Preoteasa[a]

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

**Abstract**

Runtime verification (RV) techniques are used to continuously check whether the (untrustworthy) output of a black-box system satisfies or violates a desired property. When we consider runtime verification of timed properties, physical time elapsing between actions influences the satisfiability of the property. This paper introduces *predictive* runtime verification of *timed* properties where the system is not entirely a black-box but something about its behaviour is known a priori. *A priori* knowledge about the behaviour of the system allows the verification monitor to foresee the satisfaction (or violation) of the monitored property. In addition to providing a *conclusive* verdict earlier, the verification monitor also provides additional information such as the minimum (maximum) time when the property can be violated (satisfied) in the future. The feasibility of the proposed approach is demonstrated by a prototype implementation, which is able to synthesize predictive runtime verification monitors from timed automata.

*Keywords:* monitoring, runtime verification, timed automata, real-time systems.

## 1. Introduction

Runtime verification (RV) [1, 2, 3] deals with checking whether a run of a system under scrutiny satisfies or violates a desired property $\varphi$. It is a lightweight verification technique complementing other verification techniques such as model checking and testing.

Runtime verification techniques such as [1, 2, 3] mainly focus on synthesizing a verification monitor automatically from some high-level specification of the property $\varphi$ (which the monitor should verify). A verification monitor does not influence or change the program execution. Such a monitor can be used to check the current execution of a system (online) or a stored execution of a system (offline). An execution of a system

*Corresponding author
*Email addresses:* srinu85.pinisetty@gmail.com (Srinivas Pinisetty), thierry.jeron@inria.fr (Thierry Jéron), stavros.tripakis@aalto.fi (Stavros Tripakis), ylies.falcone@imag.fr (Yliès Falcone), herve.marchand@inria.fr (Hervé Marchand), viorel.preoteasa@aalto.fi (Viorel Preoteasa)

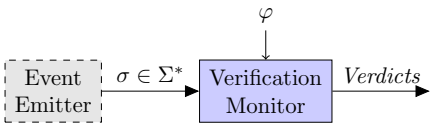



Figure 1: Runtime Verification.

is considered as a finite sequence of actions emitted by a system being monitored. As illustrated in Figure 1, a verification monitor takes a sequence of events $\sigma$ from an event emitter (system being monitored) as input and produces a verdict as output that provides information whether the current execution of the system $\sigma$ satisfies $\varphi$ or not. In runtime verification, one generally uses a specification formalism with a semantics customised for finite executions such as automata [4] or some variant of Linear Temporal Logic (LTL) such as $\text{LTL}_3$ [2] or RV-LTL [5].

*Predictive runtime verification.* In most of the existing runtime verification mechanisms such as in [2, 6], there is no assumption on the input sequence $\sigma$, which can be any sequence of events over some alphabet $\Sigma$. This can be seen as considering the event emitter (system being monitored) to be a *black-box*, i.e., its behavior is completely unknown. Recently in [7, 8], predictive semantics for runtime verification of *untimed* properties has been introduced where the program or the system being monitored is not a black-box, but assumes that some information is known (which can for instance be extracted using some kind of static-analysis on the program code) which helps the monitor to anticipate what the future of the current execution is. This knowledge of the system (which is an over-approximation) helps the monitor to foresee satisfaction or violation of the property being monitored even when the current observed execution of the system is, by itself, inconclusive.

*Runtime verification of timed properties.* When we consider properties with real-time constraints, the occurrence time of events influences the satisfaction of the property. Monitoring of properties with real-time constraints is a much difficult problem compared to the monitoring of untimed properties. Some existing works describe runtime verification mechanisms for timed properties [9, 10, 2, 6] (See Sec. 7 for related work). However, none of the works deal with predicting satisfaction or violation of the property using available knowledge/model of the system.

*Motivations.* Techniques such as RV are essential for safety-critical systems and in such systems we mostly encounter requirements with time constraints. Predictive runtime verification techniques allow the monitor to provide an earlier conclusive verdict whenever possible, even though the current observed execution might be inconclusive. Thus, it is certainly beneficial when we consider monitoring of timed properties. For instance, based on the current observation and the available knowledge of the system, if the monitor can foresee that the property $\varphi$ will certainly be violated in the future, in addition to providing the verdict earlier, the monitor can also provide additional information such as the minimum (maximum) time when the violation can happen in the future. Such additional information may be useful to take some corrective action before the actual violation occurs.

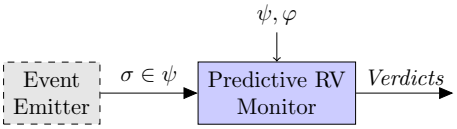



Figure 2: Predictive Runtime Verification.

Runtime monitoring mechanisms have been applied in different domains for instance in safety-critical embedded systems such as automotive systems [11] and robotic applications such as the NASA K9 Rover [9]. In the domain of network security, a monitor can be used as a firewall or a Network Intrusion Detection (NID), that monitors network traffic to detect (and prevent) some denial-of-service (DoS) attacks. Runtime monitoring mechanisms have been also used for security of mobile devices, for instance for monitoring security properties in Android applications [12, 13]. In the domain of web-services, a monitor can be used for checking contracts at runtime [14].

In the considered predictive RV framework, knowledge/abstract model of the system $\psi$ may be available (e.g., from design models, properties that are verified statically, and the properties that the system is known to enforce). If knowledge/abstract model of the system is not available, it can extracted from the system using techniques such as automata learning [15] and static-analysis [16].

*Context and objectives.* In this paper, we will focus on *predictive runtime verification* for properties with real-time constraints (timed properties). Executions of systems are considered as timed words (sequences of actions with their absolute occurrence dates). The timed property $\varphi$ to be monitored is formalised as a deterministic timed automaton (DTA) [17][1]. As illustrated in Figure 2, in addition to the property $\varphi$, we also consider providing available knowledge (over-approximation of the system or program) to the monitor denoted as $\psi$, which is also formalized as a timed automaton. Our *predictive timed RV monitor* takes a timed word $\sigma$ (current execution of the system) as input and produces a verdict as output. We focus on *online* verification monitoring, and whenever a new event is received, the monitor emits a verdict providing information on whether the current observed input sequence (followed by any possible extension of it according to $\psi$) satisfies or violates the property $\varphi$.

*Predictive timed RV problem.* Given a timed property to monitor $\varphi$ and another timed property $\psi$ describing the knowledge of the system being monitored, we want to synthesize a runtime verification monitor $M_{\psi,\varphi}$ that takes the current (partial) execution of the system $\sigma$ as input and provides information on whether the (complete) execution of the system (*currently or in the future according to* $\psi$) satisfies or violates the property $\varphi$. If the verdict is unknown according to the current observation $\sigma$, but is true (false) by anticipating the future of $\sigma$ according to $\psi$, then we also want the monitor to provide additional information such as the *minimum* (resp. maximum) time instant when the

[1] Alternatively, we could also specify timed properties using some timed logic such as TLTL.

current observation (according to $\psi$) can be extended to a sequence that satisfies (resp. violates) the property $\varphi$.

Our recent work on predictive runtime enforcement [18, 19] (see related work in Section 7) motivated us to work on the predictive runtime verification problem for timed properties that we present in this paper.

*Outline.* Section 2 introduces preliminaries and notation. First, we motivate predictive RV for timed properties via examples in Section 3. Then, we formally define and discuss the predictive RV problem for timed properties in Section 4. Later in Section 5, we provide a solution to the predictive RV problem described in Section 4, to synthesize a predictive RV monitor from timed properties $\varphi$ and $\psi$. We present in detail an *online* algorithm to solve the predictive RV problem for timed properties that takes timed properties $\varphi$ and $\psi$ as input. In Section 6, we discuss about an implementation of the predictive RV algorithm, and its performance evaluation using some example properties. Section 7 discusses related work, and in Section 8 we summarize results of this paper and mention about some ongoing (future) work.

## 2. Preliminaries

### 2.1. Untimed languages

A (finite) word over a finite alphabet $\Sigma$ is a finite sequence $w = a_1 \cdot a_2 \cdots a_n$ of elements of $\Sigma$. The *length* of $w$ is $n$ and is denoted as $|w|$. The empty word over $\Sigma$ is denoted by $\epsilon_\Sigma$, or $\epsilon$ when clear from the context. The set of all (respectively non-empty) words over $\Sigma$ is denoted by $\Sigma^*$ (respectively $\Sigma^+$). A *language* over $\Sigma$ is any subset $\mathcal{L}$ of $\Sigma^*$.

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

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

### 2.2. Timed words and timed languages

In a timed setting, the occurrence time of actions is also important. For a monitor in a timed setting, input streams 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 monitor.

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

In a timed setting, input and output streams of monitors are timed words. A timed word over the finite alphabet $\Sigma$ 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

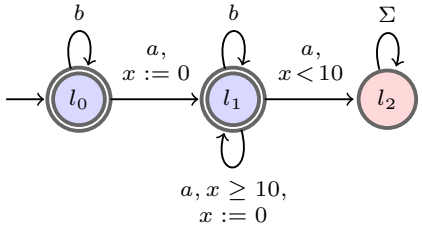



Figure 3: Timed Automaton: Example

by $\mathrm{start}(\sigma) \stackrel{\text{def}}{=} t_1$ the starting date of $\sigma$ and $\mathrm{end}(\sigma) \stackrel{\text{def}}{=} t_n$ its ending date (with the convention that the starting and ending dates are null for the empty timed word $\epsilon$).

The set of timed words over $\Sigma$ 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 $\sigma = (t_1, a_1) \cdots (t_n, a_n)$ and $\sigma' = (t'_1, a'_1) \cdots (t'_m, a'_m)$ be two timed words with $\mathrm{end}(\sigma) \leq \mathrm{start}(\sigma')$. Their concatenation is

$$\sigma \cdot \sigma' \stackrel{\text{def}}{=} (t_1, a_1) \cdots (t_n, a_n) \cdot (t'_1, a'_1) \cdots (t'_m, a'_m).$$

By convention $\sigma \cdot \epsilon \stackrel{\text{def}}{=} \epsilon \cdot \sigma \stackrel{\text{def}}{=} \sigma$. Concatenation is undefined otherwise.

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

### *2.3. Timed automata, operations on timed automata, and timed properties*

A timed automaton [17] 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 $\delta \in \mathbb{R}_{\geq 0}$, $\chi + \delta$ is the valuation assigning $\chi(x) + \delta$ to each clock $x$ of $X$. Given a set of clocks $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the clock valuation $\chi$ 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 $\chi$.

*Timed automata syntax and semantics.* Before going into the formal definitions, we introduce timed automata on an example. The timed automaton in Fig. 3 defines the requirement *"In every 10 time units, there cannot be more than 1 $a$ action"*. The set of locations is $L = \{l_0, l_1, l_2\}$, and $l_0$ is the initial location. The set of actions is $\Sigma = \{a, b\}$. There are transitions between locations upon actions. A finite set of real-valued clocks is used to model realtime behavior, the 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 $a$, the automaton moves from $l_0$ to $l_1$, and the clock $x$ is reset to 0. In location $l_1$, if action $a$ occurs, 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.

**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*, $\Sigma$ is a finite set of *actions*, $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*. $F \subseteq L$ is a set of *accepting locations*.

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 set of possible values for a clock is infinite, a timed automaton has an infinite number of 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 $\chi_0$ 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 the set of transitions of the form $(l, \chi) \xrightarrow{(\delta, a)} (l', \chi')$ with $\chi' = (\chi + \delta)[Y \leftarrow 0]$ whenever there exists $(l, g, a, Y, l') \in \Delta$ such that $\chi + \delta \models g$ for $\delta \in \mathbb{R}_{\geq 0}$.

A *run* of $\mathcal{A}$ from a state $q \in Q$ to a state $q' \in Q$ over a timed trace $w = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ is a sequence of transitions $q_0 \xrightarrow{(\delta_1, a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n, a_n)} q_n$, where $q = q_0$, $q' = q_n$, $t_1 = \delta_1$, and $\forall i \in [2, n] : t_i = t_{i-1} + \delta_i$. We denote by $q \xrightarrow{w} q'$ the fact that there exists a run from $q$ to $q'$ over $w$.

For $q \in Q$ and $K \subseteq L$, the *language of $\mathcal{A}$, starting in $q$ and ending in $K$*, denoted $\mathcal{L}(\mathcal{A}, q, K)$ is defined by

$$\mathcal{L}(\mathcal{A}, q, K) = \{w \mid \exists q' \in K \times \mathbb{R}^X_{\geq 0} : q \xrightarrow{w} q'\}.$$

The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$ is the language of $\mathcal{A}$ starting from the initial state $q_0$ and ending in a final state in $F$, $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, q_0, F)$.

**Definition 3 (Deterministic (complete) timed automata).** A timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ with its semantics $[\![\mathcal{A}]\!]$ is said to be a *deterministic* timed automaton (DTA) whenever for any location $l$ and any two distinct transitions $(l, g_1, a, Y_1, l'_1) \in \Delta$ and $(l, g_2, a, Y_2, l'_2) \in \Delta$ with same source $l$, 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 labelled by $a$ evaluates to *true* (i.e., it holds according to any valuation): $\forall l \in L, \forall a \in \Sigma : \bigvee_{(l,g,a,Y,l')} g = true$.

Note that, timed automata are not generally closed under complement, but deterministic timed automata are closed under all Boolean operations [17]. Also, not all nondeterministic timed automata are determinizable [20, 21, 22]. In the remainder of this paper, we use complete DTA's.

For a deterministic and complete automaton $\mathcal{A}$, the transition relation is a function i.e., $\forall q \in Q, \delta \in \mathbb{R}_{\geq 0}, a \in \Sigma : \exists q' \in Q : q \xrightarrow{\delta,a} q'$. We define the function $\mathsf{move}(\mathcal{A}, q, (\delta, a)) \stackrel{\text{def}}{=} q'$, the unique element $q' \in Q$ such that $q \xrightarrow{\delta,a} q'$, and we extend it to timed words by $\mathsf{move}(\mathcal{A}, q, w) \stackrel{\text{def}}{=} q'$, where $q'$ is the unique element such that $q \xrightarrow{w} q'$.

For an automaton $\mathcal{A}$, and $q \in Q$, the minimum time to reach a final state starting from $q$, denoted by $\mathsf{time2final}(\mathcal{A}, q)$, is defined by

$$\mathsf{time2final}(\mathcal{A}, q) = \mathsf{min}\{t \mid \exists w \in \mathcal{L}(\mathcal{A}, q, F) : t = \mathrm{end}(w)\}.$$

*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 DTA. That is, we consider the set of regular timed properties that can be defined as DTA. Given a timed word $\sigma \in \mathrm{tw}(\Sigma)$, we say that $\sigma$ satisfies $\varphi$ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$.

*Operations on timed automata.* We now introduce operations such as the product, and complement for DTA's. The product of timed automata is useful to intersect languages recognized by timed automata.

**Definition 4 (Product of TAs).** Given two TAs $\mathcal{A}_1 = (L_1, l_1^0, X_1, \Sigma, \Delta_1, F_1)$ and $\mathcal{A}_2 = (L_2, l_2^0, X_2, \Sigma, \Delta_2, F_2)$ with disjoint sets of clocks, their product is the TA $\mathcal{A}_1 \times \mathcal{A}_2 \stackrel{\text{def}}{=} (L, l_0, X, \Sigma, \Delta, F)$ where $L = L_1 \times L_2$, $l_0 = (l_0^1, l_0^2)$, $X = X_1 \cup X_2$ , $F = F_1 \times F_2$, and $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the *transition relation*, with $((l_1, l_2), g_1 \wedge g_2, a, Y_1 \cup Y_2, (l_1', l_2')) \in \Delta$ if $(l_1, g_1, a, Y_1, l_1') \in \Delta_1$ and $(l_2, g_2, a, Y_2, l_2') \in \Delta_2$.

Given two timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ the TA $\mathcal{A}$ obtained by computing their product according to Definition 4 recognizes the language $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$.

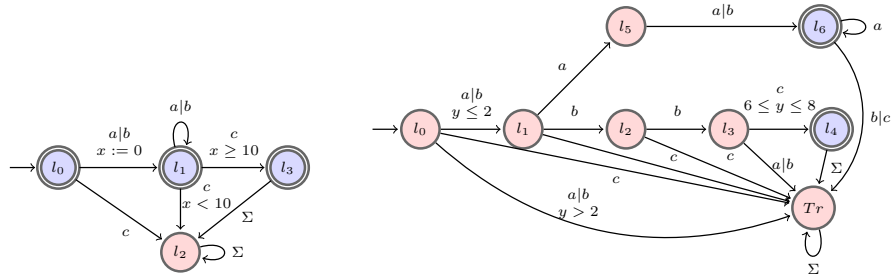
**Definition 5 (Complement of a complete DTA).** Given a property $\varphi$ defined by a complete DTA $\mathcal{A}_\varphi = (L, \ell^0, X, \Sigma, \Delta, F)$, its complement denoted as $\overline{\mathcal{A}_\varphi}$ is defined as $\overline{\mathcal{A}_\varphi} \stackrel{\text{def}}{=} (L, \ell^0, X, \Sigma, \Delta, L \setminus F)$.

Note that $\mathcal{L}(\overline{\mathcal{A}_\varphi}) = \mathrm{tw}(\Sigma) \setminus \mathcal{L}(\mathcal{A}_\varphi)$.

### *2.4. RV monitor in the non-predictive case*

Let us see the definition of the verification monitor for a given timed property $\varphi$, in the non-predictive case where we have no knowledge about the system.

**Definition 6 (RV monitor (non-predictive)).** Consider a timed property $\varphi \subseteq \mathrm{tw}(\Sigma)$ (property to monitor) defined as DTA $\mathcal{A}_\varphi = (L_\varphi, l_{0_\varphi}, X_\varphi, \Sigma, \Delta_\varphi, F_\varphi)$. A verification monitor for $\varphi$ is a function $M_\varphi : \mathrm{tw}(\Sigma) \to \mathcal{D}$, where $\mathcal{D} = \{\mathsf{true}, \mathsf{false}, \mathsf{c\_true}, \mathsf{c\_false}\}$ and is defined as follows:




(a) DTA Defining Monitored Property: $\varphi$.

(b) DTA Defining Property of the System: $\psi$.

Figure 4: DTA Defining Properties $\varphi$ and $\psi$

Let $\sigma \in \mathrm{tw}(\Sigma)$ denote a finite timed word over the alphabet $\Sigma$ (current observation).

$$M_\varphi(\sigma) = \begin{cases} \mathsf{true} & \text{if } \forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \varphi \\ \mathsf{false} & \text{if } \forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \notin \varphi \\ \mathsf{c\_true} & \text{if } \sigma \in \varphi \wedge \exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \notin \varphi \\ \mathsf{c\_false} & \text{if } \sigma \notin \varphi \wedge \exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \varphi \end{cases}$$

Property $\varphi$ is a language of finite timed words ($\varphi \subseteq \mathrm{tw}(\Sigma)$). Verdicts *true* (true) and *false* (false) are conclusive verdicts, and verdicts *currently true* (c_true), and *currently false* (c_false) are inconclusive verdicts[2].

- $M_\varphi(\sigma)$ returns true if for any continuation $\sigma' \in \mathrm{tw}(\Sigma)$, $\sigma \cdot \sigma'$ *satisfies* $\varphi$.
- $M_\varphi(\sigma)$ returns false if for any continuation $\sigma' \in \mathrm{tw}(\Sigma)$, $\sigma \cdot \sigma'$ *falsifies* $\varphi$.
- $M_\varphi(\sigma)$ returns c_true if $\sigma$ satisfies $\varphi$, and not all continuations of $\sigma$ satisfy $\varphi$ (there is a continuation $\sigma' \in \mathrm{tw}(\Sigma)$ such that $\sigma \cdot \sigma'$ does not satisfy $\varphi$).
- $M_\varphi(\sigma)$ returns c_false if $\sigma$ falsifies $\varphi$, and not all continuations of $\sigma$ falsify $\varphi$ (there is a continuation $\sigma' \in \mathrm{tw}(\Sigma)$ such that $\sigma \cdot \sigma'$ satisfies $\varphi$).

**Remark 1.** Linear temporal logics such as $LTL_3$ [23] and $LTL_4$ [23] are introduced specifically for runtime verification. Similar to $LTL_4$ (four-valued semantics for LTL), $\mathcal{D} = \{\mathsf{true}, \mathsf{false}, \mathsf{c\_true}, \mathsf{c\_false}\}$ in Definition 6.

## 3. Motivating examples

In this section, we consider some timed properties, some finite executions and compare the verdicts given by a non-predictive runtime verification monitor and a predictive one.

[2] An inconclusive verdict states an evaluation about the execution seen so far.

| $\sigma$ | $M_\varphi(\sigma)$ | $M_{\varphi,\psi}(\sigma)$ |
|---|---|---|
| $(1, a)$ | c_true | ? |
| $(1, a) \cdot (1.9, b)$ | c_true | false |
| $(1, a) \cdot (1.9, b) \cdot (3.2, b)$ | c_true | false |
| $(1, a) \cdot (1.9, b) \cdot (3.2, b) \cdot (6.2, c)$ | false | false |

Table 1: Non-predictive Vs. Predictive.

We denote the current observation using $\sigma$, the property being monitored as $\varphi$ and the property describing the input as $\psi$ (knowledge about the system). A non-predictive monitor for property $\varphi$ is denoted as $M_\varphi$, and a predictive monitor is denoted as $M_{\psi,\varphi}$. Via these examples, we illustrate the usefulness of prediction using $\psi$ to provide a conclusive verdict earlier whenever possible.

The verdicts provided by the predictive monitor belong to set $\mathcal{D} = \{\mathsf{true}, \mathsf{false}, \mathsf{c\_true}, \mathsf{c\_false}, ?\}$. Verdicts *true* (true), and *false* (false) are conclusive verdicts, and verdicts *currently true* (c_true), *currently false* (c_false) and *unknown* (?) are inconclusive verdicts.

Note that in our setting, we consider the runs of the underlying system to be finite. Properties $\varphi$ and $\psi$ are languages of finite timed words ($\varphi, \psi \subseteq \mathrm{tw}(\Sigma)$). In the predictive case (when $\psi \subseteq \mathrm{tw}(\Sigma)$), the current observation $\sigma$ is possibly a complete execution if $\sigma \in \psi$ and is certainly not a complete execution if $\sigma \notin \psi$. If the monitor cannot provide a conclusive verdict upon observing $\sigma$ and if $\sigma \in \psi$, the monitor provides a partial verdict (*currently true* (c_true) or *currently false* (c_false) depending on whether $\sigma \in \varphi$ or not. Inconclusive verdict *unknown* (?) is provided when the monitor cannot provide a conclusive verdict and the current observation cannot be a complete execution ($\sigma \notin \psi$). In the non-predictive case, the *unknown* (?) case can never occur since when we have no knowledge about the system, (i.e., when $\psi = \mathrm{tw}(\Sigma)$), the current observation $\sigma$ always belongs to $\psi$.

### *3.1. Providing a* violation *verdict earlier using prediction*

Consider the DTA's in Figure 4 defining properties $\psi$, and $\varphi$. Table 1 illustrates how a non-predictive and a predictive monitor behave when the input sequence $\sigma = (1, a) \cdot (1.9, b) \cdot (3.2, b) \cdot (6.2, c)$ is processed incrementally.

*Non-predictive case.* Let us first consider the non-predictive case. At $t = 1$, when the current observation is $\sigma = (1, a)$, $\sigma$ satisfies the property $\varphi$ but there are some extensions $\sigma' \in \mathrm{tw}(\Sigma)$ such that $\sigma \cdot \sigma'$ falsify the property $\varphi$. So the verdict provided by the monitor is c_true in the first step. Similarly, the verdict provided by the monitor is c_true in the next two steps at $t = 1.9$ and at $t = 3.2$. At $t = 6.2$, after observing the event $(6.2, c)$ (i.e., when the current observation is $\sigma = (1, a) \cdot (1.9, b) \cdot (3.2, b) \cdot (6.2, c)$), the property $\varphi$ is falsified by $\sigma$ and for any continuation $\sigma' \in \mathrm{tw}(\Sigma)$, $\sigma \cdot \sigma'$ falsifies the property $\varphi$. Thus, the non-predictive monitor provides a conclusive verdict (false) immediately after observing $(6.2, c)$ and it does not need to wait to observe more events to provide a conclusive verdict.

*Predictive case.* In the predictive case, since we have $\psi$ describing the knowledge of the system, at each step we only consider possible continuations of the current observation $\sigma$ that are allowed by $\psi$. At $t = 1$, when the current observation is $\sigma = (1, a)$, $\sigma$ satisfies the property $\varphi$ but there are some extensions $\sigma'$ according to $\psi$ such that $\sigma \cdot \sigma'$ falsify the property $\varphi$. Since $\sigma \notin \psi$, $\sigma$ cannot be a complete input sequence according to $\psi$, and the verdict will be unknown (?) at $t = 1$.

But at $t = 1.9$, when the current observation is $\sigma = (1, a) \cdot (1.9, b)$, using knowledge of the system $\psi$, a predictive monitor anticipates that the property $\varphi$ will be certainly violated in the future.

We can notice that a predictive monitor gives a conclusive verdict at time $t = 1.9$ when the second action $b$ is observed. Upon the current observation $\sigma = (1, a) \cdot (1.9, b)$, the only possible legal extensions of it according to $\psi$ are a $b$ action followed by a $c$ action (where the $c$ action happens before 8 time units). Thus a predictive monitor in this example can provide a conclusive verdict earlier at $t = 1.9$ because all continuations violate $\varphi$, whereas a non-predictive monitor has to wait longer and can give a conclusive verdict only at time $t = 6.2$.

Moreover, using $\psi$, in addition to providing a conclusive verdict earlier, a predictive-monitor can also give information indicating when the property will be violated in the future (e.g., minimum duration from the current time instant for observing a sequence that leads to a violation). In this particular example, at $t = 1.9$ when the current observation is $\sigma = (1, a) \cdot (1.9, b)$, from $\psi$ we also know that this sequence will be extended to a violating sequence between 6 and 8 time units.

### *3.2. Providing a* satisfaction *verdict earlier using prediction*

Consider the DTA in Figure 4 defining properties $\psi$, and $\varphi$. Table 2 illustrates how a predictive and a non-predictive monitor behave when the input sequence $\sigma = (1, a) \cdot (2.9, a) \cdot (3.2, b)$ is processed incrementally.

*Non-predictive case.* Let us first consider the non-predictive case. At $t = 1$, when the current observation $\sigma = (1, a)$, $\sigma$ satisfies the property $\varphi$ but there are some extensions $\sigma' \in \mathrm{tw}(\Sigma)$ such that $\sigma \cdot \sigma'$ falsify the property $\varphi$. So the verdict provided by the monitor is `c_true` in the first step. At $t = 2.9$, and $t = 3.2$, the verdict provided by the non-predictive monitor will still be `c_true` since there are some extensions of the observed input that falsify the property $\varphi$. Thus, the non-predictive monitor cannot provide any conclusive verdict upon processing the input sequence $\sigma = (1, a) \cdot (2.9, a) \cdot (3.2, b)$.

*Predictive case.* Let us now consider that the automaton $\psi$ in Figure 4b defines the property of the system (knowledge about the system being monitored). In the predictive case, at $t = 1$, when the current observation is $\sigma = (1, a)$, the verdict provided by the predictive monitor will be inconclusive, since $\sigma \notin \psi$, and according to $\psi$ there are some continuations of the current observation that falsify the property $\varphi$. At $t = 2.9$, when the current observation is $(1, a) \cdot (2.9, a)$, the predictive monitor provides a conclusive verdict `true` since all possible continuations of $(1, a) \cdot (2.9, a)$ according to $\psi$ satisfy the property $\varphi$.

| $\sigma$ | $M_{\varphi}(\sigma)$ | $M_{\varphi,\psi}(\sigma)$ |
|---|---|---|
| $(1, a)$ | c_true | ? |
| $((1, a) \cdot (2.9, a)$ | c_true | true |
| $(1, a) \cdot (2.9, a) \cdot (3.2, b)$ | c_true | true |

Table 2: Non-predictive Vs. Predictive.

We can notice that a predictive monitor gives a conclusive verdict true at time $t = 2.9$ when the second action $a$ is observed. Also notice that the non-predictive monitor cannot give a conclusive verdict upon observing $\sigma$. It is thus possible to turn off the monitor at $t = 2.9$ in the predictive case. Whenever it is possible to provide a conclusive verdict true earlier, the monitor can be turned off immediately.

## 4. Problem definition

A predictive verification monitor is a device that reads a finite trace and yields a certain verdict. We focus on *online* monitoring where the monitor considers executions in an incremental fashion. Whenever a new event is observed (changing the current observed input sequence), the monitor emits a verdict. A verdict is a truth value from some truth domain. The truth domain we consider is $\mathcal{D} = \{\mathsf{true}, \mathsf{false}, \mathsf{c\_true}, \mathsf{c\_false}, ?\}$, where verdicts *currently true* (c_true), *currently false* (c_false) and *unknown* (?) are inconclusive verdicts.

In our predictive setting, in addition to the property $\varphi$ that the monitor checks for satisfaction (violation), the monitor is also provided with another additional property $\psi$ describing the input sequences (knowledge of the system behavior). Our predictive monitor does not evaluate an observation to true (resp. false) if there still exists a continuation of the observation according to $\psi$ that violates (resp. satisfies) property $\varphi$.

Let us also recall that we consider the runs of the underlying system to be finite. Properties $\varphi$ and $\psi$ are languages of finite time words ($\varphi, \psi \subseteq \mathrm{tw}(\Sigma)$). As explained via examples in section 3, when we consider a finite execution (current observation $\sigma$), if $\sigma \in \psi$, it might be the complete execution. If the monitor cannot provide a conclusive verdict upon observing $\sigma$ and if $\sigma \in \psi$, the monitor provides a partial verdict (*currently true* (c_true) or *currently false* (c_false)) depending on whether $\sigma \in \varphi$ or not. The monitor provides the *unknown* (?) verdict if a conclusive verdict cannot be provided and $\sigma \notin \psi$ (there is necessarily a continuation according to $\psi$).

Using knowledge of $\psi$, a predictive monitor provides a conclusive verdict earlier whenever possible by anticipating all the possible extensions of the current observation according to $\psi$. Though the current observation $\sigma$ has some continuations leading to different verdicts, when every extension of $\sigma$ according to $\psi$ leads to the same conclusive verdict true (resp. false), then the monitor provides the conclusive verdict upon observing $\sigma$ itself.

The monitor produces for each consumed event a verdict indicating the status of the property depending on the event sequence seen so far. In this section, we describe the predictive RV monitor as a function.

**Definition 7 (Predictive RV monitor).** Consider two timed properties $\varphi \subseteq \mathrm{tw}(\Sigma)$ (property to monitor) and $\psi \subseteq \mathrm{tw}(\Sigma)$ (property of the system). Let $\sigma \in \mathrm{tw}(\Sigma)$ denote a finite timed word over the alphabet $\Sigma$ (current observation). A predictive verification monitor for $\psi$, $\varphi$, is a function $M_{\psi,\varphi} : \mathrm{tw}(\Sigma) \to \mathcal{D} \times \mathbb{R}_{\geq 0}$ and is defined as follows:

$$M_{\psi,\varphi}(\sigma) = \begin{cases} (\mathsf{true}, \mathsf{minT}(\sigma,\psi)) & \text{if } \forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \Rightarrow \sigma \cdot \sigma' \in \varphi \\ (\mathsf{false}, \mathsf{minT}(\sigma,\psi)) & \text{if } \forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \Rightarrow \sigma \cdot \sigma' \notin \varphi \\ (\mathsf{c\_true}, 0) & \text{if } \sigma \in \psi \wedge \sigma \in \varphi \wedge (\exists \sigma' \in \mathrm{tw}(\Sigma) : \\ & \qquad \sigma \cdot \sigma' \in \psi \wedge \sigma \cdot \sigma' \notin \varphi) \\ (\mathsf{c\_false}, 0) & \text{if } \sigma \in \psi \wedge \sigma \notin \varphi \wedge (\exists \sigma' \in \mathrm{tw}(\Sigma) : \\ & \qquad \sigma \cdot \sigma' \in \psi \wedge \sigma \cdot \sigma' \in \varphi) \\ (?, 0) & \text{otherwise} \end{cases}$$

where $\mathsf{minT}(\sigma,\psi)$ is the minimum of the end dates of all possible extensions of $\sigma$ that satisfy $\psi$:

$$\mathsf{minT}(\sigma,\psi) = \mathsf{min}\ \{t \mid \exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \wedge t = \mathrm{end}(\sigma \cdot \sigma')\}$$

The predictive verification monitor $M_{\psi,\varphi}$ takes a timed word $\sigma$ as input and returns a tuple. The first element of the output is an element of the set $\mathcal{D}$, that indicates whether the current run (based on the current observation $\sigma$ and the property of the system $\psi$) satisfies or violates the property $\varphi$. The second element (a real value) is used when the monitor provides a conclusive verdict (either true or false), and it is null in the other cases[3].

Thus, when the monitor provides a conclusive verdict earlier, this minimum time information can be useful to take some action whenever necessary and possible.

- $M_{\psi,\varphi}(\sigma)$ returns true if every continuation of the current observation $\sigma$ to a sequence satisfying $\psi$ also satisfies $\varphi$. In this case, using the function $\mathsf{minT}(\sigma,\psi)$, the monitor also computes the minimum absolute time when the observation $\sigma$ can be extended to a sequence satisfying $\psi$ (say $\sigma \cdot \sigma'$), such that any extension of $\sigma \cdot \sigma'$ satisfying $\psi$ also satisfies $\varphi$. The minimum time is computed using the function $\mathsf{minT}(\sigma,\psi)$, that returns the minimum ending date of an extension $\sigma'$ of $\sigma$ that satisfies $\psi$. Because of the condition of this case ($\forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \Rightarrow \sigma \cdot \sigma' \in \varphi$), this is also the minimum time such that $\sigma \cdot \sigma' \in \psi \cap \varphi$.
- $M_{\psi,\varphi}(\sigma)$ returns false if every continuation of the current observation $\sigma$ to a sequence satisfying $\psi$ violates $\varphi$. In this case, the monitor computes the minimum absolute time when the observation $\sigma$ can be extended to a sequence satisfy-

[3] In the sequel, $M_{\psi,\varphi}(\sigma)$ is the first component of the tuple by default.

ing $\psi$ (say $\sigma \cdot \sigma'$). The minimum time is computed using the same function $\mathsf{minT}(\sigma, \psi)$, as in the first case, but now because of the condition of this case ($\forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \Rightarrow \sigma \cdot \sigma' \notin \varphi$), this is also the minimum time such that $\sigma \cdot \sigma' \in \psi \setminus \varphi$). Moreover all extensions of $\sigma \cdot \sigma'$ that satisfy $\psi$ violate $\varphi$.

- $M_{\psi,\varphi}(\sigma)$ returns *currently true* (c_true) when the current observation $\sigma$ satisfies the property of the system $\psi$ and the requested property $\varphi$, but there exists an extension $\sigma'$ of $\sigma$ such that $\sigma \cdot \sigma'$ satisfies $\psi$ but violates $\varphi$. The observation is thus a possibly complete sequence of the system (it may stop there) that is correct, but some (complete) continuation may modify this.

- $M_{\psi,\varphi}(\sigma)$ returns *currently false* (c_false) when the current observation $\sigma$ satisfies the property of the system $\psi$ and violates the property $\varphi$, and there is an extension $\sigma'$ of $\sigma$ such that $\sigma \cdot \sigma'$ satisfies $\psi$ and satisfies $\varphi$. The observation is thus a possibly complete sequence of the system that is incorrect, but some continuation may modify this.

- The last case is the default case, where $M_{\psi,\varphi}(\sigma)$ returns the *unknown* (?) verdict when none of the above cases do not hold. This happens when the current observation $\sigma$ is incomplete (i.e., $\sigma \notin \psi$) and there exist some continuations of $\sigma$ (according to $\psi$) satisfying $\varphi$, and other ones violating $\varphi$.

Theorems 1 and 2 state that conclusive verdicts are preserved by extensions of the current observations and thus are definitive.

**Theorem 1.** $\forall \sigma \in \mathrm{tw}(\Sigma) : (M_{\psi,\varphi}(\sigma) = \mathsf{true}) \Rightarrow (\forall \sigma_c \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma_c \in \psi \Rightarrow \forall \sigma' \preccurlyeq \sigma_c : M_{\psi,\varphi}(\sigma \cdot \sigma') = \mathsf{true})$.

Theorem 1 states that for any current observation $\sigma \in \mathrm{tw}(\Sigma)$, if the verdict provided by $M_{\psi,\varphi}(\sigma)$ is true, then for any $\sigma'$ which is a prefix of an extension of $\sigma$ in $\psi$, the verdict provided by the monitor will be true.

**Theorem 2.** $\forall \sigma \in \mathrm{tw}(\Sigma) : (M_{\psi,\varphi}(\sigma) = \mathsf{false}) \Rightarrow (\forall \sigma_c \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma_c \in \psi \Rightarrow \forall \sigma' \preccurlyeq \sigma_c : M_{\psi,\varphi}(\sigma \cdot \sigma') = \mathsf{false})$.

Theorem 2 is the dual of theorem 1 for the conclusive verdict false. Theorem 2 states that if for any current observation $\sigma \in \mathrm{tw}(\Sigma)$, if the verdict provided by $M_{\psi,\varphi}(\sigma)$ is false, then for any $\sigma'$ that is a prefix of extension of $\sigma$ in $\psi$, the verdict provided by the monitor will be false.

The following theorem states that the verdict provided by the monitor is inconclusive (i.e., if it is either c_true, c_false or ?), if and only if some continuation of $\sigma$ (according to $\psi$) satisfies $\varphi$, and some continuation of $\sigma$ (according to $\psi$) violates $\varphi$. In some sense it means that the monitor provides a conclusive verdict as soon as possible. In fact it is proved by simply negating the disjunction of the two conditions for conclusive verdicts.

**Theorem 3.** $\forall \sigma \in \mathrm{tw}(\Sigma) : M_{\psi,\varphi}(\sigma) \in \{\mathsf{c\_true}, \mathsf{c\_false}, ?\} \iff (\exists \sigma_c \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma_c \in \psi \cap \varphi) \wedge (\exists \sigma'_c \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma'_c \in \psi \setminus \varphi)$.

**Remark 2.** Note that when the current observation $\sigma$ belongs to $\psi$ but there is no continuation of $\sigma$ according to $\psi$ (i.e., when we are at the end of the execution), then $M_{\psi,\varphi}(\sigma)$ provides a conclusive verdict.

Next lemma introduces how function $\mathsf{minT}$ is calculated using $\mathsf{time2final}$. It will be used in the algorithm implementing function $M_{\psi,\varphi}$.

**Lemma 1.** *If property $\psi$ is recognized by the DTA $\mathcal{A}_\psi$, then*

$$\mathsf{minT}(\sigma,\psi) = \mathrm{end}(\sigma) + \mathsf{time2final}(\mathcal{A}_\psi, \mathsf{move}(\mathcal{A}_\psi, q^0_\psi, \sigma)).$$

Recall that $\mathsf{time2final}(\mathcal{A}, q)$ (defined in Section 2.3) returns the minimum time needed to reach a final state starting from $q$ in automaton $\mathcal{A}$.

**Remark 3.** When $M_{\psi,\varphi}$ provides a conclusive verdict (first two cases of $M_{\psi,\varphi}$), the monitor also computes and provides the minimum time when the satisfaction (violation) can happen in the future according to $\psi$. Minimum time information can be useful to take some action (whenever necessary and possible) before a problem may arise. Note that the monitor could also compute the maximum time when the satisfaction (violation) can happen in the future according to $\psi$. Maximum time information could also be useful, since it indicates the maximum time before which a decision should be taken, before something bad happens in the best case. Both minimum time and maximum time (if bounded) values can be computed efficiently [24].

## 5. Predictive runtime verification algorithm

Let us now see in detail an *online* algorithm to solve the predictive RV problem described in Section 4. The algorithm requires properties $\varphi$ (i.e., property to check) and $\psi$ (i.e., property of the system describing the input) defined as deterministic timed automata. The predictive RV monitor takes a stream of timed events as input and provides a stream of verdicts as output. Whenever the monitor receives a new input event, it produces a verdict as output.

*Operations on timed automata.* The construction of a predictive RV monitor requires the use of operations on timed automata such as product (Definition 4) and complement (Definition 5). The product of two given timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ according to definition 4 recognizes the language $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$. Also, recall that for a given timed automaton $\mathcal{A}$ the language accepted by $\overline{\mathcal{A}}$ is $\mathrm{tw}(\Sigma) \setminus \mathcal{L}(\mathcal{A})$.

*Reachability and minimum-time reachability.* In the algorithm we need to check whether a certain set of states is reachable from the current state $q$ (reachability problem). Whenever the monitor can provide a conclusive verdict (either $\mathsf{true}$ or $\mathsf{false}$), we also need to provide the minimal time instant when a state in a certain set of states (e.g., a state in the set of accepting states) is reachable from the current state $q$ (minimum-time reachability problem). Both these problems are decidable and are PSPACE-complete in the size of $\mathcal{A}$ (see Theorems 4 and 5).

**Remark 4.** The region abstraction [17] is a partition of the infinite state-space of a timed automaton into a finite number of equivalent classes, which preserves reachability and other properties. Region abstraction is too fine-grained and typically does not scale well in practice. For this reason, timed automata model-checkers use the so-called *zone abstraction* [25, 26, 27, 28]. A zone is a set of valuations defined by a conjunction of clock constraints, which can be represented efficiently using Difference Bounded Matrices (DBMs) [29]. For a DTA $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$, we say that the size of $\mathcal{A}$ is $|X| + C + |L| + |\Delta|$ where $|X|$ is the number of clocks, $C$ is the maximal constant appearing in guards, $|L|$ is the number of locations, and $|\Delta|$ the number of edges. The size of the region/zone graph is $O((|\Delta| + |L|).(2C + 2)^{|X|}.|X|!.2^{|X|})$ and is thus exponential in the size of $\mathcal{A}$.

**Theorem 4 (Reachability).** *For a DTA $\mathcal{A} = (L, \ell^0, X, \Sigma, \Delta, F)$ checking whether $F$ is reachable is decidable and is PSPACE-complete in the size of $\mathcal{A}$ [17].*

The minimum-time reachability problem for a DTA $\mathcal{A} = (L, \ell^0, X, \Sigma, \Delta, F)$ consists in checking whether $F$ is reachable, and if yes finding a run with minimum duration.

**Theorem 5 (Minimum-time reachability).** *The minimum-time reachability problem for timed automata is also PSPACE-complete in the size of $\mathcal{A}$ [30, 31, 24].*

In the algorithm, we denote the function that computes minimum-time using time2final (see Section 2).

**Corollary 1 (Checking emptiness).** *For a DTA $\mathcal{A} = (L, \ell^0, X, \Sigma, \Delta, F)$ with semantics $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \rightarrow, Q^F)$, consider a state $q \in Q$ and $K \subseteq L$. The language $\mathcal{L}(\mathcal{A}, q, K)$ starting from state $q$ and ending in $K$ is empty if $K$ is not reachable from $q$.*

Since the reachability problem is PSPACE-complete (see Theorem 4), checking emptiness is also PSPACE-complete.

### *5.1. Algorithm*

Let us now see the algorithm in detail. Let $\mathcal{A}_\psi = (L_\psi, l^0_\psi, X_\psi, \Sigma, \Delta_\psi, F_\psi)$ be the DTA defining property $\psi$ and $\mathcal{A}_\varphi = (L_\varphi, l^0_\varphi, X_\varphi, \Sigma, \Delta_\varphi, F_\varphi)$ be the DTA defining property $\varphi$. Let $q^0_\psi$ and $q^0_\varphi$ be the initial states of $[\![\mathcal{A}_\psi]\!]$ and $[\![\mathcal{A}_\varphi]\!]$, and let $Q^F_\psi$ and $Q^F_\varphi$ be the sets of final states of $[\![\mathcal{A}_\psi]\!]$ and $[\![\mathcal{A}_\varphi]\!]$.

Algorithm Predictive Monitor (see Algorithm 1) is an *repeat until* loop that starts from the initial states of the two automata. In the iteration, the algorithm checks the verdict for the current states. If the verdict is conclusive (true or false), then the loop terminates. Otherwise, upon receiving an input event $(t, a)$, it updates the current states (according to the event), and it does a new iteration. The event consists of the absolute time $t$ and the letter $a \in \Sigma$.

The algorithm uses variable $t_0$ to keep track of the date of the last received event. Initially $t_0$ is equal to 0.

Variable $\mathcal{B}$ is initialized with the product automaton of $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ ($\mathcal{A}_\psi \times \mathcal{A}_\varphi$). We use $\mathcal{B}$ with two sets of final states $F = F_\psi \times F_\varphi$ and $F' = F_\psi \times (L_\psi \backslash F_\varphi)$.

Variables $p$ and $q$ are states in the semantics of $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$, and initially they are assigned the initial states of these automata. When we use the pair $(p, q)$ as a state in $[\![\mathcal{B}]\!]$, we mean actually the state corresponding to $(p, q) = ((l, \chi), (l', \chi'))$ in the product automaton:

$$((l, l'), \chi \oplus \chi')$$

where the valuation $\chi \oplus \chi'$ combines the valuations of the disjoint clocks of the two automata.

Primitive await_event is used to wait for a new input event, and primitive notify notifies the verdict (the result of the function $M_{\psi,\varphi}$) at every step.

Recall that function $\mathsf{move}(\mathcal{A}, r, (\delta, a))$ returns a new state $r'$ reached in $[\![\mathcal{A}]\!]$ from $r$ with $(\delta, a)$. Function $\mathsf{time2final}(\mathcal{A}, r)$ calculates the minimum time to reach a final state in $[\![\mathcal{A}]\!]$ from $r$. $\mathcal{L}(\mathcal{A}, r, K)$ is the language of $\mathcal{A}$ starting in semantic state $r$ and ending in semantic states with locations from $K$.

**Algorithm 1** Predictive Monitor

1: $t_0 \leftarrow 0$
2: $\mathcal{B} \leftarrow \mathcal{A}_\psi \times \mathcal{A}_\varphi$
3: $F, F' \leftarrow F_\psi \times F_\varphi, F_\psi \times (L_\varphi \backslash F_\varphi)$
4: $p, q \leftarrow q^0_\psi, q^0_\varphi$
5: **repeat**
6: **if** $\mathcal{L}(\mathcal{B}, (p, q), F') = \emptyset$ **then**
7: $\mathsf{notify}(\mathsf{true}, t_0 + \mathsf{time2final}(\mathcal{A}_\psi, p))$
8: exit
9: **else if** $\mathcal{L}(\mathcal{B}, (p, q), F) = \emptyset$ **then**
10: $\mathsf{notify}(\mathsf{false}, t_0 + \mathsf{time2final}(\mathcal{A}_\psi, p))$
11: exit
12: **else if** $p \in Q^F_\psi \wedge q \in Q^F_\varphi$ **then**
13: $\mathsf{notify}(\mathsf{c\_true}, 0)$
14: **else if** $p \in Q^F_\psi \wedge q \notin Q^F_\varphi$ **then**
15: $\mathsf{notify}(\mathsf{c\_false}, 0)$
16: **else**
17: $\mathsf{notify}(?, 0)$
18: $t, a \leftarrow \mathsf{await\_event}()$
19: $t_0, \delta \leftarrow t, t - t_0$
20: $p, q \leftarrow \mathsf{move}(\mathcal{A}_\psi, p, (\delta, a)), \mathsf{move}(\mathcal{A}_\varphi, q, (\delta, a))$
21: **until** false

At the beginning of each iteration step of the algorithm $p = \mathsf{move}(\mathcal{A}_\psi, q^0_\psi, \sigma)$, $q = \mathsf{move}(\mathcal{A}_\varphi, q^0_\varphi, \sigma)$, and $t_0 = \mathrm{end}(\sigma)$, where $\sigma$ is the sequence of the events received so far. The iteration of the algorithm starts with checking the verdict for current states

$p$ and $q$. There are five possible cases corresponding to the five cases of $M_{\psi,\varphi}$:

- *Case 1:* If the language of $\mathcal{B}$ starting in $(p,q)$ and ending in $F'$ is empty (i.e., $\mathcal{L}(\mathcal{B},(p,q),F') = \emptyset$), then a conclusive verdict true is returned as a result and the algorithm stops. Note that $\mathcal{L}(\mathcal{B},(p,q),F') = \mathcal{L}(\mathcal{A}_\psi,p,F_\psi) \cap (\mathrm{tw}(\Sigma) \setminus \mathcal{L}(\mathcal{A}_\varphi,q,F_\varphi))$. Therefore, $\mathcal{L}(\mathcal{B},(p,q),F') = \emptyset$ iff $\mathcal{L}(\mathcal{A}_\psi,p,F_\psi) \subseteq \mathcal{L}(\mathcal{A}_\varphi,q,F_\varphi)$. Since $p = \mathsf{move}(\mathcal{A}_\psi,q^0_\psi,\sigma)$ and $q = \mathsf{move}(\mathcal{A}_\varphi,q^0_\varphi,\sigma)$, this emptiness check is equivalent to the test on the current observation $\sigma$ in the first case of the definition of $M_{\psi,\varphi}$ (Definition 7). In addition, the algorithm also provides the minimum time information, computed using $t_0 + \mathsf{time2final}(\mathcal{A}_\psi,p)$, and this is exactly $\mathsf{minT}(\psi,\sigma)$ by Lemma 1 because $t_0 = \mathrm{end}(\sigma)$. If the condition $\mathcal{L}(\mathcal{B},(p,q),F') = \emptyset$ does not hold, then the algorithm proceeds with the next case.
- *Case 2:* This is similar to Case 1, but we use the set $F$ instead of $F'$, and the condition $\mathcal{L}(\mathcal{B},(p,q),F) = \emptyset$ is equivalent to the second condition of $M_{\psi,\varphi}$. If the condition of this case is also false, then we proceed to the next case.
- *Case 3:* The condition $(p \in Q^F_\psi \wedge q \in Q^F_\varphi)$ of this test is equivalent to $\sigma \in \psi \wedge \sigma \in \varphi$ that is part of the condition of the third case of $M_{\psi,\varphi}$. The third case of $M_{\psi,\varphi}$ contains also the condition $C := (\exists \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \wedge \sigma \cdot \sigma' \notin \varphi)$. However, $C$ is equivalent to the negation of $D := (\forall \sigma' \in \mathrm{tw}(\Sigma) : \sigma \cdot \sigma' \in \psi \Rightarrow \sigma \cdot \sigma' \in \varphi)$ and $D$ is equivalent to the first condition of the algorithm as we noticed already. Since we reach this case only when $D$ is false, the condition of this case is equivalent to the corresponding condition of $M_{\psi,\varphi}$.
- *Case 4:* This case is similar to Case 3.
- *Case 5:* If all the above cases do not hold, then the algorithm provides the inconclusive verdict *unknown* (?).

After Cases 3, 4 and 5, the algorithm waits for a new event, updates the current states $p$ and $q$, and also $t_0$, and starts a new iteration.

We thus showed that Algorithm 1 implements the predictive RV monitor in Definition 7.

**Remark 5 (Pre-computing symbolic states and verdicts).** In Algorithm 1, the product DTA $\mathcal{B}$ and the sets of locations $F$ and $F'$ are computed off-line. All reachable symbolic states in $\mathcal{B}$ (where a symbolic state is a pair consisting of a location and a zone) can also be computed off-line. Moreover, for each symbolic state, checking whether the set of locations $F$ (resp. $F'$) is reachable from it can also be computed off-line. Thus, the verdict corresponding to each symbolic state that is reachable from the initial state can be computed off-line, considerably improving the real-time performance of the monitor.

## 6. Implementation and evaluation

The predictive runtime verification algorithm for timed properties presented in Section 5 has been implemented in Python and is available for download at: https://github.com/SrinivasPinisetty/PredictiveRV-TimedProperties. The main goal of this prototype implementation is to validate the feasibility and practicality of the proposed framework, and to measure and analyse the performance of the proposed algorithm.

Our implementation uses publicly available libraries and tools for timed automata based on the widely used UPPAAL model-checker [27] [4]. Algorithm 1 is implemented in 700 lines of code in Python (excluding libraries). An overview of the implementation modules and how their functionality is divided is presented in Figure 5. Properties such as the property of the system $\psi$ and the property to be verified $\varphi$ are defined as DTA's, and represented in the UPPAAL .XML format.

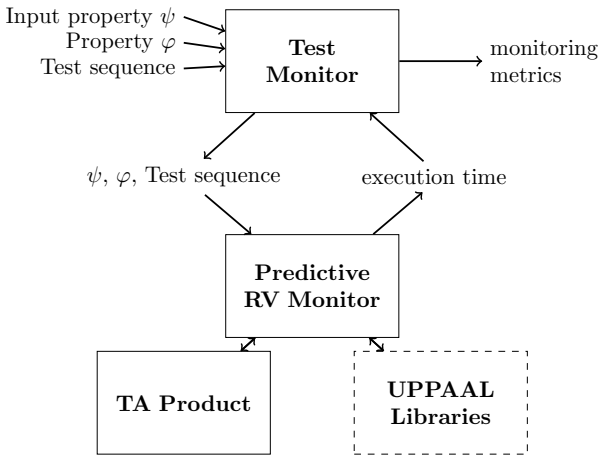



Figure 5: Implementation Overview.

The `main` test method in module `Test Monitor` is invoked with two DTA defining $\psi$ and $\varphi$, and a test sequence which is a timed word that belongs to the property of the system $\psi$. The `main` test method invokes the `predictiveRVmonitor` method in the `Predictive RV Monitor` module multiple times for the given inputs, computes average values over multiple executions and stores the performance results. The `Predictive RV Monitor` module contains other methods such as for checking reachability of a set of locations, and to move in a DTA from a given state by consuming an input event. In Algorithm 1, we compute the product of the automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ and locations $F$ and $F'$ in steps 2 and 3. The `TA Product` module contains functionality to compute the product of two DTAs, and the sets of locations $F$ and $F'$. The `Predictive RV Monitor` module also uses pyuppaal and UPPAAL DBM libraries.

*Example properties.* We evaluated the performance of the prototype implementation using a set of example properties $\varphi$ and $\psi_i$, for $i = 1, ..., 10$. In these properties, $\Sigma = \{a, b\}$, and the corresponding timed automata $\mathcal{A}_{\psi_i}$ and $\mathcal{A}_\varphi$ consist of one clock each. In all the examples, the property to be verified $\varphi$ is the same, and can be expressed in English as *"In every 300 time units, there are no more than 6 "b" actions."* The timed automaton in Figure 6 defines property $\varphi$. Property of the system $\psi_i$ in each

[4] Libraries such as pyuppaal for parsing/manipulating UPPAAL models are available at: `http://people.cs.aau.dk/~adavid/python/`.

example can be stated as *"There should be* `i` *"a" actions, which should be immediately followed by a "b" action in every* `k` *time units"*, where $k$ is a parameter equal to 30. Increasing `i` increases the number of locations and transitions in $\mathcal{A}_{\psi_i}$, and consequently in the product automaton $\mathcal{A}_{\psi_i} \times \mathcal{A}_{\varphi}$. All these example automata used for evaluation are provided (in UPPAAL .XML format) together with the implementation, and can be visualized using the UPPAAL tool.

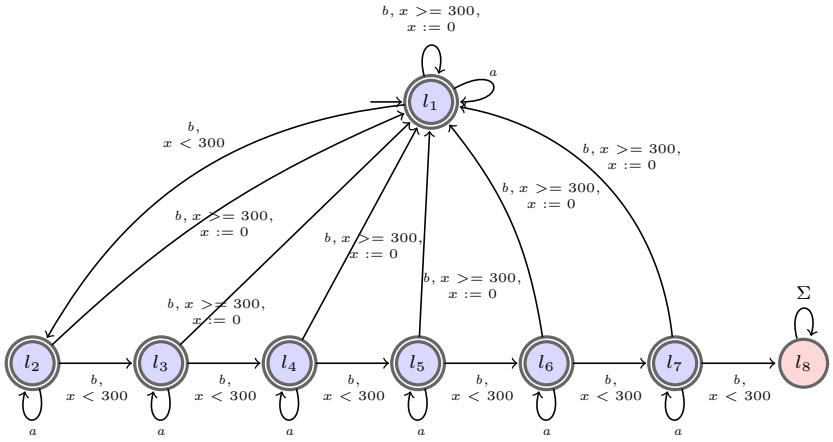



Figure 6: Automaton $\mathcal{A}_{\varphi}$.

*Evaluation.* In Algorithm 1, the product DTA $\mathcal{B}$ in line 2, and the sets of locations $F$ and $F'$ in line 3 are computed off-line. Moreover, all the reachable symbolic states in the DTA $\mathcal{B}$ (starting from the initial state), are computed off-line and for each symbolic state, reachability of locations in $F$ (resp. reachability of locations in $F'$) is also computed off-line before online monitoring starts (i.e., before entering the repeat loop, line 5) and the results are stored in a table.

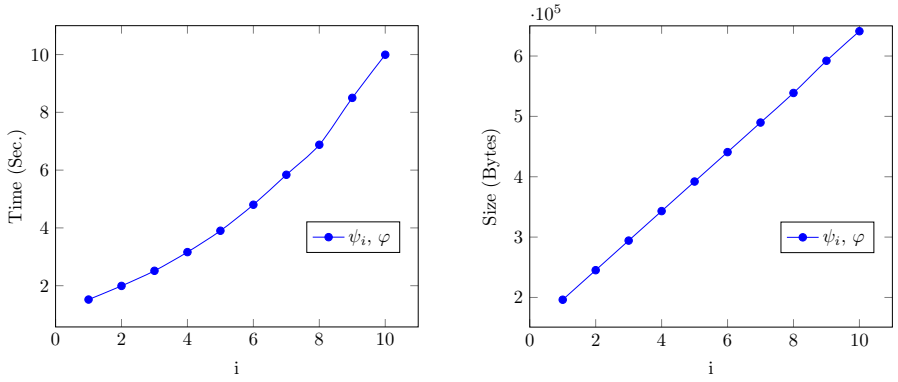
We focus on benchmarking the total off-line time required (i.e., the time required to read/parse the input DTA's, plus the time taken for computing the DTA $\mathcal{B}$ from $\mathcal{A}_{\psi_i}$ and $\mathcal{A}_{\varphi}$ and storing it as a UPPAAL model, plus the time taken for computing all the reachable symbolic states and the table containing the results of the emptiness checks (i.e., reachability of locations in $F$ and reachability of locations in $F'$) for each symbolic state. The examples vary in number of locations and transitions in the DTA's $\mathcal{A}_{\psi_i}$ 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 Ubuntu 14.04 LTS.

*Results.* In Table 3, entry *Example* shows the properties considered, *Loc* ($\mathcal{A}_{\psi_i}$) is the number of locations in the automaton $\mathcal{A}_{\psi_i}$ and *Loc* ($\mathcal{A}_{\varphi}$) is the number of locations in the automaton $\mathcal{A}_{\varphi}$. The entry *Time (Sec.)* presents the total (off-line) time, and the entry *Size (Bytes)* shows the size of the table (in Bytes) that stores all the reachable symbolic states in the automaton $\mathcal{A}_{\psi_i} \times \mathcal{A}_{\varphi}$, and information about reachability of locations $F$ and $F'$ for each entry.

Table 3: Performance analysis.

| *Example* | *Loc* ($\mathcal{A}_{\psi_i}$) | *Loc* ($\mathcal{A}_{\varphi}$) | *Time (Sec.)* | *Size (Bytes)* |
|---|---|---|---|---|
| $\psi_1, \varphi$ | 3 | 8 | 1.520 | 196,236 |
| $\psi_2, \varphi$ | 4 | 8 | 1.992 | 245,192 |
| $\psi_3, \varphi$ | 5 | 8 | 2.515 | 294,215 |
| $\psi_4, \varphi$ | 6 | 8 | 3.162 | 343,088 |
| $\psi_5, \varphi$ | 7 | 8 | 3.902 | 392,044 |
| $\psi_6, \varphi$ | 8 | 8 | 4.802 | 440,807 |
| $\psi_7, \varphi$ | 9 | 8 | 5.839 | 489,787 |
| $\psi_8, \varphi$ | 10 | 8 | 6.878 | 538,795 |
| $\psi_9, \varphi$ | 11 | 8 | 8.501 | 592,168 |
| $\psi_{10}, \varphi$ | 12 | 8 | 9.994 | 641,108 |




(a) Time (Sec.): $\psi_i, \varphi$.

(b) Memory (Bytes): : $\psi_i, \varphi$.

Figure 7: Performance Analysis.

From Table 3 and Figure 7, we can notice that the off-line time increases with increase in $i$ as the number of locations in $\mathcal{A}_{\psi_i}$ (and correspondingly in the DTA $\mathcal{B}$) grow. We can notice that the increase is not linear because with increase in number of locations, the time required to compute emptiness check (i.e., reachability of a set of locations) increases. Regarding memory, we can notice that increasing the number of locations in $\mathcal{A}_{\psi_i}$ increases the size of the emptiness check table linearly.

Note that to store the results of the pre-computation phase (reachable symbolic states and the emptiness test results for each symbolic state), additional memory is required. For instance, in the example with 10 locations in $\mathcal{A}_{\psi_i}$ and 8 locations in $\mathcal{A}_{\varphi}$, we can notice that we need around 0.641108 MB of additional memory. Thus there is a trade-off between memory usage Vs. online performance. For larger properties/models, we meed more memory to store the results of the pre-computation phase in order to improve the performance at runtime.

## 7. Related work

Several approaches have been proposed for the runtime verification and enforcement of properties, that are related to the predictive RV framework proposed in this paper. We briefly discuss about RV frameworks in the untimed setting, and the predictive RV framework introduced in [7, 8], and a brief comparison of the predictive RV framework with the model checking problem. Later, we discuss about frameworks proposed for the runtime monitoring of real-time properties in the non-predictive case.

*Runtime verification.* Runtime verification is an emerging research area, and many RV frameworks have been proposed. For a short survey and tutorial on runtime verification techniques, see [32, 33]. RV approaches focus on the synthesis of a verification monitor from a given property. Properties are usually specified in high-level formalisms such as Linear Temporal Logic (LTL) and regular expressions. For instance, in [2] monitors are generated from specifications expressed in LTL. The approach in [2] involves building an automaton that accepts $\varphi$, and an automaton that accepts negation of $\varphi$, and testing emptiness from the states reached in both these automata upon the current observed input word. In all these approaches, the system being monitored is considered to be a *black-box*, and predicting the future of the current observation is not considered.

*Predictive runtime monitoring of* untimed *properties.* In [7, 8], predictive RV framework for properties expressed in LTL has been proposed. This approach considers some knowledge (abstract model of the system being monitored) $\psi$ to be known in addition to the property $\varphi$ to be verified. Property $\psi$ helps the monitor to anticipate possible extensions of the current observation, enabling the monitor to foresee property satisfaction or violation before the observed execution satisfies or violates it. In this approach, since $\psi$ defines all possible input words, the two automata considered here are $\mathcal{A}\psi \times \mathcal{A}\varphi$, and $\mathcal{A}\psi \times \overline{\mathcal{A}\varphi}$, and the monitor provides a verdict by testing emptiness from the states reached in both of these product automata.

Runtime enforcement (RE) is a technique [34, 35, 36] to monitor the execution of a system and ensure its compliance against a set of properties. Using an enforcement monitor (EM), an (untrustworthy) input sequence is modified into an output sequence that complies with a property $\varphi$ (e.g., a safety requirement). Recently in [18], we presented predictive runtime enforcement framework for untimed properties, where a-priori knowledge of the system $\psi$ allows to output some events immediately (whenever possible) instead of delaying them until more events are observed. The predictive RE approach in [18] uses operations on automata such as product and negation, and emptiness-check to decide whether an input event can be released immediately. However, there are several aspects that need to be explored to be able to extend the predictive RE approach in [18] for timed properties, since delaying a timed event has the affect of changing the timed event at the output of the enforcement mechanism. Thus, the sequence released as output may not be a prefix of the input observation.

In [7, 8] predictive RV of untimed properties is introduced, and in [18] predictive RE of untimed properties is introduced. None of these works addresses the runtime monitoring (verification, enforcement) of timed properties. In this paper, we mainly focus on predictive runtime verification of timed properties.

*Predictive runtime verification versus model checking.* In predictive RV frameworks such as [7], when we consider $\psi$ to be the model of the system, initially when the input word (execution of the system) is $\epsilon$, the predictive RV problem answers the model-checking question, i.e., whether all possible executions of the system satisfies the property $\varphi$ we want to check/verify. Thus, in this case the complexity will be similar to performing model-checking. However, when $\psi$ (property of the system) is an abstraction (over-approximation) of the model of the system, the complexity varies with the level of abstraction of $\psi$. For instance, when the system is considered to be a black-box, $i.e., \psi = \Sigma^*$, then the predictive RV problem reduces to the non-predictive case, and is thus simpler.

*Runtime monitoring of timed properties.* We briefly discuss about some approaches for the runtime monitoring of timed properties for real-time systems. In timed approaches, the observed time between events may influence the truth-value of the property.

A comprehensive testing and monitoring framework based on timed automata is proposed in [10]. This work uses (possibly non-deterministic and partially observable) timed automata with input and output actions as the basic model, and introduces on-the-fly determinization as the key technique for automatic generation of online as well as offline monitors and testers. [10] also proposes a timed conformance relation, and considers both analog-clock (infinite-precision) and digital-clock (finite-precision) monitors and testers. The latter can be either programs, as in this paper, but in some cases they can also be represented symbolically as timed automata. Coverage criteria are also discussed in [10]. [9] shows how the framework of [10] can be combined with automatic generation of timed automata from higher-level planning specifications, and reports on an monitoring application with NASA's Mars K9 rover controller as the case study.

There are some works related to the runtime verification of timed properties that propose to synthesise decision procedures for logic-based timed specification formalisms. Bauer et al. propose an approach to runtime verify timed-bounded properties expressed in a variant of Timed Linear Temporal Logic (TLTL) [2]. Another variant of LTL in a timed context is Metric Temporal Logic (MTL), a dense extension of LTL. Thati et al. propose an online monitoring algorithm which works by rewriting of the monitored formula [37]. A general comparison of monitoring algorithms for real-time systems is provided in [6] by Basin et al. Most of the other works related to runtime verification of timed properties are tools such as AMT [38] and LARVA [39]. UPPAAL-TRON [40] is a tool for online testing of real-time properties defined as timed automata.

There are some recent works related to the runtime enforcement of timed properties [41, 42], where enforcement monitors are synthesized from timed properties defined as timed automata.

All the previously mentioned approaches related to monitoring of timed properties consider the system being monitored as a black-box. In this paper, we introduce a predictive framework for the monitoring of timed properties, and we focus on the predictive runtime verification problem for timed properties.

## 8. Conclusion and future work

*Conclusion.* Runtime monitoring techniques are essential for safety-critical systems, and we commonly encounter requirements with time constraints in such systems. In this paper, we present a *predictive* runtime verification framework for systems with timing requirements. Using knowledge about system's behavior allows a monitor to anticipate ("predict") future input events, allowing the monitor to *foresee* satisfaction (violation) of the property $\varphi$ being monitored. The monitor can thus provide conclusive verdict earlier (whenever possible), which is certainly advantageous when we consider monitoring of timed properties. Whenever the monitor provides a conclusive verdict, it also provides information about the minimum time instant when a violation (satisfaction) can happen in the future, that may allow to take some corrective action before the actual violation occurs.

The timed property to be verified at runtime ($\varphi$) as well as the knowledge about the system ($\psi$) are modeled as deterministic timed automata. We show how to synthesize a runtime verification monitor for any regular timed property given by a DTA and provide algorithms to implement these mechanisms. The proposed algorithm has been implemented, illustrating the practical feasibility of synthesis of predictive RV monitors for timed properties.

*Future work.* There are some recent works [41, 42] related to enforcement for real-time systems. It is important to study whether prediction (using available knowledge of the system) is useful in runtime enforcement frameworks for real-time properties. We are working on extending runtime enforcement frameworks for real-time systems to consider prediction. We also plan to further study on applying our approach to real case-studies, and also evaluate and compare online performance of the proposed predictive monitoring approach with non-predictive monitoring, and other existing tools for monitoring timed properties.

## 9. Acknowledgements

This work was supported in part by the Academy of Finland and the U.S. National Science Foundation (awards #1329759 and #1139138).

## Bibliography

[1] K. Havelund, A. Goldberg, Verify your runs, in: Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 374–383. doi:10.1007/978-3-540-69149-5_40.
URL http://dx.doi.org/10.1007/978-3-540-69149-5_40

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

[3] J. O. Blech, Y. Falcone, K. Becker, Towards certified runtime verification, in: T. Aoki, K. Taguchi (Eds.), Formal Methods and Software Engineering: 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 494–509. doi:10.1007/978-3-642-34281-3_34.
URL http://dx.doi.org/10.1007/978-3-642-34281-3_34

[4] Y. Falcone, J. Fernandez, L. Mounier, Runtime verification of safety-progress properties, in: S. Bensalem, D. A. Peled (Eds.), Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers, Vol. 5779 of Lecture Notes in Computer Science, Springer, 2009, pp. 40–59. doi:10.1007/978-3-642-04694-0_4.
URL http://dx.doi.org/10.1007/978-3-642-04694-0

[5] A. Bauer, M. Leucker, C. Schallhart, Comparing LTL semantics for runtime verification, J. Log. Comput. 20 (3) (2010) 651–674. doi:10.1093/logcom/exn075.

[6] D. Basin, F. Klaedtke, E. Zalinescu, Algorithms for monitoring real-time properties, in: S. Khurshid, K. Sen (Eds.), Proceedings of the 2nd International Conference on Runtime Verification (RV 2011), Vol. 7186 of Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 260–275. doi:10.1007/978-3-642-29860-8_20.

[7] M. Leucker, Sliding between model checking and runtime verification, in: Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 82–87. doi:10.1007/978-3-642-35632-2_10.
URL http://dx.doi.org/10.1007/978-3-642-35632-2_10

[8] X. Zhang, M. Leucker, W. Dong, Runtime verification with predictive semantics, in: A. E. Goodloe, S. Person (Eds.), NASA Formal Methods: 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 418–432. doi:10.1007/978-3-642-28891-3_37.
URL http://dx.doi.org/10.1007/978-3-642-28891-3_37

[9] S. Bensalem, M. Bozga, M. Krichen, S. Tripakis, Testing Conformance of Real-Time Applications by Automatic Generation of Observers, in: 4th International Workshop on Runtime Verification (RV'04), Vol. 113 of Electr. Notes Theor. Comput. Sci., Elsevier, 2005, pp. 23–43.

[10] M. Krichen, S. Tripakis, Conformance Testing for Real-Time Systems, Formal Methods in System Design 34 (3) (2009) 238–304.

[11] T. Nguyen, E. Bartocci, D. Ničković, R. Grosu, S. Jaksic, K. Selyunin, The harmonia project: Hardware monitoring for automotive systems-of-systems, in:

Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II, Springer International Publishing, Cham, 2016, pp. 371–379. doi:10.1007/978-3-319-47169-3_28.
URL http://dx.doi.org/10.1007/978-3-319-47169-3_28

[12] A. Bauer, J.-C. Küster, G. Vegliach, Runtime verification meets android security, in: Proceedings of the 4th International Conference on NASA Formal Methods, NFM'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 174–180. doi:10.1007/978-3-642-28891-3_18.
URL http://dx.doi.org/10.1007/978-3-642-28891-3_18

[13] K. El-Harake, Y. Falcone, W. Jerad, M. Langet, M. Mamlouk, Blocking advertisements on android devices using monitoring techniques, in: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 239–253. doi:10.1007/978-3-662-45231-8_17.
URL http://dx.doi.org/10.1007/978-3-662-45231-8_17

[14] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, R. Villemaire, Runtime verification of web service interface contracts, IEEE Computer 43 (3) (2010) 59–66. doi:10.1109/MC.2010.76.
URL http://dx.doi.org/10.1109/MC.2010.76

[15] H. Raffelt, B. Steffen, T. Berg, T. Margaria, Learnlib: a framework for extrapolating behavioral models, International Journal on Software Tools for Technology Transfer 11 (5) (2009) 393–407. doi:10.1007/s10009-009-0111-8.
URL http://dx.doi.org/10.1007/s10009-009-0111-8

[16] V. D'Silva, D. Kroening, G. Weissenbacher, A survey of automated techniques for formal software verification, IEEE Trans. on CAD of Integrated Circuits and Systems 27 (7) (2008) 1165–1178. doi:10.1109/TCAD.2008.923410.
URL http://dx.doi.org/10.1109/TCAD.2008.923410

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

[18] S. Pinisetty, V. Preoteasa, S. Tripakis, T. Jéron, Y. Falcone, H. Marchand, Predictive runtime enforcement, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, ACM, New York, NY, USA, 2016, pp. 1628–1633. doi:10.1145/2851613.2851827.
URL http://doi.acm.org/10.1145/2851613.2851827

[19] S. Pinisetty, V. Preoteasa, S. Tripakis, T. Jéron, Y. Falcone, H. Marchand, Predictive runtime enforcement (extended version), Formal Methods in System Design (Currently under submission).

[20] O. Finkel, Undecidable problems about timed automata, in: Formal Modeling and Analysis of Timed Systems: 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 187–199. doi:10.1007/11867340_14.
URL http://dx.doi.org/10.1007/11867340_14

[21] S. Tripakis, Folk theorems on the determinization and minimization of timed automata, Inf. Process. Lett. 99 (6) (2006) 222–226. doi:10.1016/j.ipl.2006.04.015.
URL http://dx.doi.org/10.1016/j.ipl.2006.04.015

[22] C. Baier, N. Bertrand, P. Bouyer, T. Brihaye, When are timed automata determinizable?, in: Automata, Languages and Programming, 36th Internatilonal Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part II, 2009, pp. 43–54. doi:10.1007/978-3-642-02930-1_4.
URL http://dx.doi.org/10.1007/978-3-642-02930-1_4

[23] A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly—but how ugly is ugly?, in: Proceedings of the 7th International Workshop on Runtime Verification (RV'07), Vol. 4839 of Lecture Notes in Computer Science, Springer-Verlag, Springer-Verlag, Vancouver, Canada, 2007, p. 126–138.

[24] P. Bouyer, M. Colange, N. Markey, Symbolic optimal reachability in weighted timed automata, in: Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I, Springer International Publishing, Cham, 2016, pp. 513–530. doi:10.1007/978-3-319-41528-4_28.
URL http://dx.doi.org/10.1007/978-3-319-41528-4_28

[25] S. Tripakis, C. Courcoubetis, Extending Promela and Spin for Real Time, in: T. Margaria, B. Steffen (Eds.), 2nd Intl. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96), Vol. 1055 of LNCS, Springer, 1996, pp. 329–348.

[26] C. Daws, A. Olivero, S. Tripakis, S. Yovine, The Tool KRONOS, in: R. Alur, T. Henzinger, E. Sontag (Eds.), Hybrid Systems III: Verification and Control, Vol. 1066 of LNCS, Springer, 1996, pp. 208–219.

[27] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, International Journal on Software Tools for Technology Transfer 1 (1997) 134–152.

[28] S. Tripakis, Checking Timed Büchi Automata Emptiness on Simulation Graphs, ACM Transactions on Computational Logic (TOCL) 10 (3) (2009) 1–19. doi:http://doi.acm.org/10.1145/1507244.1507245.

[29] D. Dill, Timing assumptions and verification of finite-state concurrent systems, in: J. Sifakis (Ed.), Automatic Verification Methods for Finite State Systems, Vol. 407 of LNCS, Springer, 1989, pp. 197–212.

[30] P. Niebert, S. Tripakis, S. Yovine, Minimum-time reachability for timed automata, in: IEEE Mediteranean Control Conference, 2000.

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

[32] M. Leucker, C. Schallhart, A brief account of runtime verification, Journal of Logic and Algebraic Programming 78 (5) (2009) 293–303.
URL http://dx.doi.org/10.1016/j.jlap.2008.08.004

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

[34] F. B. Schneider, Enforceable security policies, ACM Trans. Inf. Syst. Secur. 3 (1) (2000) 30–50.

[35] Y. Falcone, L. Mounier, J.-C. Fernandez, J.-L. Richier, Runtime enforcement monitors: composition, synthesis, and enforcement abilities, FMSD 38 (3) (2011) 223–262.

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

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

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

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

[40] K. G. Larsen, M. Mikucionis, B. Nielsen, A. Skou, Testing real-time embedded software using uppaal-tron: An industrial case study, in: Proceedings of the 5th ACM International Conference on Embedded Software, EMSOFT '05, ACM, New York, NY, USA, 2005, pp. 299–306. doi:10.1145/1086228.

1086283.
URL http://doi.acm.org/10.1145/1086228.1086283

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

[42] Y. Falcone, T. Jéron, H. Marchand, S. Pinisetty, Runtime enforcement of regular timed properties by suppressing and delaying events, Science of Computer Programming 123 (2016) 2 – 41. doi:http://dx.doi.org/10.1016/j.scico.2016.02.008.