Runtime Enforcement of Parametric Timed Properties with Practical Applications

Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand

Runtime Enforcement of Parametric Timed Properties with Practical Applications

Srinivas Pinisetty * **Yliès Falcone** ** **Thierry Jéron** * **Hervé Marchand** *

* *INRIA Rennes, Bretagne Atlantique, France (First.Last@inria.fr).*
** *Laboratoire d'Informatique de Grenoble, Université Grenoble I, France (Ylies.Falcone@ujf-grenoble.fr).*

Abstract: Runtime enforcement (RE) is a technique where a so-called monitor modifies the execution of a system to comply with a desired property. RE consists in using a so called monitor to modify an input sequence of events so that it complies with the property. Very few convincing applications of runtime enforcement have been proposed so far since most of the proposed approaches remain on the theoretical level. In network security, RE monitors can detect and prevent Denial-of-Service attacks. In resource allocation, RE monitors can ensure fairness. Specifications in these domains express data-constraints over the received events where the timing between events matters. To formalize these requirements, we introduce Parameterized Timed Automata with Variables (PTAVs), an extension of Timed Automata (TAs) with internal and external variables. We then extend enforcement for TAs to enforcement for PTAVs. We model requirements from the considered application domains and show how enforcement monitors can ensure system correctness w.r.t. these requirements. Finally, we propose a prototype implementation to experiment RE monitors on some properties. Our experiments and the performance of RE monitors demonstrate the feasibility of our approach.

Keywords: Monitoring, Runtime Enforcement, Timed Automaton, Parametric property.

1. INTRODUCTION

Observing the behavior of a system and checking its compliance to a given user-provided specification (describing the intended behavior of the system) has been referred under different names such as passive testing or runtime verification. Going further, runtime enforcement is a technique aiming at ensuring that a (possibly incorrect) observation input to the (so-called) monitor is transformed and output as a correct observation.

Many theoretical frameworks have been proposed for the runtime enforcement of high-level specifications on systems (cf. Falcone et al. (2011) or Falcone (2010) for an overview). In these frameworks, an enforcement monitor is (automatically) obtained from a specification, and acts as a "filter" on (some observable and controllable representation) of the behavior of the system, possibly using an auxiliary memory. It is worth mentioning that runtime enforcement differs from supervisory-control as enforcement monitors are usually plugged at the exit or entrance of the system, but does not modify its internal behavior, contrarily to controllers. Moreover, in enforcement monitoring, knowing the property to be enforced is sufficient, and we don't require knowledge or model of the system. In enforcement frameworks, specifications are formalized as *propositional* properties (i.e., sets of words over a propositional alphabet) and executions are words over the considered alphabet. Enforcement monitors input a word and suppress, insert, or reorder events so as to output a word that complies to the property.

A limitation to the applicability of theoretical approaches to runtime enforcement is the expressiveness of the considered specification formalisms. In most modern application domains, propositional specification formalisms are not expressive enough to meet and formalize complex requirements. Time and data are two particularly desirable features. In timed specifications the physical time that elapses between two events matters, i.e., it influences satisfiability. Considering timed specifications in runtime techniques is intricate because the execution time of the monitor, which is not part of the initial system, can change the evaluation of an execution, w.r.t. the same execution on the non-monitored system. Handling parameters in (the verification approaches of) monitoring is receiving a growing attention. Parametric specifications feature events that carry data from the execution of the monitored system. Few approaches tackle parametric specifications (cf. Chen and Rosu (2009); Barringer et al. (2012)). These approaches are concerned only with verification and do not consider time.

The two following (simplified) properties of mail servers cannot be specified using propositional specification formalisms.

R1 *If the number of request messages from a client is greater than max_req, then there should be a delay of at least del t.u. before responding positively to the client.*

R2 *After processing a request message from a client, if the server response message is an error (user unknown or not found), then there should be a delay of at least 10 t.u. before sending this reply message back to the client.*

To express these properties, we need features to express constraints over time and data. Moreover, the server has to differentiate the messages from each client and treat them separately. Features to keep track of some information internally, such as the number of request messages received, are also necessary.

In this paper we make one step towards practical runtime enforcement by considering event-based specifications where i) time between events matters and ii) events carry data values from the monitored system. We refer to this problem as *en-*

forcement monitoring for parametric timed specifications. Because of the timing feature, we consider enforcement monitors as *time retardants*. That is, monitors are endowed with only one realistic feature (for the considered application domains): being able to *delay* events. Monitors read a trace of events and output the same trace where delays between consecutive events are augmented to comply with a specification. Following the compositional approach (the so-called "plugin approach") introduced in Chen and Rosu (2009), for a parametric timed specification, the input trace is sliced according to the parameter value of events, and redirected to the appropriate instance of the monitor. Contrary to the plugin approach which focuses on verification in the untimed case, events cannot be duplicated. Thus, slicing should be performed in such a way that each event is sent to only one enforcement monitor.

This paper extends runtime enforcement for *non*-parametric (i.e., propositional) timed properties, introduced in Pinisetty et al. (2012). We introduce *Parametrized Timed Automata with Variables* (P-TAVs) (Sec. 3). The expressiveness features of P-TAVs have been chosen as a balance between expressiveness and efficiency of the synthesized enforcement monitors and ensure that the previously mentioned requirement on slicing holds. To guide us in the choice of expressiveness features we considered requirements in several application domains. Moreover, we show how the "slicing" approach of Chen and Rosu (2009) can be adapted to our specification formalism (Sec. 4). Furthermore, we show how these application domains can benefit from using runtime enforcement monitors synthesized from requirements formalized as P-TAVs (Sec. 5). The enforcement monitors synthesized from P-TAVs are able to ensure several requirements in the considered application domains. Our experiments validate our choice on the expressiveness of P-TAVs and assess the efficiency of obtained enforcement monitors.

2. PRELIMINARIES AND NOTATION

Monitors input and output timed words. A *timed word* is a sequence $\sigma = e_1 \cdot e_2 \cdots e_n$ of events where $\forall i \in [1, n] : e_i = (\delta_i, a_i(\pi_i, \eta_i))$. Action $a_i \in \Sigma$ is an action in an alphabet Σ, and $\pi_i \in \mathcal{D}_p$ is the value of parameter p, identifying the monitor instance to which the action should be fed as input. $\eta_i \in \mathcal{D}_V$ is a vector of values of a tuple of variables V. We denote by $\mathrm{delay}\,(\delta, a\,(\pi, \eta)) \stackrel{\mathrm{def}}{=} \delta$ the projection of events on delays, by σ^i, the i^{th} event, and by $\mathrm{time}(\sigma)$ the total duration of σ, i.e., the sum of its delays. $|\sigma|$ is used to denote the length of σ. The projection of σ on actions is denoted by $\Pi_\Sigma(\sigma)$.

For a given parameter value π, we denote by $\sigma{\downarrow_\pi}$ the *projection* of σ on the actions carrying parameter value π, where delays are summed up in the delay of the next action carrying π, e.g., if $\sigma = (0.5, a(1, \eta_1)) \cdot (0.3, a(2, \eta_2)) \cdot (0.2, a(1, \eta_3)) \cdot (0.4, a(2, \eta_4))$, then $\sigma \downarrow_1 = (0.5, a(1, \eta_1)) \cdot (0.5, a(1, \eta_3))$ and $\sigma \downarrow_2 = (0.8, a(2, \eta_2)) \cdot (0.6, a(2, \eta_4))$.

Conversely, we (inductively) define the *merge* of several sequences related to different parameter values (where we omit vectors of external variables for readability) as follows:

- $\mathrm{merge}\{\epsilon\} = \epsilon$;
- $\mathrm{merge}\{\sigma_1, \ldots, \sigma_n\} = \mathrm{merge}\{\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n\}$ if $\sigma_i = \epsilon$;
- $\mathrm{merge}\left\{(\delta_1, a_1(1)) \cdot \sigma_1, \ldots, (\delta_n, a_n(n)) \cdot \sigma_n\right\} =$ let i s.t. $\delta_i = \min\{\delta_j \mid j \in [1, n]\}$ in $(\delta_i, a_i(i)) \cdot$ $\mathrm{merge}\left\{(\delta_1 - \delta_i, a_1(1)) \cdot \sigma_1, \ldots, \sigma_i, \ldots, (\delta_n - \delta_i, a_n(n)) \cdot \sigma_n\right\}$.

The merge of the empty sequence ϵ is the empty sequence. The merge of a set of sequences that contains an empty sequence is equal to the merge of this set where the empty sequence has been removed. The merge of a set of non-empty sequences is the sequence s.t. the first event is the one of the merged sequences with the least delay (say the i-th sequence), and, the remainder of the sequence is the merge of σ_i with the previous sequences where δ_i is substracted from the delays of the first events for the sequences with index different from i.

For example, if $\sigma_1 = (0.5, a(1, \eta_{1.1})) \cdot (0.5, a(1, \eta_{1.2}))$ and $\sigma_2 = (0.8, a(2, \eta_{2.1})) \cdot (0.6, a(2, \eta_{2.2}))$ then $\mathrm{merge}\{\sigma_1, \sigma_2\} = (0.5, a(1, \eta_{1.1})) \cdot (0.3, a(2, \eta_{2.1})) \cdot (0.2, a(1, \eta_{1.2})) \cdot (0.4, a(2, \eta_{2.2}))$.

The *domain* of a trace σ, denoted by $\mathcal{D}om(\sigma)$, is the set of monitor instances (set of values appearing as the first parameter of events in σ).

The *observation of σ at time t* is the longest prefix of σ with duration lower than t: $\mathrm{obs}(\sigma, t) \stackrel{\mathrm{def}}{=}$ $\max_{\preccurlyeq}\{\sigma' \in (\mathbb{R}_{\geq 0} \times \Lambda)^* \mid \sigma' \preccurlyeq \sigma \wedge \mathrm{time}(\sigma') \leq t\}$ where $\preccurlyeq$ denotes the prefix ordering.

For $\sigma, \sigma' \in (\mathbb{R}_{\geq 0} \times \Lambda)^*$, we say that σ' *delays* σ (denoted as $\sigma' \preccurlyeq_d \sigma$) if their projections on actions are ordered by prefix ($\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$) but delays in σ' may be increased: $\forall i \leq |\sigma'| : \mathrm{delay}({\sigma'}^i) \geq \mathrm{delay}(\sigma^i)$.

3. PARAMETRIZED TIMED AUTOMATA WITH VARIABLES

To handle requirements with constraints on data, events with associated values, and separate input flows according to parameter values, we define an extension of timed automata called Parametrized Timed Automata with Variables (P-TAV). P-TAVs are partly inspired from Input-Output Symbolic Transition Systems (IOSTS) of Rusu et al. (2000), Timed Input-Output Symbolic Transition Systems (TIOSTS) of Andrade et al. (2011), parametric trace slicing of Chen and Rosu (2009) and Quantified Event Automata of Barringer et al. (2012) (see Sec. 7 for a comparison). The features of P-TAVs have been chosen to fit a balance between expressiveness (to express requirements from some application domains - see Sec. 5) and runtime efficiency.

3.1 Definition

A P-TAV can be seen as a timed automaton with finite set of locations, and a finite set of clocks used to represent time evolution, extended with internal and external variables used for representing system data. A transition comprises of an action carrying values of external variables, a guard on internal variables, external variables and clocks, and an assignment of internal variables, and reset of clocks. External variables model the data carried by the actions from the monitored system, and internal variables are used for internal computation. For a variable v, $\mathcal{D}_v$ denotes its domain, and for a tuple of variables $V = \langle v_1, \ldots, v_n\rangle$, $\mathcal{D}_V$ is the product domain $\mathcal{D}_{v_1} \times \cdots \times \mathcal{D}_{v_n}$. A predicate $P(V)$ on a tuple of variables V is a logical formula whose semantics is a function $\mathcal{D}_V \rightarrow \{\mathtt{true}, \mathtt{false}\}$, and can also be seen as the subset of $\mathcal{D}_V$ which maps to $\mathtt{true}$. A valuation of the variables in V is a mapping ν which maps every variable $v \in V$ to a value $\nu(v)$ in $\mathcal{D}_v$.

Given X a set of clocks, and $\mathbb{R}_{\geq 0}$ the set of non-negative real numbers, a clock valuation is a mapping $\chi : X \rightarrow \mathbb{R}_{\geq 0}$. If χ is a valuation over X and $t \in \mathbb{R}_{\geq 0}$, then $\chi + t$ denotes the valuation that assigns $\chi(x) + t$ to every $x \in X$. For $X' \subseteq X$, $\chi_{[X' \leftarrow 0]}$

denotes the valuation equal to χ on $X \setminus X'$ and assigning 0 to all clocks in X'.

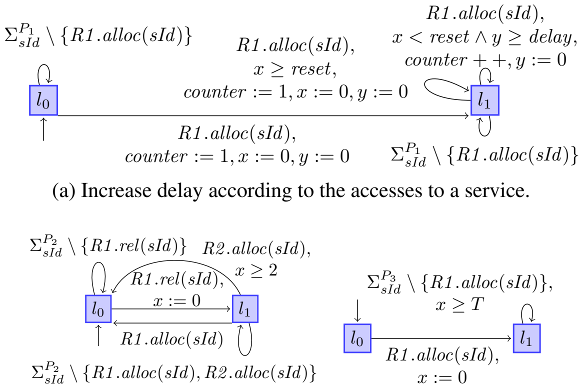
Definition 1. (Syntax of P-TAVs). A P-TAV is a tuple $\langle p, V, C, \Theta, L, l_0, L_G, X, \Sigma_p, \Delta\rangle$ where:

- p is a parameter ranging over a countable set $\mathcal{D}_p$;
- V is a tuple of typed internal variables and C is a tuple of external variables;
- $\Theta \subseteq \mathcal{D}_{\{p\}\cup V}$ the initial condition, is a computable predicate over V and p;
- L is a finite non-empty set of locations, with $l_0 \in L$ the initial location, and $L_G \subseteq L$ the set of accepting locations;
- Σ is a non-empty finite set of actions, and an action $a \in \Sigma$ has a signature $sig(a) = \langle t_0, t_1, \ldots, t_k\rangle$ which is a tuple of types of the external variables, where $t_0 = \mathcal{D}_p$ is the type of the parameter p;
- X is a finite set of clocks;
- Δ is a finite set of transitions, and each transition $t \in \Delta$ is a tuple $\langle l, a, p, c, G, A, l'\rangle$ also written $l \xrightarrow{a(p,c),G(V,p,c),V':=A(V,p,c)} l'$ such that,
  - $l, l' \in L$ are respectively the origin and target locations of the transition;
  - $a \in \Sigma$ is the action, p is the parameter and $c = \langle c_1, \ldots c_k\rangle$ is a tuple of external variables local to the transition;
  - $G = G^D \wedge G^X$ is the guard where
    - $G^D \subseteq \mathcal{D}_V \times \mathcal{D}_{sig(a)}$ is a computable predicate over internal variables, the parameter and external variables in $V \cup \{p\} \cup c$;
    - G^X is a clock constraint over X defined as a conjunction of constraints of the form $x \sharp f(V \cup \{p\} \cup c)$, where $x \in X$ and $f(V \cup \{p\} \cup c)$ is a computable function, and $\sharp \in \{<, \leq, =, \geq, >\}$;
  - $A{=}(A^D, A^X)$ is the assignment of the transition where
    - $A^D : \mathcal{D}_V \times \mathcal{D}_{sig(a)} \rightarrow \mathcal{D}_V$ defines the evolution of internal variables.
    - $A^X \subseteq X$ is the set of clocks to be reset.

A P-TAV $\mathcal{A}(p)$ should be understood as a pattern, where parameter p is a constant defined at runtime. For a value π of p, the instance of $\mathcal{A}(p)$ is denoted as $\mathcal{A}(\pi)$. P-TAVs allow to describe a set of identical timed automata extended with internal and external variables that only differ by the value of p. In Sec. 4 we explain how a parametric enforcement monitor generated from a P-TAV is instantiated into a set of monitors (generated on-the-fly), one for each value of p, each observing the corresponding projection $\sigma\downarrow_\pi$ of the input timed word σ. This thus allows for example to tackle timed words corresponding to several sessions of a web service where each session is treated independently.

Remark 1. (Multiple parameters). For the sake of simpler notations, P-TAVs are presented with only one parameter p. P-TAVs can handle a tuple of parameters. Using some indexing mechanism, each combination of values of the parameters can be mapped to a unique value.

Let $\mathcal{A}(p) = \langle p, V, C, \Theta, L, l_0, L_G, X, \Sigma, \Delta\rangle$ be a P-TAV. For a value π of parameter p, the semantics of the instance $\mathcal{A}(\pi)$ is a timed transition system, where states explore the set of locations, valuations of internal variables V and clocks X, and transitions explore pairs of delays in $\mathbb{R}_{\geq 0}$ and actions associated with values of the parameter and external variables in C. [1]


$\Sigma^{P_1}_{sId} \setminus \{R1.alloc(sId)\}$
$R1.alloc(sId)$, $x \geq reset$, $counter := 1, x := 0, y := 0$
$R1.alloc(sId)$, $x < reset \wedge y \geq delay$, $counter++, y := 0$
l_0
l_1
$R1.alloc(sId)$, $counter := 1, x := 0, y := 0$
$\Sigma^{P_1}_{sId} \setminus \{R1.alloc(sId)\}$
$\Sigma^{P_2}_{sId} \setminus \{R1.rel(sId)\}$
$R2.alloc(sId)$, $x \geq 2$
$R1.rel(sId)$, $x := 0$
l_0
l_1
$R1.alloc(sId)$
$\Sigma^{P_2}_{sId} \setminus \{R1.alloc(sId), R2.alloc(sId)\}$
$\Sigma^{P_3}_{sId} \setminus \{R1.alloc(sId)\}$, $x \geq T$
l_0
l_1
$R1.alloc(sId)$, $x := 0$

(a) Increase delay according to the accesses to a service.

(b) At least 2 t.u. before allocating R2 after releasing R1.

(c) At least T t.u. before operating on a resource.

Fig. 1. P-TAVS for resource allocation.

Definition 2. (Semantics of P-TAVs). For a value π of p, the semantics of $\mathcal{A}(\pi)$, is a timed transition system $[\![\mathcal{A}(\pi)]\!] = \langle Q, q_0, Q_G, \Gamma, \rightarrow\rangle$, defined as follows:

- $Q = L \times \mathcal{D}_V \times (X \rightarrow \mathbb{R}_{\geq 0})$, is the set of states of the form $q = \langle l, \nu, \chi\rangle$ where $l \in L$ is a location, $\nu \in \mathcal{D}_V$ is a valuation of internal variables, χ is a valuation of clocks;
- $Q_0 = \{\langle l_0, \nu, \chi_{[X \leftarrow 0]}\rangle \mid \Theta(\pi, \nu) = \mathtt{true}\}$ is the set of initial states;
- $Q_G = L_G \times \mathcal{D}_V \times (X \rightarrow \mathbb{R}_{\geq 0})$ is the set of accepting states;
- $\Gamma = \mathbb{R}_{\geq 0} \times \Lambda$ where $\Lambda = \{a(\pi, \eta) \mid a \in \Sigma \wedge (\pi, \eta) \in \mathcal{D}_{sig(a)}\}$ is the set of transition labels;
- $\rightarrow \subseteq Q \times \Gamma \times Q$ the transition relation is the smallest set of transitions of the form $\langle l, \nu, \chi\rangle \xrightarrow{(\delta, a(\pi,\eta))} \langle l', \nu', \chi'\rangle$ such that $\exists\langle l, a, p, c, G, A, l'\rangle \in \Delta$, with $G^X(\chi + \delta) \wedge G^D(\nu, \pi, \eta)$ evaluating to $\mathtt{true}$, $\nu' = A^D(\nu, \pi, \eta)$ and $\chi' = (\chi + \delta)[A^X \leftarrow 0]$.

A *run* ρ of $[\![\mathcal{A}(\pi)]\!]$ from a state $q \in Q$ is a sequence of moves of the form: $\rho = q \xrightarrow{(\delta_1, a_1(\pi,\eta_1))} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n, a_n(\pi,\eta_n))} q_n$, for some $n \in \mathbb{N}$. *Accepted runs* are runs starting in Q_0 and ending in Q_G. The *trace* of a run ρ is a timed word $\sigma = (\delta_1, a_1(\pi, \eta_1)) \cdots (\delta_n, a_n(\pi, \eta_n))$ obtained by projection on pairs of delays and actions associated with the parameter value and values of external variables. The set of traces of accepted runs is noted $\mathrm{Trace}_{Q_G}(\mathcal{A}(\pi))$.

A trace σ from the system is accepted by $\mathcal{A}(p)$ if for each value π of p appearing in the trace, the projection of σ corresponding to π is an accepted run of the instance $\mathcal{A}(\pi)$: $\forall \pi \in \mathcal{D}om(\sigma) : \sigma{\downarrow_\pi} \in \mathrm{Trace}_{Q_G}(\mathcal{A}(\pi))$.

Safety P-TAVs. In this paper, we shall only consider parametric timed safety properties that can be represented by P-TAVs. [2]

Definition 3. (Safety P-TAV). A P-TAV $\mathcal{A}(p) = \langle p, V, C, \Theta, L, l_0, L_G, X, \Sigma, \Delta\rangle$ is said to be a *safety* P-TAV if $l_0 \in L_G \wedge \nexists\langle l, a, p, c, G, A, l'\rangle \in \Delta : l \in L \setminus L_G \wedge l' \in L_G$.

Then, for any instance $\mathcal{A}(\pi)$, its associated property is $\varphi_{\mathcal{A}(\pi)} = \mathrm{Trace}_{Q_G}(\mathcal{A}(\pi))$ and $\sigma \models \varphi_{\mathcal{A}(\pi)}$ is for $\sigma \in \mathrm{Trace}_{Q_G}(\mathcal{A}(\pi))$. The *parametric timed safety property* associated to $\mathcal{A}(p)$ is the set of sets of traces $\varphi_{\mathcal{A}(p)} = \{\varphi_{\mathcal{A}(\pi)} \mid \pi \in \mathcal{D}_p\}$.

[1] Considering a tuple of parameters instead of a single parameter will not have any effect on the semantics of a P-TAV.

[2] The results of Pinisetty et al. (2012) for (propositional) co-safety properties can be lifted to parametric co-safety properties in a similar fashion.

3.2 A motivating example

Concurrent accesses to shared resources by various services can lead to a Denial of Service (DoS) because of e.g., starvation or deadlock. We can formalize requirements for resource allocation and DoS prevention using P-TAVs. The P-TAV shown in Fig. 1a models the property *"There should be a dynamic delay between two allocation requests to the same resource by a service. This delay increases as the number of allocations increases and also depends on the service id"*. Squares denote accepting locations. Non-accepting locations can be omitted in (the representation of) safety P-TAVs.

We now explain the different notions of the P-TAV model through this example. The P-TAV in Fig. 1a keeps track of the number of allocations of a resource to a service, and increases the delay between allocations as the number of allocations increases. It has the set of actions $\Sigma_{sId} = \{R1.alloc(sId), R1.setMA(sId, maxAlloc)\}$. The tuple of internal variables is $\langle counter, reset, delay\rangle$, and the tuple of external variables is $\langle maxAlloc\rangle$. The variable $counter$ is an integer incremented after each $R1.alloc(sId)$ event, and $maxAlloc$ is an integer that defines the allowed number of $R1.alloc(sId)$ messages per each increment of the delay. Note that the variable $maxAlloc$ is an external variable since it is the data that is received via the action $R1.setMA(sId, maxAlloc)$. The variable $reset$ defines the time period for resetting the counter, and $delay$ is defined as 0 if $counter < maxAlloc$ and int $\left(\frac{counter*sId}{maxAlloc}\right)$ otherwise.

4. ENFORCEMENT MONITORING OF PARAMETRIC TIMED PROPERTIES

In Pinisetty et al. (2012), enforcement monitors are synthesized from propositional safety and co-safety timed properties modeled as timed automata. We first recall the basic principles for safety properties, and lift them to parametric timed properties modeled with P-TAVs.

4.1 The propositional case

The input/output behavior of an enforcement monitor is specified by an enforcement function.

Definition 4. For a given timed safety property φ, an *enforcement function* is a function E_φ from $(\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0}$ to $(\mathbb{R}_{\geq 0} \times \Sigma)^*$.

The enforcement function E_φ transforms some input timed word σ which is possibly incorrect w.r.t. φ. The output $E_\varphi(\sigma, t)$ at time t is a timed word with same actions, but possibly increased delays between actions to satisfy the property. Some requirements are defined on the enforcement function: *soundness*, *transparency* and *optimality*. Soundness means that the output of an enforcement mechanism should satisfy the property. Transparency expresses how an enforcement mechanism is allowed to correct the input sequence: the output sequence delays the input sequence. Optimality expresses how to best choose the delays to release the output as soon as possible. Formal constraints, detailed in Pinisetty et al. (2012), depend on the considered class of properties. We recall them for safety properties, at an abstract level.

Definition 5. (Soundness, transparency, optimality). An enforcement function $E_\varphi : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ for a safety property φ is:

- **sound** for an input timed word σ if at any time, the output satisfies the property: $\forall t \in \mathbb{R}_{\geq 0} : E_\varphi(\sigma, t) \models \varphi$; noted $sound(E_\varphi, \sigma)$;
- **transparent** for an input timed word σ, if at any time t, the output *delays* the input observed at time t: $\forall t \in \mathbb{R}_{\geq 0} : E_\varphi(\sigma, t) \preccurlyeq_d \mathrm{obs}(\sigma, t)$; noted $transparent(E_\varphi, \sigma)$;
- **optimal:** if $E_\varphi(\sigma, t)$ is among the longest correct timed words delaying $\mathrm{obs}(\sigma, t)$, and, every prefix of $E_\varphi(\sigma, t)$ has the shortest possible last delay; noted $optimal(E_\varphi, \sigma)$;

Example 1. (soundness, transparency and optimality). Consider the property shown in Fig. 1a. Let the initial values of variables $delay$ and $reset$ be 5 and 100, respectively. Consider the input sequence $\sigma = (2, R1.alloc(1)) \cdot (2, R1.alloc(1))$.
We illustrate the soundness, transparency and optimality constraints, on the monitor instance with parameter value $sId = 1$. Let the output of the monitor instance $E_{\varphi_{\mathcal{A}(1)}}$ for the input sequence σ be $(2, R1.alloc(1)) \cdot (5, R1.alloc(1))$. $E_{\varphi_{\mathcal{A}(1)}}$ is sound since its output satisfies the property. $E_{\varphi_{\mathcal{A}(1)}}$ is transparent since the delays of all events are greater than or equal to the actual delays in σ. $E_{\varphi_{\mathcal{A}(1)}}$ is optimal since the delays are the shortest possible, to satisfy the property. Note, $(2, R1.alloc(1)) \cdot (6, R1.alloc(1))$ also satisfies soundness and transparency, but not optimality, since 5 is the shortest possible delay of the second event.

At an abstract level, the actual definitions of the enforcement function and monitor rely on the Update function. The Update function keeps track of the current state (in the semantics of) the timed automaton, takes an input (timed) action and then $\mathrm{Update}(q, (\delta, a))$ is the smallest (optimal) delay δ' greater than or equal to δ allowing the TA to remain in an accepting location when triggering the action a while being in q.

An enforcement function E_φ is implemented by an enforcement monitor (EM) defined as a transition system. Roughly speaking, an EM is equipped with a memory and the following enforcement operations:

- **Store** reads an input event (δ, a), computes the optimal delay δ' associated with a, and stores (δ', a) in memory where δ' is computed by the Update function.
- **Dump** releases the first action a stored in memory as (δ', a), when the time elapsed since the previous dump operation is equal to δ'.
- **Idle** only allows time to elapse, and neither receives nor releases actions.

The configuration of an EM consists of the timed word in memory, two clocks keeping track of the time since the last store and last dump operations, the current location of the automaton representing the property (the state reached after the concatenation of the released output and the memory), and a Boolean indicating if this location is accepting.

For any property φ, we generate an EM, and the enforcement function E_φ implemented by the EM is sound transparent and optimal. For any input σ, at any time t, the output of the enforcement function E_φ, and output behavior of the associated EM are equal. An enforcement function describes in a declarative manner the input-output behavior of an enforcement monitor, whereas the enforcement monitor is an operational description. At any time t, the input $\mathrm{obs}(\sigma, t)$ is the concatenation of all events read by EM (Store) over various steps, until time t. At any time t, the output is the concatenation of all released events (Dump) until t.
The EM definition, with detailed description of the operations is in Pinisetty et al. (2012). Here, we explain the operations briefly via an example.

Example 2. (Enforcement operations). Consider again the scenario described in Example 1. We illustrate the enforcement operations, on the monitor instance with parameter value $sId = 1$ and input sequence $\sigma = (2, R1.alloc(1)) \cdot (2, R1.alloc(1))$. Since the first $R1.alloc(1)$ event is observed after 2 time units, store or dump operations cannot be applied, and time elapses using the idle operation. The EM reads the first $R1.alloc(1)$ event after 2 time units using the store operation, and stores $(2, R1.alloc(1))$ in the memory, meaning that the EM can release $R1.alloc(1)$ using the dump operation without introducing additional delay. Again, idle operation is used to let time elapse, and after 2 more time units, the second $R1.alloc(1)$ event is read. Then, when applying the store rule, $(5, R1.alloc(1))$ is stored in memory, where 5 is the optimal delay computed by the update function. Since an additional delay of 3 time units is needed in the output, EM can release this event using the dump operation and after letting 3 more time units elapse using the idle operation.

4.2 *The Parametric Case*

The techniques of the propositional case can be adapted to generate a parametric enforcement function/monitor from a P-TAV $\mathcal{A}(p)$ with parameter p. For space reasons, we only describe the adaptation of enforcement functions. The main adaptation lies in the definition of the Update function to compute optimal delays, taking into account the semantics of P-TAVs instead of the semantics of TAs. Indeed, for an instance $\mathcal{A}(\pi)$, given the state q reached after a given timed trace σ, a delay δ and the next action $a(\pi, \eta)$ in this trace, Update is the optimal delay δ' greater than δ for the next action to stay in accepting states. Formally, $\mathrm{Update}(q, a(\pi, \eta), \delta) = \min(D)$ if $D \neq \emptyset$, and ∞ otherwise, where $D = \{\delta' \in \mathbb{R}_{\geq 0} \mid \exists q_1 \in Q_G : q \overset{(\delta', a(\pi,\eta))}{\rightarrow} q_1 \wedge \delta' \geq \delta\}$ is the set of delays that allow to reach an accepting state from q and the next action.

Note that Update is computable as the parameter value π is a constant known at runtime, the state q (values of internal variables ν and location) and the external variables η are known, and it is assumed in Definition 1 that guards and assignments are computable.

It is then not hard to adapt the proofs of Pinisetty et al. (2012) to the following proposition:

Proposition 1. Given a safety P-TAV $\mathcal{A}(p)$ specifying a parametric timed safety property $\varphi_{\mathcal{A}(p)}$, for all $\pi \in D_p$, the enforcement function $E_{\varphi_{\mathcal{A}(\pi)}}$, obtained by following the definition in the propositional case and with the above adaptations, is sound, transparent, and optimal w.r.t. $\mathcal{A}(\pi)$, as per Definition 5.

Indexed enforcement monitors and slicing.

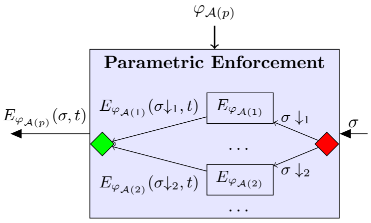

$\varphi_{\mathcal{A}(p)}$
Parametric Enforcement
$E_{\varphi_{\mathcal{A}(p)}}(\sigma, t)$
$E_{\varphi_{\mathcal{A}(1)}}(\sigma{\downarrow_1}, t)$
$E_{\varphi_{\mathcal{A}(1)}}$
$\sigma \downarrow_1$
σ
$\dots$
$E_{\varphi_{\mathcal{A}(2)}}(\sigma{\downarrow_2}, t)$
$E_{\varphi_{\mathcal{A}(2)}}$
$\sigma \downarrow_2$
$\dots$

Fig. 2: Global Scenario

The proposed strategy uses indexed enforcement monitors and slicing of Chen and Rosu (2009) that is simplified in a way slicing is meaningful for runtime enforcement. [3]

For each value π of parameter p, we have an instance of the corresponding enforcement function and enforcement monitor. For example, $E_{\varphi_{\mathcal{A}(1)}}$ is an instance of the enforcement function for $\varphi_{\mathcal{A}(1)}$. The input to the monitor σ consists of events with different values of parameter p. However, each enforcement function $E_{\varphi_{\mathcal{A}(\pi)}}$ takes as input only the projection $\sigma{\downarrow_\pi}$ of σ on actions with parameter value π. The global output is obtained by merging the outputs of all enforcement functions.

Definition 6. (Parametric Enforcement Function). The enforcement function $E_{\varphi_p}(\sigma, t) : (\mathbb{R}_{\geq 0} \times \Lambda)^* \times \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}_{\geq 0} \times \Lambda)^*$ for P-TAV $\mathcal{A}(p)$ is defined as:

$$\begin{aligned} E_{\varphi_p}(\sigma, t) = \text{let } n = |\mathcal{D}om(\sigma)| \text{ in} \\ \text{let } o_\pi = E_{\varphi_{\mathcal{A}(\pi)}}(\sigma{\downarrow_\pi}, t), \text{ for } \pi \in [1, n], \text{ in} \\ \mathrm{obs}\big(\,\mathrm{merge}(\{o_1, \dots, o_n\}), t\big). \end{aligned}$$

The global output of the enforcement function is defined as the observation of the merge of the local outputs $(o_1, \dots, o_n)$ produced by the enforcement functions synthesized for P-TAV instances that read local projections $(\sigma{\downarrow_\pi}, \pi \in [1, n])$ of the global trace σ. However, as each projection of the input stream corresponding to a value π of p is treated independently, with respect to the product of $E_{\varphi_{\mathcal{A}(\pi)}}, \pi \in \mathcal{D}_p$, the global output $E_{\varphi_p}(\sigma, t)$ remains sound (in the sense that the output satisfies $\varphi_{\mathcal{A}(p)}$), but is neither transparent nor optimal. In particular, the architecture allows to reorder independent output flows, even though each flow is not reordered.

Consequently, the following definitions of parametric soundness, transparency and optimality stem from the fact that, using an indexed strategy, enforcement monitors act and output events independently.

Definition 7. (Parametric soundness, transparency, optimality). A parametric enforcement function $E_{\varphi_p} : (\mathbb{R}_{\geq 0} \times \Lambda)^* \times \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}_{\geq 0} \times \Lambda)^*$ for a P-TAV $\mathcal{A}(p)$ with parameter p is:

- **sound** if for any input timed word σ, for any possible value π of p, at any time, the output obtained from the projection of the input timed word on π, satisfies the property: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in (\mathbb{R}_{\geq 0} \times \Lambda)^* : \mathit{sound}(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$;
- **transparent** if for any input timed word σ, for any possible value π of p, at any time t, the output obtained from the projection of the input timed word on π *delays* the input observed at time t: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in (\mathbb{R}_{\geq 0} \times \Lambda)^* : \mathit{transparent}(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$;
- **optimal** if $E_{\varphi_{\mathcal{A}(\pi)}}(\sigma, t)$ is among the longest correct timed words delaying $\mathrm{obs}(\sigma{\downarrow_\pi}, t)$, and, every prefix of $E_{\varphi_{\mathcal{A}(\pi)}}(\sigma, t)$ has the shortest possible last delay, for all values π of p: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in (\mathbb{R}_{\geq 0} \times \Lambda)^* : \mathit{optimal}(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$.

where predicates sound, $\mathit{transparent}$ and $\mathit{optimal}$ are lifted to parametric traces.

Using Proposition 1, Definitions 6 and 7 and the results from Pinisetty et al. (2012), one can prove the following proposition.

Proposition 2. Given a safety P-TAV $\mathcal{A}(p)$ specifying a parametric timed safety property $\varphi_{\mathcal{A}(p)}$, the enforcement function E_{φ_p} as per Definition 6 is sound, transparent and optimal with respect to $\mathcal{A}(\pi)$, as per Definition 7.

Example 3. Consider again the scenario described in Example 1. Consider the input sequence $\sigma = (2, R1.alloc(1)) \cdot (1, R1.alloc(2)) \cdot (1, R1.alloc(1))$. Here, $\mathcal{D}om(\sigma) = \{1, 2\}$, and thus we have two monitor instances and both are sound, transparent and optimal. The projection of the input se-

[3] Moreover, we do not discuss the dynamic creation of monitors when new values are observed in the trace. At runtime, upon a new event, a new instance of enforcement function/monitor is simply created if the parameter value has not been seen by reading previous events.

quence on the actions carrying parameter value 1 is $\sigma \downarrow_1 = (2, \mathit{R1.alloc}(1)) \cdot (2, \mathit{R1.alloc}(1))$, which is input to $E_{\varphi_{\mathcal{A}(1)}}$, and the output of $E_{\varphi_{\mathcal{A}(1)}}$ is $(2, \mathit{R1.alloc}(1)) \cdot (5, \mathit{R1.alloc}(1))$, satisfying soundness, transparency and optimality. The projection of the input sequence on the actions carrying parameter value 2 is $\sigma \downarrow_2 = (3, \mathit{R1.alloc}(2))$, which is input to $E_{\varphi_{\mathcal{A}(2)}}$, and the output of $E_{\varphi_{\mathcal{A}(2)}}$ is $(3, \mathit{R1.alloc}(2))$, satisfying soundness, transparency and optimality.

5. LEVERAGING RUNTIME ENFORCEMENT IN SOME APPLICATION DOMAINS

Let us illustrate how we can leverage runtime enforcement of parametric timed properties in some application domains. For each application domain, we provide requirements modeled as P-TAVs from which we synthesize enforcement monitors that try to maintain the P-TAV in accepting locations by introducing some delays when necessary and possible.

Using P-TAVs we obtain abstract and concise representations of the requirements. Enforcement monitors are a lightweight, modular, and flexible implementation of these requirements, and, they prevent problems in these application domains.

5.1 Resource Allocation

Let us consider a common client-server model used in distributed applications and web servers (described in Fradet and Hong Tuan Ha (2010)). A system consists of three layers: clients, services and shared resources. Clients send their requests to services and wait for their response. Requests to a service are stored in a FIFO queue, and a service processes them sequentially. To process a request, a service has to make some computation using resources such as processors, files and network connection managers. Concurrent accesses to shared resources by various services may lead to a DoS because of problems such as starvation when a service cannot allocate a shared resource, or deadlock when two services wait for a resource allocated to the other service.

Leveraging runtime enforcement for fair resource allocation. In Fradet and Hong Tuan Ha (2010), a domain-specific aspect language to prevent DoS caused by improper resource management is presented. Inspiring from this work, we formalize (richer) requirements for resource allocation and DoS prevention using P-TAVs. The event $\mathit{R1.alloc}(\mathit{sId})$ (resp. $\mathit{R2.alloc}(\mathit{sId})$) corresponds to the allocation of $R1$ (resp. $R2$), and $\mathit{R1.rel}(\mathit{sId})$ (resp. $\mathit{R2.rel}(\mathit{sId})$) corresponds to the release of $R1$ (resp. $R2$) in the session sId. The requirements are listed below.

P1 *There should be a dynamic delay between two allocation requests to the same resource by a service* (Fig. 1a).

P2 *After releasing* $R1$*, there should be a delay of at least 2 t.u. before allocating* $R2$ (Fig. 1b).

P3 *After a resource is acquired by a service, the service has to wait at least for T t.u. before performing operations on the resource.* The set of events is $\Sigma_{\mathit{sId}}^{P_3} = \{\mathit{R1.alloc}(\mathit{sId}), \mathit{R1.op1}(\mathit{sId}), \mathit{R1.op2}(\mathit{sId})\}$ where $\mathit{op1}$, and $\mathit{op2}$ are possible operations on a resource.

Using the indexed approach described in Section 4, for each requirement a parametric enforcement monitor can be derived from the requirement modeled as a P-TAV formalizing the requirement. One monitor instance is associated to each service instance.

5.2 Robust Mail Servers

Context. Many protocols (e.g., Simple Mail Transfer Protocol, SMTP) are used by email clients and servers to send and relay messages. After connecting to a server, a client should provide a MAIL_FROM message with sender's address as argument, and an RCPT_TO message with receiver's address as argument. The server responds with an OK_250 message if the addresses are valid. The client waits for the server response to transmit data. **The spam issue.** A high volume of spam messages can cause a Denial of Service (DoS). Slowing down SMTP conversations can refrain automated spam. Intentionally introducing delays between messages in a network is known as tarpitting (Hunter et al. (2003)). Tarpitting reduces the spam sending rate and prevents servers from processing a large number of spams.

Leveraging runtime enforcement to protect mail servers. Runtime enforcement mechanisms can be used as tarpits on mail servers to protect them. The expressiveness of P-TAVs also allows to model dynamic tarpits, where the delay introduced between messages can be increased (or decreased) based on the observed pattern of messages sent by a client over time. When a client establishes a connection, an instance of enforcement monitor can be created on-the-fly, to monitor all the incoming/outgoing messages of that particular session.

We consider an architectural setting where enforcement monitors run on a server. The input events to the monitors are both from the client and the server. Monitors delay the response from the server according to the client's behavior.

Let us see some requirements and P-TAVs parameterized by a client identifier (id).

R1 *If the number of RCPT_TO messages from a client is greater than* $\mathit{max\_req}$*, then there should be a delay of at least del t.u. before responding an OK_250 to the client.*

Requirement R1is formalized by the P-TAV in Fig. 3a. The delay is computed dynamically and depends on the number of received $\mathit{rcpt\_to}(\mathit{id})$ messages. The delay del is defined as 0 if $\mathit{counter} < \mathit{max\_req}$ and $\mathrm{int}(\frac{\mathit{counter}}{\mathit{max\_req}})$ otherwise. Upon receiving an $\mathit{rcpt\_to}(\mathit{id})$ message, if $\mathit{counter} \leq \mathit{max\_req}$, the P-TAV moves from l_0 to l_1. Otherwise, the P-TAV goes to l_2, resetting the clock x. Upon receiving an $\mathit{ok\_250}(\mathit{id})$, the P-TAV moves from l_1 to l_0, or from l_2 to l_0 if $x \geq \mathit{del}$.

R2 *After processing an RCPT_TO message from a client, if the server response message is ERROR_550 (user unknown or not found), then there should be a delay of at least 10 t.u. before sending this reply message back to the client.*

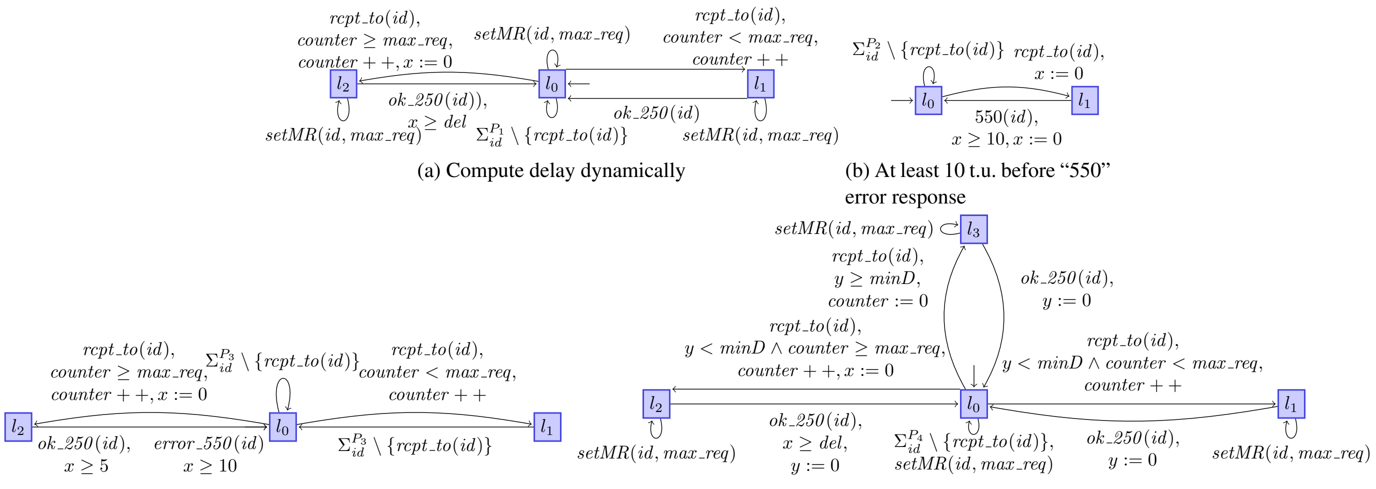
Requirement R2 is formalized by the P-TAV in Fig. 3b.

R3 *If the number of RCPT_TO messages is greater than* $\mathit{max\_req}$ *and the response of the server is OK_250 (resp. ERROR_550) then there should be a delay of at least 5 (resp. 10) t.u. before sending the response.*

R3 is formalized by the P-TAV in Fig. 3c.

R4 This requirement is formalized by the P-TAV in Fig. 3d. Compared to the P-TAV for R1, an additional feature is handled: *counter* is reset according to the computed delay, if there is sufficient time between two $\mathit{rcpt\_to}(\mathit{id})$ messages from the client.

The enforcement monitors synthesized from these requirements (modeled as P-TAVs), when integrated with the server can prevent the server from processing a large number of spam messages.



(a) Compute delay dynamically

(b) At least 10 t.u. before “550” error response

(c) Different delays based on the response message

(d) Decrease or Increase delay dynamically

Fig. 3. Robust Mail Servers

6. IMPLEMENTATION AND EVALUATION

We have implemented a prototype tool in Python based on the algorithms proposed in Pinisetty et al. (2012). Regarding performance, note that the tool is a prototype, implemented using open-source libraries and tools provided for quickly prototyping algorithms based on timed automata. The purpose of the tool is first to show feasibility and then to have a first assessment of performance. An enforcement monitor for a P-TAV instance is implemented with two concurrently running processes (Store and Dump). The Store process takes care of receiving input events, computing optimal delays of actions, and storing them in memory, and the Dump process deals with reading events stored in the memory and outputting them.

The tool inputs a property modeled with UPPAAL (Larsen et al. (1997)) and stored in XML. We use UPPAAL as a library to implement the update and post functions (see Pinisetty et al. (2012)), and the PyUPPAAL library to parse the properties. [4] Experiments were conducted on an Intel Core i7-2720QM (4 cores) at 2.20 GHz CPU, with 4 GB RAM, and running on Ubuntu 10.10. To illustrate our experiments, we use the P-TAV in Fig. 1b. In an indexed setting, we have an enforcement monitor per session, we need a front-end mechanism to slice the input trace, i.e., identify the monitor related to an input event, based on the value of the parameter.

Algorithm 1 FrontEndProcess

```
Enforcers ← {}
while tt do
    $(\delta, a(p, c))$ ← await($event$)
    $e$ ← Enforcers.get($p$)
    if ($e \neq$ NONE) then
        $e$.addEvent(($\delta, a(p, c)$))
    else
        $E$ = new Enforcer()
        $E$.addEvent(($\delta, a(p, c)$))
        Enforcers.add($p, E$)
    end if
end while
```

Enforcement monitors can be created and deleted on-the-fly. As explained in Sec. 4, some mechanism is needed to split the input trace based on parameter values. For slicing as in Chen and Rosu (2009) but based on a single parameter, the simple procedure is described in Algorithm 1.

We use an initially-empty hash-map (Enforcers) to keep track of active enforcers. The “get” method takes the key as input, and returns the associated enforcer, if present, and “NONE” otherwise. The “addEvent” method adds an input event to the input queue of the enforcer e. The “add” method adds the given key-value pair to the hash-map. Algorithm 1 was implemented in Python. The most expensive statement is the call to the “get” method, used to search and retrieve the enforcer associated to a key value. We measured the average execution time of the “get” method: with 100 enforcers, and an input trace with equal number of events per enforcer, we obtained 3.5 ms over 10,000 calls. Results of performance analysis are presented in Table 1. We have performed benchmarks for several runs until the variation in the obtained values was negligible.

We performed experiments, varying the number of instances of P-TAVs (entry N). For example, $N = 10$ means that there are 10 instances, each differing only in the value of the parameter ranging from 1 to 10.

N	$\|tr\|$	t_EM	t_post
1	100	4.9	0.040
1	500	71	0.18
1	1000	305	0.38
10	100	1	0.0095
10	500	4.5	0.026
10	1000	19.3	0.043
20	100	0.8	0.0083
20	500	3.8	0.015
20	1000	10	0.023
30	100	0.7	0.079
30	500	3.1	0.012
30	1000	4.2	0.019

Table 1: Performance evaluation

The entry $|tr|$ denotes the length of the input timed trace, generated using a trace-generator. The entry t_EM denotes the total time (in seconds) required to process a given input trace (and to compute optimal delays). The entry t_post denotes the time (in seconds) taken for one call to the post function, upon receiving the last event of the input trace. The trace-generator balances the number of events sent to each P-TAV instance.

Analysis. From Table 1, for a fixed value of N, t_post increases as the length of the input trace increases. However, these values should be almost equal. This known undesirable behavior is due to the invocation of UPPAAL for realizing the post function in the current implementation. The input trace is represented as an automaton. After each event, the input trace grows (the underlying automaton is updated), and the computation by UPPAAL restarts from the initial state. From Table 1, we can

[4] Given an input state and event, the post function computes the state reached in the underlying timed transition system of a P-TAV.

also observe that, given a fixed length for the input traces, the total simulation time t_EM decreases when N increases because the number of enforcers (more concurrent processes) increases, and thus more events are treated concurrently.

7. RELATED WORK AND DISCUSSION

Our approach is by no means the first to consider parametric specifications in runtime monitoring and is inspired from some previous endeavors proposed in the runtime verification community. This paper extends a paper introducing runtime enforcement for (propositional) timed properties (see Pinisetty et al. (2012)). With more expressive specifications (with parameterized events, internal variables, and session parameters), we improve the practicality and illustrate the usefulness of runtime enforcement on application scenarios.

Handling parametric specifications in runtime monitoring. Most of the frameworks handling parametric specifications deal only with their verification. Note however that the Monitoring Oriented Programming (MOP) framework, through the so-called notion of handler, allows some form of runtime enforcement by executing an arbitrary piece of code on property deviation. More generally, the usual questions addressed in verification of parametric specifications include defining a suitable semantics (that differs from the usual ones of model-checking), and providing monitor-synthesis algorithms that generate runtime-efficient mechanisms.

P-TAVs are inspired from existing parametric formalisms for untimed properties. The idea of using a non-parametric specification formalism along with an indexing mechanism according to the values of parameters (aka the "plugin" approach) was first proposed by the MOP team in Chen and Rosu (2009). In term of indexing, our framework uses a subset of the possibilities described in Chen and Rosu (2009) in that we use a totally-ordered set to index monitors instead of a partially-ordered set on bindings (i.e., partial functions from parameter names to their domains). Note however that our approach can be extended to multiple parameters using indexing mechanisms. Parameters should get bound on the first events and all events should carry values for all parameters (to ease the retrieval of the associated monitors). Monitoring algorithms could remain equivalent in terms of efficiency as monitor instances will be accessible with one index. The restriction considered in this paper allows to simplify the issue of indexing and determining the compatible monitors, given an input event. The acceptable expressiveness features related to parameters, their slicing mechanisms, and their overhead, are certainly application-dependent. Note also that MOP is restricted to propositional formalisms to specify (indexed) plugin monitors while P-TAV instances handle variables, guards, and assignments.

Distinguishing parameters from external variables (parameters yield new instances of monitors while external variables get rebound) was first introduced in Barringer et al. (2012) with Quantified Event Automata (QEAs). QEAs feature a general use of quantifiers over parameters but do not consider time.

Timed specification in monitoring techniques. Two usages of timed specifications have been proposed for monitoring purposes: i) rather theoretical efforts aiming at synthesizing monitors, and ii) tools for runtime monitoring of timed properties. In the first category, one seeks to obtain an operational mechanism (close to a timed automaton) from a declarative description of a property expressed in a timed logic such as TLTL$_3$ in Bauer et al. (2011) or MTL in Nickovic and Piterman (2010), Thati and Rosu (2005) and Basin et al. (2011). In the second category, the research efforts are tool-oriented. The Analog Monitoring Tool (Nickovic and Maler (2007)) monitors specifications written in STL/PSL over continuous signals. LARVA Colombo et al. (2009) takes as input properties expressed as DATEs (Dynamic Automata with Timers and Events) which basically resemble timed automata with stop watches but also feature resets, pauses, and can be composed into networks.

REFERENCES

Andrade, W.L., Machado, P.D.L., Jéron, T., and Marchand, H. (2011). Abstracting time and data for conformance testing of real-time systems. In *IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops*, 9–17.

Barringer, H., Falcone, Y., Havelund, K., Reger, G., and Rydeheard, D.E. (2012). Quantified event automata: Towards expressive and efficient runtime monitors. In *18th International Symposium on Formal Methods*, volume 7436, 68–84.

Basin, D., Klaedtke, F., and Zalinescu, E. (2011). Algorithms for monitoring real-time properties. In *2nd Int. Conf. on Runtime Verification*, volume 7186 of *LNCS*, 260–275.

Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, (4), 14:1–14:64.

Chen, F. and Rosu, G. (2009). Parametric trace slicing and monitoring. In *15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, 246–261.

Colombo, C., Pace, G.J., and Schneider, G. (2009). LARVA — safer monitoring of real-time Java programs. In *7th IEEE Int. Conf. on Software Engineering and Formal Methods*, 33–37.

Falcone, Y. (2010). You should better enforce than verify. In *1st Int. conference on Runtime verification*, volume 6418 of *LNCS*, 89–105.

Falcone, Y., Mounier, L., Fernandez, J.C., and Richier, J.L. (2011). Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3), 223–262.

Fradet, P. and Hong Tuan Ha, S. (2010). Aspects of availability. *Sci. Comput. Program.*, (7), 516–542.

Hunter, T., Terry, P., and Judge, A. (2003). Distributed tarpitting: Impeding spam across multiple servers. In *17th Large Installation Systems Administration*, USENIX.

Larsen, K.G., Pettersson, P., and Yi, W. (1997). UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1, 134–152.

Nickovic, D. and Piterman, N. (2010). From MTL to deterministic timed automata. In *8th Int. Conf. on Formal Modelling and Analysis of Timed Systems*, volume 6246, 152–167.

Nickovic, D. and Maler, O. (2007). AMT: a property-based monitoring tool for analog systems. In *5th Int. Conf. on Formal modeling and analysis of timed systems*, volume 4763 of *LNCS*, 304–319.

Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., and Timo, O.L.N. (2012). Runtime enforcement of timed properties. In *3rd Int. Conf. on Runtime Verification*, volume 7687 of *LNCS*. Springer.

Rusu, V., Bousquet, L.D., and Jéron, T. (2000). An approach to symbolic test generation. In *Proc. Integrated Formal Methods*, 338–357.

Thati, P. and Rosu, G. (2005). Monitoring algorithms for metric temporal logic specifications. *Electron. Notes Theor. Comput. Sci.*, 145–162.