Agence Nationale de la Recherche

Projet VACSIM

Sous Projet SP4

Livrable L4.2

Etudes de techniques de validation à l'exécution pour des systèmes réactifs critiques.

Synthèse

Le projet VACSIM (Validation de la commande des systèmes critiques par couplage simulation et méthodes d'analyse formelle), référencé ANR-11-INSE-004, étudie les avantages respectifs des techniques de simulation, en incluant des modèles des processus commandés, et des méthodes d'analyse formelles, pour la validation de la commande des systèmes critiques. Ce projet est structuré en 6 tâches.

La tâche 4 "Validation formelle de propriétés quantitatives : approche par automates" a pour but de contribuer à l'avancée de techniques de validation de systèmes temporisés. Elle est divisée en trois sous-tâches complémentaires, visant chacune une problématique particulière de validation : l'analyse quantitative des automates temporisés, les techniques de validation à l'exécution (test, monitoring et enforcement), et la vérification d'automates temporisés communicants. Ce livrable L4.2 du projet VACSIM est issu des travaux de la sous-tâche T4.2 relative à la validation à l'exécution de systèmes temporisés.

Dans ce livrable, nous proposons un nouveau cadre théorique permettant de synthétiser un enforceur pour des propriétés de safety et de co-safety décrites par des automates temporisés équipés d'états "acceptants" et "refusants" (souvent appelés "Good" et "Bad"). Nous avons étudié différents niveaux d'abstraction : la fonction d'enforcement, le moniteur d'enforcement, et enfin l'algorithme d'enforcement. Ce travail a été validé à l'aide d'un prototype. Ces résultats ont été publiés dans [PFJ+12].

Ce livrable a été initialement rédigé par le LaBRI et INRIA Rennes-Bretagne Atlantique.

Table des matières

1 Introduction

Beaucoup de techniques de vérification et de test sont basées sur le fait qu'une spécification formelle du système est disponible. Cependant, dans la réalité, il existe de nombreuses situations où ça n'est pas le cas, notamment dans le cas d'applications développées il y a longtemps. Le monitoring et l'enforcement à l'exécution sont des techniques qui permettent respectivement de vérifier et d'assurer certaines propriétés souhaitées sur le système analysé, et ce au cours de son exécution. Comme ces techniques focalisent sur des propriétés particulières en des points spécifiques, la spécification complète du système n'est pas nécessaire.

Les techniques de monitoring et d'enforcement ont été étudiées depuis longtemps dans un cadre non temporisé, mais à notre connaissance, aucun cadre théorique n'a été présenté pour enforcer des systèmes où le temps doit être pris en compte. Nous avons proposé dans [PFJ+12] de nouveaux fondements pour l'enforcement temporisé, en montrant comment synthétiser des enforceurs à l'exécution pour des propriétés de safety et de co-safety décrites par des automates temporisés. Nous avons adapté les notions de correction et de transparence de la théorie de l'enforcement dans un cadre temporisé. Dans notre travail, les enforceurs à l'exécution sont "time-retardant", i.e. il peuvent retarder la diffusion d'un événement. Les délais entre actions sont modifiés si besoin par le moniteur d'enforcement en utilisant un mémoire interne dans laquelle les (suites d') événements sont stockés puis relâchés après le délai approprié permettant d'assurer (dans la mesure du possible) la propriété souhaitée. Nous avons étudié différents niveaux d'abstraction : la fonction d'enforcement, le moniteur d'enforcement, et enfin l'algorithme d'enforcement. Ce travail a été validé à l'aide d'une implémentation prototype.

2 Enforcement dans un contexte temporisé

Notations préliminaires Soit $\mathbb{R}_{\geq 0}$ l'ensemble des réels positifs, et Σ un alphabet fini d'*actions*. Une paire $(\delta, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$ est appelée un *événement*. On note $\text{del}(\delta, a) = \delta$ et $\text{act}(\delta, a) = a$ les projections des événements sur les délais et les actions respectivement. Un *mot temporisé* sur Σ est une séquence finie d'événements appartenant à $(\mathbb{R}_{\geq 0} \times \Sigma)^*$. Pour $\sigma = (\delta_1, a_1) \cdot (\delta_2, a_2) \cdots (\delta_n, a_n)$, δ_i $(2 \leq i \leq n)$ est le délai entre a_{i-1} et a_i et δ_1 le temps écoulé avant la première action.

Dans notre framework, un moniteur d'enforcement (ME) peut être vu comme une boîte qui reçoit une séquence d'entrée (potentiellement erronée) σ, et envoyant une séquence de sortie o certifiée correcte par rapport à une propriété φ. Il réalise une fonction de transformation E prenant en paramètre la séquence d'entrée, et renvoyant une séquence de sortie. Mais comme nous sommes dans un contexte temporisé, le temps entre les événements doit être pris en compte dans les séquences et a évidemment dans le modèle ou le langage utilisé pour décrire les propriétés souhaitées. Ainsi, les séquences d'entrée et de sortie sont décrites par des mots temporisés. Par ailleurs, ce procédé est fait durant l'exécution, et ainsi le temps courant est nécessaire pour décider quelles actions devraient avoir été relâchées ou non par le moniteur d'enforcement. Ainsi, la fonction d'enforcement E est une fonction de $(\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0}$ vers $(\mathbb{R}_{\geq 0} \times \Sigma)^*$ pour une propriété donnée φ. La séquence retournée correspond à la séquence de sortie du système à un instant précis. La figure 1 illustre ce point.

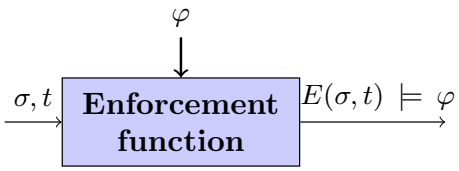


FIGURE 1 – Fonction d'enforcement E

De façon usuelle, la fonction d'enforcement doit respecter des règles de correction, transparence et d'optimalité. Mais dans notre cas ces règles incluent aussi des aspects temporels et dépendent du type de propriété que l'on souhaite assurer. Ce point sera discuté par la suite.

Comme nous autorisons le ME uniquement à retenir certaines actions pour assurer des propriétés, il a besoin d'une *Mémoire* permettant de les conserver. De façon similaire au cas non temporisé, à tout moment le ME a la possibilité de stocker une action reçue dans sa mémoire (opération *Store*), ou de relâcher la première action de la mémoire (opération *Dump*). Des règles additionnelles peuvent éventuellement être utilisées dans certaines situations, e.g ; lorsque la propriété requise sera toujours satisfaite (opération *Off*), où lorsque la propriété ne peut plus être satisfaite (opération (*Halt*). Dans un contexte temporisé, le moment où ces opérations doivent être appliquées doit être pris en compte.

Propriétés de Safety et de co-safety dans un contexte temporisé Une façon usuelle de définir une propriété consiste à utiliser un automate avec des états "acceptants" qui identifient les comportements gagnants. Il est aussi possible d'utiliser e.g. une logique temporelle comme LTL de manière équivalente. Maintenant, nous souhaitons décrire des propriétés du style : "le message a ne devrait pas arriver plus de deux fois par minute", ou "l'occurrence do b doit avoir lieu avant n unités de temps". Nous utilisons des automates temporisés (TA) [AD94] équipés de localités acceptantes pour décrire les propriétés à enforcer. La définition de TA que nous utilisons est similaire à celle de [AD94] dans laquelle certaines localités sont identifiées comme acceptantes. Nous rappelons ces définitions (un lecteur expert pourra passer cette partie, et revenir éventuellement en cas de doute sur une notation).

Soit $X = \{X_1, \ldots, X_k\}$ un ensemble fini d' *horloges*. Une *valuation d'horloge* pour X est une fonction ν de X vers $\mathbb{R}^X_{\geq 0}$ où $\mathbb{R}^X_{\geq 0}$ dénote les valuations de X. Pour $\nu \in \mathbb{R}^X_{\geq 0}$ et $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ est la valuation qui affecte $\nu(X_i) + \delta$ à chaque horloge X_i de X. Soit un ensemble donné d'horloges $X' \subseteq X$, $\nu[X' \leftarrow 0]$ est la valuation d'horloge ν où toutes les horloges de X' sont affectées à 0. $\mathcal{G}(X)$ dénote l'ensemble des contraintes d'horloge défini comme des combinaisons booléennes de contraintes simples de la forme $X_i \bowtie c$ avec $X_i \in X$, $c \in \mathbb{N}$ et $\bowtie \in \{<, \leq, =, \geq, >\}$. Soit $g \in \mathcal{G}(X)$ et $\nu \in \mathbb{R}^X_{\geq 0}$, on écrit $\nu \models g$ quand $g(\nu) \equiv true$.

Definition 1 (Automate Temporisé)**.** *Un* Automate Temporisé *(TA) est un n-uplet* $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$*, t.q.* L *est un ensemble fini de* localités *avec* $l_0 \in L$ *la* localité initiale*,* X *est un semble fini d'horloges,* Σ *est un ensemble fini d'*événements*,* $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ *est la* relation de transition*, et* $G \subseteq L$ *est l'ensemble des localités acceptantes.*

La sémantique *d'un TA est un système de transition temporisé* $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$ *où* $Q = L \times \mathbb{R}^X_{\geq 0}$ *est l'ensemble (infini) d'*états*,* $q_0 = (l_0, \nu_0)$ *est l'état initial où* ν_0 *est la valuation qui assigne toutes les horloges à* 0*,* $F_G = G \times \mathbb{R}^X_{\geq 0}$ *est l'ensemble des états acceptants,* $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ *est l'ensemble des* labels *de transition, i.e., des paires composées d'un délai et d'une action. La relation de transition* $\rightarrow \subseteq Q \times \Gamma \times Q$ *est un ensemble de transitions de la forme* $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ *avec* $\nu' = (\nu + \delta)[Y \leftarrow 0]$ *quand il existe* $(l, g, a, Y, l') \in \Delta$ *t.q.* $\nu + \delta \models g$ *pour* $\delta \geq 0$*.*

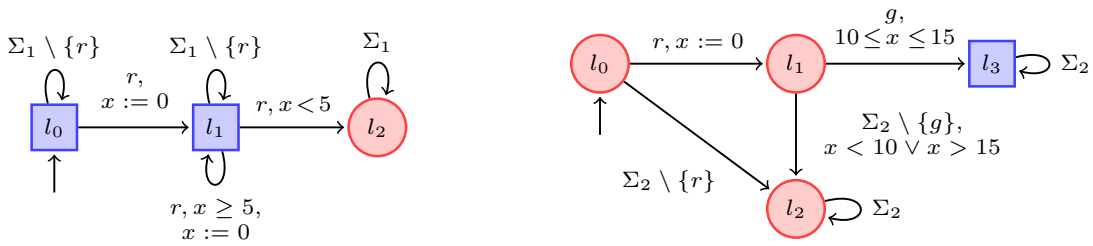
Nous ne manipulerons que des automates temporisés déterministes, avec la définition usuelle de [AD94].

Une propriété temporisée est définie par un langage temporisé $\varphi \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Soit un mot temporisé donné $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, nous disons que σ satisfait φ (noté $\sigma \models \varphi$) si $\sigma \in \varphi$. Dans notre cadre, nous nous intéressons uniquement aux propriétés de safety et de co-safety. De façon intuitive, un propriété de safety signifie que "rien de mauvais ne doit arriver" (usuellement caractérisé par un langage préfixe-clos), alors que la co-safety signifie que "quelque-chose de bon doit arriver" (usuellement caractérisé par un langage suffixe-clos). Dans notre cas, nous considérons les propriétés qui peuvent être décrites par un TA.

Definition 2 (TA de Safety et Co-safety)**.** *Un TA complet et déterministe* $\langle L, l_0, X, \Sigma, \Delta, G\rangle$*, où* $G \subseteq L$ *est l'ensemble des localités acceptantes, est dit :*

– *un TA de* safety *si* $\nexists \langle l, g, a, Y, l' \rangle \in \Delta, l \in L \setminus G \wedge l' \in G$;
– *un TA de* co-safety *si* $\nexists \langle l, g, a, Y, l' \rangle \in \Delta, l \in G \wedge l' \in L \setminus G$.

Les figures 2a et 2b montrent deux exemples de propriétés temporisées. Les localités acceptantes sont représentées par des carrés. La figure 2a est une propriété de safety sur $\Sigma_1 = \{a, r\}$ qui signifie "*Il doit y avoir un délai d'au moins 5 unités de temps entre deux requêtes (r) utilisateur*" ; La figure 2b est une propriété de co-safety sur $\Sigma_2 = \{a, g, r\}$ stipulant "*L'utilisateur peut effectuer une action a seulement après une authentification réussie, i.e. après avoir envoyé une requête r, et reçu l'accord g. Après r, g doit avoir lieu entre 10 et 15 unités de temps.*"



(a) Un TA de safety pour φ_1 (b) Un TA de co-safety pour φ_2

FIGURE 2 – Exemple de propriétés temporisées

Dans la partie suivante, nous donnons les définitions de correction, de transparence, et d'optimalité dans un contexte de safety temporisé, et nous fournissons la sémantique réalisant la fonction d'enforcement. Ensuite, nous appliquons le même procédé dans le cas de la co-safety.

3 Enforcement de propriétés de safety

Dans [PFJ+12] nous avons formellement défini les notions de correction, transparence, et d'optimalité avec du temps. Comme ces notions nécessitent un nombre important de notations, nous allons plutôt donner les idées principales. Nous considérons $E : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ une fonction d'enforcement pour une propriété de safety φ.

Correction La correction signifie qu'à tout moment t, la sortie produite doit satisfaire la propriété φ, i.e. $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}$, $E(\sigma, t) \models \varphi$.

Transparence La transparence permet de restreindre l'ensemble des possibilités d'enforcement du moniteur. Dans notre cas, la fonction d'enforcement ne devrait pas modifier l'ordre de événements, pas réduire le délai entre deux événements consécutifs, et pas produire de sortie plus rapidement que les entrées. La sortie $E(\sigma, t)$ devrait *retarder* la séquence d'entrée observée au temps t. "*Retard*" signifie ici que l'intervalle de temps entre deux actions de la séquence de sortie doit être supérieur ou égal à leur intervalle dans la séquence d'entrée ; et si on considère ω_i (resp. ω_o) la projection non temporisée de la séquence d'entrée (resp. de la sortie) au temps t, on doit avoir $\omega_o \preccurlyeq \omega_i$.

Optimalité De façon intuitive, l'optimalité signifie que la séquence de sortie observée devrait être "aussi proche que possible" de la séquence d'entrée. En fait, cette notion dépend du type de système analysé, et peut varier. Dans [PFJ+12], nous considérons deux aspects pour l'optimalité. Le premier est hérité des techniques d'enforcement non temporisé, et requiert qu'à tout instant t, la séquence de sortie $E(\sigma, t)$ soit le mot temporisé correct le plus long qui retarde la séquence d'entrée. Optimalité signifie aussi que le ME doit relâcher une action de sortie dès que possible ; i.e. tout préfixe de $E(\sigma, t)$ a le délai le plus petit possible.

Sémantique du moniteur d'enforcement Nous fournissons dans [PFJ+12] la sémantique formelle du ME qui réalise le fonction d'enforcement pour des propriétés de safety respectant les contraintes de correction, transparence et optimalité définies ci-dessus. C'est un système de transition avec des configurations de la forme $\langle \sigma_s, s, d, b, q \rangle$ où σ_s est l'état courant de la mémoire, s (resp. d) une valeur d'horloge exprimant le temps écoulé de la dernière opération store (resp. dump), b un booléen vrai tant que la propriété de safety est toujours satisfaite, et q l'état courant de $[\![\mathcal{A}]\!]$. Trois règles s'appliquent, par ordre de priorité :

- *Store :* lorsque l'enforceur reçoit un événement temporisé (δ, a), il stocke immédiatement dans sa mémoire l'événement (δ', a) où δ' est le délai minimal que l'on doit attendre pour que la propriété reste satisfaite, si cette valeur existe. s est remis à zéro.
- *Dump* : le délai du premier événement temporisé dans la mémoire correspond au temps qu'il faut attendre depuis la dernière action relâchée. Ainsi, la règle *dump* est exécutée quand d est égal à cette valeur. d est réinitialisé et le premier événement en mémoire est relâché et supprimé.

– *Elapse* : Le temps s'écoule et les valeurs de s et d augmentent.

La sémantique formelle est donnée ci-dessous. La fonction update_s calcule le délai minimal $\delta' \geq \delta$, tel que le TA de safety reste toujours dans une localité acceptante après avoir effectué l'action a (détaillé de façon formelle dans [PFJ+12]).

Definition 3 (Moniteur d'Enforcement pour safety)**.** *Un moniteur d'enforcement pour φ est un système de transitions $ME = \langle C, C_0, \Gamma_{ME}, \hookrightarrow \rangle$ t.q. :*

– $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B} \times Q$ *est l'ensemble des configurations ;*
– *la configuration initiale est* $C_0 = \langle \epsilon, 0, 0, \texttt{tt}, q_0 \rangle \in C$ *;*
– $\Gamma_{ME} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ *est l'alphabet entrée-opération-sortie, où* $Op = \{\text{store}(\cdot), \text{dump}(\cdot), \text{elapse}(\cdot)\}$ *;*
– $\hookrightarrow \subseteq C \times \Gamma_{ME} \times C$ *est la relation de transition définie comme la plus petite relation obtenue en appliquant les règles suivantes (dans cet ordre) :*
  – *store :* $\langle \sigma_s, \delta, d, \texttt{tt}, q \rangle \overset{(\delta,a)/\text{store}(\delta',a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta', a), 0, d, (\delta' \neq \infty), q' \rangle$ *avec :*
    – $\delta' = \text{update}_\text{s}(q, a, \delta)$,
    – q' *est défini comme* $q \overset{(\delta',a)}{\to} q'$ *si* $\delta' < \infty$ *et* $q' = q$ *sinon ;*
  – *dump :* $\big\langle (\delta, a) \cdot \sigma_s, s, \delta, b, q \big\rangle \overset{\epsilon/\text{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle \sigma_s, s, 0, b, q \rangle$ *si* $\delta \neq \infty$ *;*
  – *elapse :* $\langle \sigma_s, s, d, b, q \rangle \overset{\epsilon/\text{elapse}(\delta)/\epsilon}{\hookrightarrow} \langle \sigma_s, s + \delta, d + \delta, b, q \rangle$.

La figure 3a montre les règles successives appliquées à la propriété φ_1 représentée par le TA de la figure 2a. Elle montre l'évolution des configurations avec en entrée le mot temporisé $\sigma = (1, a) \cdot (3, r) \cdot (1, r)$. La figure 3b présente la fonction d'enforcement correspondante à tout moment de l'exécution.

Implémentation Nous avons présenté dans [PFJ+12] les algorithmes permettant d'implémenter le ME. L'implémentation est composée d'une mémoire en file d'attente, contenant des mots temporisés, et deux processus concurrents, un pour gérer la règle "store" (`StoreProcess`), et l'autre pour la règle "dump" (`DumpProcess`), comme illustré par la figure 4.

– l'Algorithme 1 correspond à `DumpProcess`. Quand la mémoire n'est pas vide, il s'intéresse au premier élément stocké (δ, a) et relâche a quand le temps depuis le dernier "dump" est égal à δ
– l'Algorithme 2 correspond à `StoreProcess`. Quand un événement d'entrée (δ, a) a lieu, le processus vérifie sur le TA de la propriété que l'état atteint est acceptant (fonction `post`), ce qui signifie que le propriété est satisfaite pour l'instant. Dans ce cas, (δ, a) est stocké dans la mémoire. Sinon, un nouveau délai δ' assurant que le TA de la propriété va atteindre un état acceptant de façon optimale est calculé (fonction `update`), et (δ', a) est stocké si δ' existe. Quand un événement est stocké, l'état symbolique courant est mis à jour sur le TA de la propriété.

$t \doteq 0 / (\epsilon, 0, 0, \texttt{tt}, < l_0, 0 >) / (1,a) \cdot (3,r) \cdot (1,r)$
$\downarrow$ elapse(1)
$t = 1 / (\epsilon, 1, 1, \texttt{tt}, < l_0, 1 >) / (1,a) \cdot (3,r) \cdot (1,r)$
$\downarrow$ Store
$t = 1 \epsilon / ((1,a), 0, 1, \texttt{tt}, < l_0, 1 >) / (3,r) \cdot (1,r)$
$\downarrow$ Dump
$t = 1 (1,a) / (\epsilon, 0, 0, \texttt{tt}, < l_0, 1 >) / (3,r) \cdot (1,r)$
$\downarrow$ elapse(3)
$t = 4 (1,a) / (\epsilon, 3, 3, \texttt{tt}, < l_0, 4 >) / (3,r) \cdot (1,r)$
$\downarrow$ Store
$t = 4 \ (1,a) / ((3,r), 0, 3, \texttt{tt}, < l_1, 0 >) / (1,r)$
$\downarrow$ Dump
$t = 4 (1,a) \cdot (3,r) / (\epsilon, 0, 0, \texttt{tt}, < l_1, 0 >) / (1,r)$
$\downarrow$ elapse(1)
$t = 5 (1,a) \cdot (3,r) / (\epsilon, 1, 1, \texttt{tt}, < l_1, 1 >) / (1,r)$
$\downarrow$ **Store**
$t = 5 (1,a) \cdot (3,r) / ((5,r), 0, 1, \texttt{tt}, < l_1, 0 >) / \epsilon$
$\downarrow$ elapse(4)
$t = 9 (1,a) \cdot (3,r) / ((5,r), 4, 5, \texttt{tt}, < l_1, 4 >) / \epsilon$
$\downarrow$ Dump
$t = 9 (1,a) \cdot (3,r) \cdot (5,r) / (\epsilon, 4, 0, \texttt{tt}, < l_1, 4 >) / \epsilon$

(a) Evolution de la configuration de l'enforceur

$t \doteq 0$	$\mathbf{E}(\sigma, \mathbf{t})$	**Op**	$\mathbf{obs}(\sigma, \mathbf{t})$
$t = 1$	ϵ	$\lvert$elapse(1)$\rvert$	ϵ
	$\downarrow$		
$t = 1$	ϵ	$\lvert store \rvert$	$(1,a)$
	$\downarrow$		
$t = 1$	$(1,a)$	$\lvert dump \rvert$	ϵ
	$\downarrow$		
$t = 4$	ϵ	$\lvert$elapse(3)$\rvert$	ϵ
	$\downarrow$		
$t = 4$	ϵ	$\lvert store \rvert$	$(3,r)$
	$\downarrow$		
$t = 4$	$(3,r)$	$\lvert dump \rvert$	ϵ
	$\downarrow$		
$t = 5$	ϵ	$\lvert$elapse(1)$\rvert$	ϵ
	$\downarrow$		
$t = 5$	ϵ	$\lvert store \rvert$	$(1,r)$
	$\downarrow$		
$t = 9$	ϵ	$\lvert$elapse(4)$\rvert$	ϵ
	$\downarrow$		
$t = 9$	$(5,r)$	$\lvert dump \rvert$	ϵ

(b) Fonction $d'enforcementE(\sigma, t)$ associée à ME

FIGURE 3 – Evolution de l'enforceur avec la séquence d'entrée $(1,a) \cdot (3,r) \cdot (1,r)$

Algorithm 1 DumpProcess

$d \leftarrow 0$
while tt **do**
  await $(|\sigma_s| \geq 1)$
  $(\delta, a) \leftarrow$ dequeue (σ_s)
  wait $(\delta - d)$
  dump (a)
  $d \leftarrow 0$
end while

Algorithm 2 StoreProcess

$(l, X) \leftarrow (l_0, [X \leftarrow 0])$
while tt **do**
  $(\delta, a) \leftarrow$ await *event*
  if $(\text{post}(l, X, a, \delta) \notin G)$ **then**
    $\delta' \leftarrow \text{update}(l, X, a, \delta)$
    if $\delta' = \infty$ **then**
      terminate StoreProcess
    end if
  else
    $\delta' \leftarrow \delta$
  end if
  $(l, X) \leftarrow \text{post}(l, a, X, \delta')$
  enqueue (δ', a)
end while

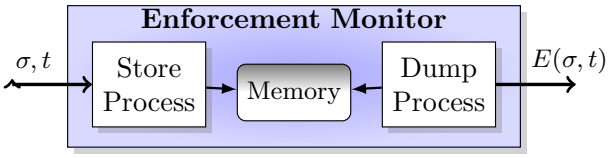


FIGURE 4 – Architecture de l'implémentation du ME

Un prototype basé sur ces algorithmes a été développé. Etant donné que le calcul de δ est assez fastidieux, il est déporté à la librairie Uppaal [LPY97], qui permet d'effectuer des calculs sur les TA de manière efficace. Une synthèse des résultats expérimentaux montrant la pertinence de l'approche est fournie dans [PFJ+12].

4 Enforcement de propriétés de co-safety

Dans [PFJ+12], nous avons utilisé le même état d'esprit pour enforcer des propriétés de co-safety. Une différence majeure entre les deux, c'est qu'une propriété de co-safety consiste à rester dans un état "Bad" jusqu'à ce qu'il devienne définitivement "Good" (extension clos). D'un point de vue enforcement, cela implique que le ME ne doit pas relâcher le moindre événement avant d'être sûr qu'un état gagnant sera atteint, et ce afin de respecter la contrainte de transparence. Nous avons adopté cette stratégie. Ceci peut donc induire un délai important entre la réception et la transmission d'un événement. D'un point de vue pratique, cela revient à adopter une stratégie "store and forward"[1]. D'un point de vue formel, cela implique aussi que la sortie vide (lorsque l'enforceur stocke les événements sans rien relâcher) satisfait toujours la condition de correction. Nous avons présenté dans [PFJ+12] le cadre complet pour l'enforcement de propriétés de co-safety. Dans ce résumé, nous allons plutôt nous concentrer sur les points communs et les différences entre les approches safety et co-safety.

Comme dans le cas précédent, nous considérons une propriété de co-safety φ décrite par un TA $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ et sa sémantique associée $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$. Une fonction d'enforcement E pour une propriété de co-safety φ doit satisfaire de nouvelles conditions de correction, transparence, et d'optimalité.

Correction, transparence, optimalité Dans le cas de la co-safety, si un mot temporisé est stocké, alors la sortie devrait satisfaire la propriété φ dans le futur. Par conséquent, le mot vide ϵ satisfait toujours la condition de correction. La transparence est similaire au cas de la safety (conserver l'ordre des événements, et ne pas réduire le délai entre deux événements), mais elle est exprimée en utilisant un temps futur. Cela est du au fait que le premier élément est relâché seulement lorsque l'état acceptant a été atteint par la séquence d'entrée. Pour la même raison, l'optimalité est exprimée avec un délai égal à la longueur (i.e. le somme des délais) de la séquence stockée en entier. Contrairement au cas de la safety, l'optimalité est calculée à partir du temps total de la séquence stockée, et correspond à la somme minimale des délais pour le plus petit préfixe qui satisfait la propriété. Pour le reste de la séquence (i.e. quand la propriété est satisfaite), la sortie est égale à l'entrée.

Moniteur d'enforcement Dans [PFJ+12], nous avons proposé une sémantique de ME basée sur cinq règles. Lorsqu'un événement (δ, a) est reçu et que la propriété n'est toujours pas satisfaite, alors (δ, a) est stockée dans la mémoire and les variables mises à jour (règle store-$\overline{\varphi}$). En revanche, si φ devient satisfaite, la règle store-φ_{init} s'applique : les délais en mémoire sont remplacés par les meilleures valeurs possibles. Cette règle est appliquée une seule fois. A partir de ce moment, la propriété sera satisfaite, et la règle store-φ est appliquée lorsqu'un événement est reçu, stockant simplement (δ, a) en mémoire. Les règles "dump" et "elapse" sont similaires au cas de la safety. Un exemple d'évolution des

1. Expression héritée du domaine réseaux : par exemple, un commutateur "store and forward" conserve une trame reçue en mémoire, et commence la transmission seulement après réception complète

configurations est donné figure 5 pour la propriété φ_2 de la figure 2b avec la séquence d'entrée $(1,r)\cdot(8,g)\cdot(5,a)$.

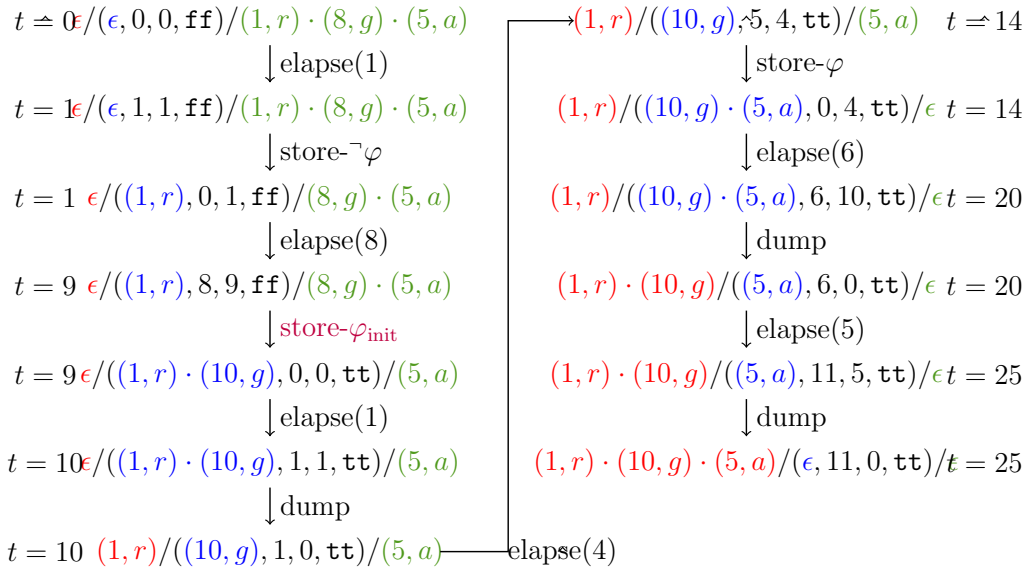


FIGURE 5 – Evolution de l'enforceur avec la séquence d'entrée $(1,r)\cdot(8,g)\cdot(5,a)$

Implémentation L'implémentation consiste encore en une mémoire et deux processus concurrents. `StoreProcess` applique maintenant deux stratégies selon le fait que l'état acceptant a été atteint ou non. Lorsque l'événement reçu permet d'atteindre une localité acceptante, les délais optimisés sont calculés (fonction `update`) et une notification est envoyée à `DumpProcess` qui commence alors à relâcher des actions. Le nouveau `StoreProcess` est décrit dans l'algorithme 3. `DumpProcess` reste inchangé.

Nous avons présenté dans [PFJ+12] un outil prototype qui implémente un enforceur dans le cas de propriétés de safety, mais aussi de co-safety.

Algorithm 3 StoreProcess

```
goalReached ← ff
while tt do
  (δ, a) ← await (event)
  enqueue(δ, a)
  if goalReached = ff then
    (newDelays, R) ← update_c(σ_s)
    if R = tt then
      modify delays
      goalReached ← tt
      notify (startDump)
    end if
  end if
end while
```

Références

[AB06] Eugene Asarin and Patricia Bouyer, editors. *Formal Modeling and Analysis of Timed Systems, 4th International Conference, FORMATS 2006, Paris, France, September 25-27, 2006, Proceedings*, volume 4202 of *LNCS*. Springer, 2006.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[BF09] Manuela L. Bujorianu and Michael Fisher, editors. *Proceedings FM-09 Workshop on Formal Methods for Aerospace*, volume 20 of *EPTCS*, 2009.

[BFF+10] Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors. *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*. Springer, 2010.

[BFH+12] Howard Barringer, YliÃÍs Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified Event Automata : Towards Expressive and Efficient Runtime Monitors. In *FM 2012 : 18th International symposium on Formal Methods*, 2012. Accepted for publication. To appear.

[BGHS09] Howard Barringer, Alex Groce, Klaus Havelund, and Margaret H. Smith. An entry point for formal methods : Specification and analysis of event logs. In Bujorianu and Fisher [BF09], pages 16–21.

[BKZ11] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Algorithms for monitoring real-time properties. In Khurshid and Sen [KS12], pages 260–275.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4) :14, 2011.

[CF09] Darren D. Cofer and Alessandro Fantechi, editors. *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, volume 5596 of *LNCS*. Springer, 2009.

[CH10] Krishnendu Chatterjee and Thomas A. Henzinger, editors. *Formal Modeling and Analysis of Timed Systems - 8th International Conference, FORMATS 2010, Klosterneuburg, Austria, September 8-10, 2010. Proceedings*, volume 6246 of *LNCS*. Springer, 2010.

[CPA10] Christian Colombo, Gordon J. Pace, and Patrick Abela. Compensation-aware runtime monitoring. In Barringer et al. [BFF+10], pages 214–228.

[CPS08] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *FMICS*, pages 135–149, 2008.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In *SEFM*, pages 33–37, 2009.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Safe runtime verification of real-time properties. In *Formal Modeling and Analysis of Timed Systems, 7th International Conference (FORMATS)*, volume 5813 of *LNCS*, pages 103–117, Budapest, Hungary, 2009.

[CR09] Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Kowalewski and Philippou [KP09], pages 246–261.

[Fal10] Yliès Falcone. You should better enforce than verify. In *Runtime Verification*, pages 89–105, 2010.

[FFM12] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime ? *STTT*, 14(3) :349–382, 2012.

[HK09] Dang Van Hung and Padmanabhan Krishnan, editors. *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*. IEEE Computer Society, 2009.

[HLMN10] John Havlicek, Scott Little, Oded Maler, and Dejan Nickovic. Property-based monitoring of analog and mixed-signal systems. In Chatterjee and Henzinger [CH10], pages 23–24.

[KP09] Stefan Kowalewski and Anna Philippou, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *LNCS*. Springer, 2009.

[KS12] Sarfraz Khurshid and Koushik Sen, editors. *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *LNCS*. Springer, 2012.

[KVK+04] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac : A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2) :129–155, 2004.

[LBW09] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transaction Information System Security.*, 12(3), 2009.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2) :134–152, 1997.

[LY04] Yassine Lakhnech and Sergio Yovine, editors. *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *LNCS*. Springer, 2004.

[Mat07] Ilaria Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electron. Notes Theor. Comput. Sci.*, 186 :101–120, July 2007.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In Lakhnech and Yovine [LY04], pages 152–166.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In Asarin and Bouyer [AB06], pages 274–289.

[NM07] Dejan Nickovic and Oded Maler. AMT : A property-based monitoring tool for analog systems. In *Formal Modeling and Analysis of Timed Systems*, pages 304–319, 2007.

[NP10] Dejan Nickovic and Nir Piterman. From MTL to deterministic timed automata. In Chatterjee and Henzinger [CH10], pages 152–167.

[PGMN10] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot : A hard real-time runtime monitor. In *Proceedings of the 1st Intl. Conference on Runtime Verification*, LNCS. Springer, November 2010.

[PNW11] Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the 2nd Intl. Conference on Runtime Verification*, LNCS. Springer, September 2011.

[RT07] Jean-François Raskin and P. S. Thiagarajan, editors. *Formal Modeling and Analysis of Timed Systems, 5th International Conference, FORMATS 2007, Salzburg, Austria, October 3-5, 2007, Proceedings*, volume 4763 of *LNCS*. Springer, 2007.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1), 2000.

[TR05] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electr. Notes Theor. Comput. Sci.*, 113 :145–162, 2005.

Annexes

[PFJ+12] Srinivas Pinisetty, Ylies Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena-Timo. Runtime enforcement of timed properties. In *Third International Conference on Runtime Verification RV 2012*, volume 7687 of *Lecture Notes in Computer Science*, pages 229–244, Istanbul, Turkey, September 2012. Springer-Verlag.

Runtime Enforcement of Timed Properties

Srinivas Pinisetty[1], Yliès Falcone[2], Thierry Jéron[1], Hervé Marchand[1], Antoine Rollet[3] and Omer Nguena Timo[4]

[1] INRIA Rennes - Bretagne Atlantique, France First.Last@inria.fr
[2] LIG, Université Grenoble I, France Ylies.Falcone@ujf-grenoble.fr
[3] LaBRI, Université de Bordeaux - CNRS, France Antoine.Rollet@labri.fr
[4] IRIT, France Omerlandry.Nguenatimo@enseeiht.fr

Abstract. Runtime enforcement is a powerful technique to ensure that a running system respects some desired properties. Using an enforcement monitor, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies to a property. Runtime enforcement has been extensively studied over the last decade in the context of untimed properties.
This paper introduces runtime enforcement of timed properties. We revisit the foundations of runtime enforcement when time between events matters. We show how runtime enforcers can be synthesized for any safety or co-safety timed property. Proposed runtime enforcers are time retardant: to produce an output sequence, additional delays are introduced between the events of the input sequence to correct it. Runtime enforcers have been prototyped and our simulation experiments validate their effectiveness.

1 Introduction

Runtime verification [1–6] (resp. enforcement [7–9]) refers to the theories, techniques, and tools aiming at checking (resp. ensuring) the conformance of the executions of systems under scrutiny w.r.t. some desired property. The first step of those monitoring approaches consists in instrumenting the underlying system so as to partially observe the events or the parts of its global state that may influence the property under scrutiny. A central concept is the verification or enforcement *monitor* that is generally synthesized from the property expressed in a high-level formalism. Then, the monitor can operate either *online* by receiving events in a lock-step manner with the execution of the system or *offline* by reading a log of system events. When the monitor is only dedicated to verification, it is a decision procedure emitting verdicts stating the correctness of the (partial) observed trace generated from the system execution.

Three categories of runtime verification frameworks can be distinguished according to the formalism used to express the input property. In *propositional* approaches, properties refer to events taken from a finite set of propositional names. For instance, a propositional specification may rule the ordering of function calls in a program. Monitoring such kind of specifications has received a lot of attention. *Parametric* approaches have received a growing interest in the last five years. Here, events are augmented with formal parameters, instantiated at runtime. In *timed* approaches, the observed time between events may influence the truth-value of the property. It turns out that monitoring of (continuous) time specifications is a much harder problem. Intuitively, when monitoring a timed specification, the problem that arises is that the overhead induced by the monitor (i.e., the time spent executing monitor's code) influences the truth-value of the monitored specification. Consequently, not much information can be gained from the

verdicts produced by the monitor. Few attempts have been made on monitoring systems w.r.t. timed properties (see Sec. 8 for related work). Two lines of work can be distinguished: synthesis of automata-based decision procedures for timed formalisms (e.g., [1, 3–5]), and, tools for runtime verification of timed properties [10, 11].

In runtime enforcement, an enforcement monitor (EM) is used to transform some (possibly) incorrect execution sequence into a correct sequence w.r.t. the property of interest. In the propositional case, the transformation performed by an EM should be *sound* and *transparent*. Soundness means that the resulting sequence obeys the property. Transparency means that, if the input sequence already conforms to the property, the monitor has to modify it in a minimal way. According to how a monitor is allowed to modify the input sequence (i.e., the primitives afforded to the monitor), several models of enforcement monitors have been proposed [7–9]. In a nutshell, an EM can definitely block the input sequence (as done by security automata), suppress an event from the input sequence (as done by suppression automata), insert an event to the input sequence (as done by insertion automata), or perform any of these primitives (as is the case with edit-automata). Moreover, according to how transparency is effectively formalized, several definitions of runtime enforcement have been proposed (see [9] for an overview).

In this paper we focus on *online enforcement of timed properties*. To the best of our knowledge, no approach was proposed to enforce timed properties. Motivations for extending runtime enforcement to timed properties abound. First, timed properties are a more precise tool to specify desired behaviors of systems since they allow to explicitly state how time should elapse between two events. Moreover, several applications of runtime enforcement of timed properties can be considered. For instance, in the context of security monitoring, enforcement monitors can be used as firewalls to prevent denial of service attacks by ensuring a minimal delay between input events (carrying some request for a protected server). On a network, enforcement monitors can be used to synchronize streams of events together, or, ensuring that a stream of events conforms to the pre-conditions of some service.

Contributions. We propose a context where, under some reasonable assumptions, runtime enforcement of timed properties is possible. For this purpose, we adapt soundness and transparency to a timed context. Runtime enforcement monitors are built from safety and co-safety properties expressed by timed automata. In contrast with previous runtime enforcement approaches, we afford only the primitives of being able to delay the input events to our enforcer. By possibly increasing delays between events of the input sequence, the output timed sequence conforms to the property. Delays are modified by monitors using an internal memory where (sequence of) events are stored and released after appropriate delays. Experiments have been performed on prototype monitors to show their effectiveness and the feasibility of our approach.

Paper organization. Section 2 introduces preliminaries and notation. Section 3 introduces the notion of enforcement for timed properties. Sections 4 and 5 describe how one can enforce safety and co-safety properties, respectively. Our prototype implementations of monitors and experiments are in Sec. 6 and Sec. 7, respectively. Section 8 discusses related work. Finally, conclusions and open perspectives are drawn in Sec. 9.

2 Preliminaries and Notation

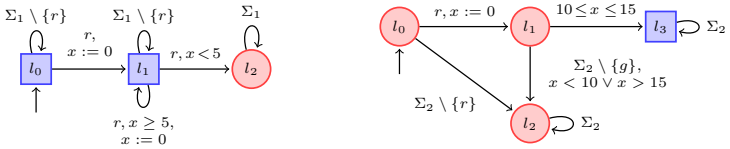
Untimed notions. An alphabet is a finite set of elements. A (finite) word over an alphabet A is a finite sequence of elements of A. The *length* of a word w is noted $|w|$. The empty word over A is denoted by ϵ_A or ϵ when clear from context.The set of all (resp. non-empty) words over A is denoted by A^* (resp. A^+). A *language* over A is a subset $\mathcal{L} \subseteq A^*$. The concatenation of two words w and w' is noted $w \cdot w'$. For an interval $[j,k]$ in $\mathbb{N}$, by $\bigodot_{i\in[j,k]}(a_i)$ we denote the concatenation $a_j \cdot a_{j+1} \cdots a_k$. A word w' is a prefix of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$. For a word w and $1 \leq i \leq |w|$, the i-th letter (resp. prefix of length i, suffix starting at position i) of w is noted $w(i)$ (resp. $w_{[\cdots i]}$, $w_{[i\cdots]}$) – with the convention $w_{[\cdots 0]} \stackrel{\text{def}}{=} \epsilon$. $\mathrm{pref}(w)$ denotes the set of prefixes of w and by extension, $\mathrm{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \{\mathrm{pref}(w) \mid w \in \mathcal{L}\}$ the prefix of $\mathcal{L}$. $\mathcal{L}$ is said to be *prefix-closed* whenever $\mathrm{pref}(\mathcal{L}) = \mathcal{L}$ and *extension-closed* whenever $\mathcal{L} = \mathcal{L} \cdot A^*$. Given a tuple of symbols $e = (e_1, \ldots, e_n)$, $\Pi_i(e)$ is the projection of e on its i^{th} element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$).

Timed languages. Let $\mathbb{R}_{\geq 0}$ denote the set of non negative real numbers, and Σ a finite alphabet of *actions*. A pair $(\delta, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$ is called an *event*. We note $\mathrm{del}(\delta, a) = \delta$ and $\mathrm{act}(\delta, a) = a$ the projections of events on delays and actions, respectively. A *timed word* over Σ is a finite sequence of events ranging over $(\mathbb{R}_{\geq 0} \times \Sigma)^*$. For $\sigma = (\delta_1, a_1)\cdot(\delta_2, a_2) \cdots (\delta_n, a_n)$, δ_i $(2 \leq i \leq n)$ is the delay between a_{i-1} and a_i and δ_1 the time elapsed before the first action. Note that the alphabet is infinite in this case. Nevertheless, previous notions and notations defined above (related to length, concatenation, prefix, etc) naturally extend to timed words. *The sum of delays* of a timed word σ is noted $\mathrm{time}(\sigma)$. Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we define the *observation of σ at time t* as the timed word $\mathrm{obs}(\sigma, t) \stackrel{\text{def}}{=} \max\{\sigma' \mid \sigma' \preccurlyeq \sigma \wedge \mathrm{time}(\sigma') \leq t\}$, i.e., the longest prefix of σ with a sum of delays less than t. The *untimed projection* of σ is $\Pi_\Sigma(\sigma) \stackrel{\text{def}}{=} a_1 \cdot a_2 \cdots a_n$ in Σ^* (i.e., delays are ignored). A *timed language* is any subset $\mathcal{L} \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. We define the following order on timed words: *σ' delays σ* (noted $\sigma' \preccurlyeq_d \sigma$) if $\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$ and $\forall i \leq |\sigma'|, \mathrm{del}(\sigma(i)) \leq \mathrm{del}(\sigma'(i))$.

Timed Automata. Let $X = \{X_1, \ldots, X_k\}$ be a finite set of *clocks*. A *clock valuation* for X is a function ν from X to $\mathbb{R}^X_{\geq 0}$ where $\mathbb{R}^X_{\geq 0}$ denotes the valuations of X. For $\nu \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ is the valuation assigning $\nu(X_i) + \delta$ to each clock X_i of X. Given a set of clocks $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation ν where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of clock constraints defined as boolean combinations of simple constraints of the form $X_i \bowtie c$ with $X_i \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and $\nu \in \mathbb{R}^X_{\geq 0}$, we write $\nu \models g$ when $g(\nu) \equiv true$.

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

The semantics *of a TA is a timed transition system $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$ where $Q = L \times \mathbb{R}^X_{\geq 0}$ is the (infinite) set of* states*, $q_0 = (l_0, \nu_0)$ is the initial state where ν_0*



(a) A safety TA for φ_1 (b) A co-safety TA for φ_2

Fig. 1: Example of Timed Properties

is the valuation that maps every clock to 0, $F_G = G \times \mathbb{R}^X_{\geq 0}$ *is the set of of accepting states,* $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ *is the set of transition* labels*, i.e., pairs composed of a delay and an action. The transition relation* $\rightarrow \subseteq Q \times \Gamma \times Q$ *is a set of transitions of the form* $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ *with* $\nu' = (\nu + \delta)[Y \leftarrow 0]$ *whenever there exists* $(l, g, a, Y, l') \in \Delta$ *s.t.* $\nu + \delta \models g$ *for* $\delta \geq 0$.

In the following, we consider a timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ with its semantics $[\![\mathcal{A}]\!]$. $\mathcal{A}$ is *deterministic* whenever for any (l, g_1, a, Y_1, l'_1) and (l, g_2, a, Y_2, l'_2) in Δ, $g_1 \wedge g_2$ is *false*. $\mathcal{A}$ is *complete* whenever for any location $l \in L$ and every event $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labeled by a is *true*. In the remainder of this paper, we shall consider only deterministic timed automata.

A *run* ρ from $q \in Q$ is a sequence of moves in $[\![\mathcal{A}]\!]$ of the form: $\rho = q_0 \xrightarrow{(\delta_1, a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n, a_n)} q_n$. The set of runs from $q_0 \in Q$ is denoted $Run(\mathcal{A})$ and $Run_{F_G}(\mathcal{A})$ denotes the subset of runs *accepted* by $\mathcal{A}$, i.e., ending in F_G. The *trace* of a run ρ is the timed word $(\delta_1, a_1) \cdot (\delta_2, a_2) \cdots (\delta_n, a_n)$. We note $\mathcal{L}(\mathcal{A})$ the set of traces of $Run(\mathcal{A})$. We extend this notation to $\mathcal{L}_{F_G}(\mathcal{A})$ in a natural way.

Timed Properties A timed property is defined by a timed language $\varphi \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Given a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$. In the sequel, we shall be interested in safety and co-safety timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). Safety (resp. co-safety) properties can be characterized by prefix-closed (resp. extension-closed) languages. We consider only the sets of safety and co-safety properties that can be represented by timed automata (Definition 1).

Definition 2 (Safety and Co-safety TA). *A complete and deterministic TA* $\langle L, l_0, X, \Sigma, \Delta, G \rangle$*, where* $G \subseteq L$ *is the set of accepting locations, is said to be:*

- *a* safety *TA if* $\nexists \langle l, g, a, Y, l' \rangle \in \Delta, l \in L \setminus G \wedge l' \in G$*;*
- *a* co-safety *TA if* $\nexists \langle l, g, a, Y, l' \rangle \in \Delta, l \in G \wedge l' \in L \setminus G$.

It is easy to check that safety and co-safety TAs define safety and co-safety properties.

Example 1 (Safety and co-safety TA). Fig. 1a and 1b present two properties formalized with safety and co-safety TA. Accepting locations are represented by squares. The safety TA formalizes the property φ_1 defined over $\Sigma_1 = \{a, r\}$: *"There should be a delay of at least 5 time units between any two user requests (r)"*. The co-safety TA formalizes the property φ_2 defined over $\Sigma_2 = \{r, g, a\}$: *"The user can perform an action a only after a successful authentication, i.e., after sending a request r and receiving a grant g. After an r, g should occur between 10 and 15 time units"*.

3 Enforcement Monitoring in a Timed Context

Roughly speaking, both in the timed and untimed settings, the purpose of an enforcement monitor (EM) is to read some (possibly incorrect) input sequence σ produced by a running system (input to the enforcer), and to transform it into an output sequence o that is correct w.r.t. a property φ, here modeled by a TA. From an abstract point of view, an enforcement monitor realizes an enforcement function E that transforms timed words into timed words according to global time.

Definition 3. *For a given property φ, an* enforcement function *is a function E from $(\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0}$ to $(\mathbb{R}_{\geq 0} \times \Sigma)^*$.*

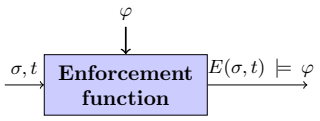


Fig. 2: Enforcement function E

An enforcement function E transforms some timed word σ given as input and possibly incorrect w.r.t. the desired property (see Fig. 2). The resulting output $E(\sigma, t)$ at time t is a timed word with same actions, but possibly increased delays between actions so that it satisfies the property. Similar to the untimed setting, additional constraints on $E(\sigma, t)$, namely *soundness* and *transparency*, are required on actions. However, in the timed setting, those constraints also depend on both delays between events and the class of the enforced property, as we shall discuss later.

An enforcement function E is realized by an *enforcement monitor* EM. This monitor is equipped with a memory and a set of enforcement operations used to store and dump some timed events to and from the memory, respectively. The memory of an EM is basically a queue containing a timed word, the received actions with increased delays that have not been released yet. In addition, the EM also keeps track of the state of the TA modeling the property, satisfaction of the property using a Boolean variable, and some variables indicating the clock values used to count time between input and output events.

The specific operations of the EM are the *Store* operation which stores in memory the received action together with a possibly modified delay; the *Dump* operation which releases the first action from the memory; and the optional *Halt* operation which stops the enforcer, i.e., blocks the input sequence and stops producing outputs. *Off* operation which turns off the enforcer. The *Off* and *Halt* operations can be added for optimization. The *Off* can be used when we observe that the property will be satisfied for any future input events. The *Halt* operation is useful if the property cannot be satisfied anymore.

In the following sections, we will present enforcement monitors for both safety and co-safety properties and analyze constraints on the associated enforcement functions.

4 Enforcement of Safety Properties

In this section we focus on the enforcement of a safety property φ specified by a safety automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ and its associated semantics $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$. Without loss of generality, we assume that the set of locations $L \setminus G$ is reduced to a singleton $\{Bad\}$. Given φ, and a timed word σ, an enforcement function E for φ should satisfy the following soundness, transparency and optimality conditions.

Definition 4 (Soundness, transparency and optimality). *Let $E : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}_{\geq 0} \times \Sigma)^*$ be an enforcement function for a safety property φ. E is:*
- sound *if $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}$, $E(\sigma, t) \models \varphi$;*

- transparent *if* $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}$, $E(\sigma,t) \preccurlyeq_d \mathrm{obs}(\sigma,t) \wedge \mathrm{time}(E(\sigma,t)) \leq t$.

If E is both sound and transparent, we say that it is optimal *if, for any input $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, at any time $t \in \mathbb{R}_{\geq 0}$, the following constraints hold:*

(Op1) $\nexists \omega', \omega' \models \varphi \ \wedge \ \omega' \preccurlyeq_d obs(\sigma,t) \ \wedge \ |\omega'| > |E(\sigma,t)|$

(Op2) $\forall i \in [1, |E(\sigma,t)|], \nexists \delta'' \in \mathbb{R}_{\geq 0}, \mathrm{del}(\mathrm{obs}(\sigma,t)(i)) \leq \delta'' \leq \mathrm{del}(E(\sigma,t)(i)) \wedge E(\sigma,t)_{[\cdots i-1]} \cdot (\delta'', \mathrm{act}(E(\sigma,t)(i))) \models \varphi$

Soundness means that, at any time t, the produced timed word should satisfy the property φ. Transparency means that, at any time instant t, the output $E(\sigma,t)$ delays the input $\mathrm{obs}(\sigma,t)$: the enforcement function should not modify the order of events, should not reduce the delays between consecutive events, and should not produce outputs faster than inputs. Optimality means that the enforcement function should provide the output as soon as possible. The optimality condition **(Op1)** extends the requirement on the output sequences of the enforcement function in the untimed case (cf. [9]): at any time instant t, the output sequence $E(\sigma,t)$ should be the longest correct timed word delaying the input sequence $\mathrm{obs}(\sigma,t)$. Here, taking physical time into account, **(Op2)** requires that the input and output sequences are as close as possible w.r.t. physical observation, i.e., every prefix of $E(\sigma,t)$ has the shortest possible last delay.

We now design an enforcement monitor whose semantics effectively realizes the enforcement function as described Definition 4.

Definition 5 (Enforcement Monitor for safety). *An enforcement monitor for φ is a transition system $EM = \langle C, C_0, \Gamma_{EM}, \hookrightarrow \rangle$ s.t.:*

- $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B} \times Q$ *is the set of configurations;*
- *the initial configuration is $C_0 = \langle \epsilon, 0, 0, \mathtt{tt}, q_0 \rangle \in C$;*
- $\Gamma_{EM} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ *is the input-operation-output alphabet, where $Op = \{\mathrm{store}(\cdot), \mathrm{dump}(\cdot), \mathrm{del}(\cdot)\}$;*
- $\hookrightarrow \subseteq C \times \Gamma_{EM} \times C$ *is the transition relation defined as the smallest relation obtained by the following rules applied in the following order:*
  - *store:* $\langle \sigma_s, \delta, d, \mathtt{tt}, q \rangle \overset{(\delta,a)/\mathrm{store}(\delta',a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta', a), 0, d, (\delta' \neq \infty), q' \rangle$ *with:*
    - $\delta' = \mathrm{update_s}(q, a, \delta)$, *where* $\mathrm{update_s}$[5] *is the function defined as:*

$$Q \times \Sigma \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$$
$$(q, a, \delta) \mapsto \begin{cases} \infty & \text{if } \forall \delta' \in \mathbb{R}_{\geq 0}, \forall q_1 \in Q, (\delta' \geq \delta \wedge q \overset{(\delta',a)}{\to} q_1) \Rightarrow q_1 \notin F_G \\ \min\{\delta' \in \mathbb{R}_{\geq 0} \mid \exists q_1 \in F_G, q \overset{(\delta',a)}{\to} q_1 \wedge \delta' \geq \delta\} \end{cases}$$

    - q' *is defined as* $q \overset{(\delta',a)}{\to} q'$ *if* $\delta' < \infty$ *and* $q' = q$ *otherwise;*
  - *dump:* $\big\langle (\delta, a) \cdot \sigma_s, s, \delta, b, q \big\rangle \overset{\epsilon/\mathrm{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle \sigma_s, s, 0, b, q \rangle$ *if* $\delta \neq \infty$;
  - *delay:* $\langle \sigma_s, s, d, b, q \rangle \overset{\epsilon/\mathrm{del}(\delta)/\epsilon}{\hookrightarrow} \langle \sigma_s, s + \delta, d + \delta, b, q \rangle$.

[5] The $\mathrm{update_s}$ function computes the minimal delay $\delta' \geq \delta$, such that the safety-property automaton still remains in an accepting state after processing the action a.

A configuration $\langle \sigma_s, s, d, b, q \rangle$ of the EM consists of the current stored sequence (i.e., the memory content) σ_s, two clock values s and d indicating respectively the time elapsed since the last store and dump operations, a Boolean b indicating whether the underlying enforced property is satisfied or not on the output sequence, and q the current state of $[\![\mathcal{A}]\!]$ reached after processing the sequence already released followed by the timed word in memory. Regarding its alphabet, in the input (resp. output) sequence, the EM either lets time elapse and no event is read or released, or reads and stores (resp. dumps and releases) a symbol event after some delays. Semantics rules can be understood as follows:

- The *store* rule is executed upon the reception of an event (δ, a). The timed event (δ', a) is appended to the memory content, where δ' is the minimal delay that has to be waited so that the property remains satisfied – if such a delay exists. The value of s is then reinitialized to 0. If a delay can be found through the $\mathrm{update_s}$ function, q is updated to the state that will be reached by appending the timed event (δ', a) to the output sequence concatenated with the contents of the memory, and b remains `tt` and becomes `ff` otherwise.
- The *dump* rule is executed when the value of d is equal to the delay of the first timed event in the memory. The value of d is then reinitialized to 0. The first event in memory is suppressed (and released from the enforcer). Other elements of the configuration remain unchanged.
- The *delay* rule adds the time elapsed δ to the current values of s and d when no store nor dump operation is possible.

Remark 1. The model of enforcement monitor presented in Definition 5 can be easily extended by relaxing two hypothesis: in the *store* rule, we check whether there is a delay greater than δ allowing the output sequence to stay in the accepting states of the property ($\delta' = \infty$). Of course, this condition can be adapted to a given time bound in $\mathbb{R}_{\geq 0}$. More complex conditions are also possible according to some desired quality of service. Similarly, processing input and output actions is assumed to be done in zero time. Some delay (either fixed or depending on additional parameters) can be considered for this action by modifying the *store* rule.

We define the language of runs of an enforcement monitor EM:

$$\mathcal{L}(EM) \subseteq (\Gamma_{EM})^* = \Big(\big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)\Big)^*$$

It is worth noticing that enforcement monitors are deterministic. Hence, given $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ and $t \in \mathbb{R}_{\geq 0}$, let $w \in \mathcal{L}(EM)$ be the unique maximal sequence such that

$$\Pi_\epsilon\Big(\bigodot_{i \in [1,|w|]} (\Pi_1(w(i)))\Big) = \mathrm{obs}(\sigma, t),$$

where Π_ϵ is the projection that erases ϵ from words in $((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})^*$.

Now, we define the enforcement function E associated to EM as

$$\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*,\ \forall t \in \mathbb{R}_{\geq 0},\ E(\sigma, t) = \Pi_\epsilon\Big(\bigodot_{i \in [1,|w|]} (\Pi_3(w(i)))\Big) \tag{1}$$

Proposition 1. *Given an enforcement monitor EM for a safety property φ and E defined as in Eq. (1), E verifies the soundness, transparency and optimality conditions of Definition 4.*

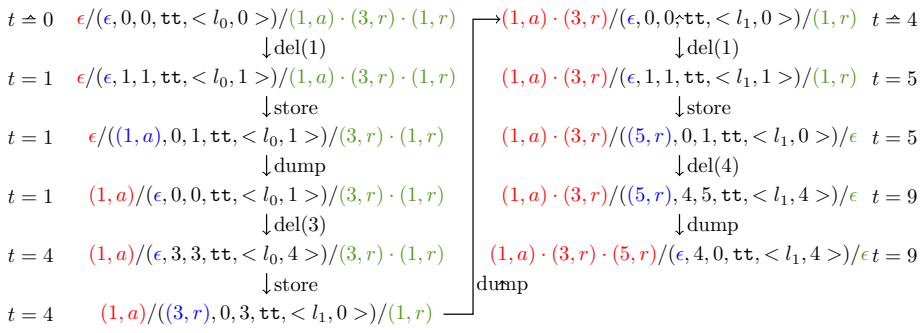


Fig. 3: Enforcer configuration evolution

Example 2. Let us illustrate how these rules are applied to enforce φ_1 (represented by the TA in Fig. 1a with $\Sigma_1 = \{a, r\}$). Let us consider the input timed word $\sigma = (1,a)\cdot(3,r)\cdot(1,r)$. Figure 3 shows how successive rules are applied and the evolution of the configurations of the EM. The variable t describes global time. The input is represented on the right-hand (resp. left-hand) side of the configuration.

5 Enforcement of Co-safety Properties

Let us now focus on the enforcement of co-safety properties. We assume a co-safety property φ specified by a co-safety timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ and its associated semantics $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$. An enforcer function E for a co-safety property φ should satisfy new soundness, transparency and optimality conditions.

Before defining those constraints, the notion of a sequence delaying another has to be modified in the context of co-safety properties. Let $\sigma, \sigma' \in (\mathbb{R}_{\geq 0}\times\Sigma)^*$ be two timed sequences, we note $\sigma' \preccurlyeq_c \sigma$ for $\Pi_\Sigma(\sigma') = \Pi_\Sigma(\sigma) \wedge \forall i \leq |\sigma'|, \mathrm{del}(\sigma'(i)) \geq \mathrm{del}(\sigma(i))$. This order between timed words shall be used in the transparency and optimality conditions below to constrain the sequences produced by an enforcer. We define $\gamma(\sigma) \stackrel{\text{def}}{=} \{\sigma' \preccurlyeq_c \sigma \mid \sigma' \models \varphi\}$, the set of sequences delaying σ and satisfying the property φ and $\gamma_t(\sigma) \stackrel{\text{def}}{=} \{\mathrm{time}(\sigma') \mid \sigma' \in \gamma(\sigma)\}$ the set of sums of delays of these sequences.

Definition 6 (Soundness, transparency, optimality). *An enforcement function* $E : (\mathbb{R}_{\geq 0}\times\Sigma)^* \times \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}_{\geq 0}\times\Sigma)^*$ *for a co-safety property* φ *is*
- sound if $\forall\sigma \in (\mathbb{R}_{\geq 0}\times\Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma,t) \neq \epsilon \Rightarrow (\exists t' \geq t, E(\sigma,t') \models \varphi)$.
- transparent if $\forall\sigma \in (\mathbb{R}_{\geq 0}\times\Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma,t) \neq \epsilon \Rightarrow (\exists t' \geq t,$
$E(\sigma,t') \preccurlyeq_c \mathrm{obs}(\sigma,t))$.

If E is sound and transparent, it is optimal if for any input $\sigma \in (\mathbb{R}_{\geq 0}\times\Sigma)^*$, at any time $t \in \mathbb{R}_{\geq 0}$, the following constraints hold:

(Op1) $\gamma\big(\mathrm{obs}(\sigma,t)\big) \neq \emptyset \wedge \forall t' < t, \gamma(obs(\sigma,t') = \emptyset) \Rightarrow$
$\big(\exists t' \geq t, |E(\sigma,t')| = |obs(\sigma,t)| \wedge t' = t + time(E(\sigma,t'))\big)$;
(Op2) $E(\sigma,t) \neq \epsilon \Rightarrow (1) \wedge (2)$, *where*
let $E(\sigma,t)_{[\cdots n]}$ *be the smallest prefix of* $E(\sigma,t)$ *s.t.* $E(\sigma,t)_{[\cdots n]} \models \varphi$ *in*
(1) $\nexists(\delta_1', \ldots, \delta_n'), \sum_{i=1}^{n}\delta_i' \leq \sum_{i=1}^{n}\mathrm{del}(E(\sigma,t)(i))$
$\wedge \bigodot_{i\in[1,n]}(\delta_i', \mathrm{act}((\sigma(i)))) \models \varphi \wedge \forall i \in [1,n], \mathrm{del}(\sigma(i)) \leq \delta_i'$
(2) $E(\sigma,t) \models \varphi \Rightarrow \big(E(\sigma,t) = E(\sigma,t)_{[\cdots n]} \cdot \mathrm{obs}(\sigma,t)_{[n+1\cdots|E(\sigma,t)|]}\big)$

Soundness means that if a timed word is released by the enforcement function, in the future, the output timed word of the enforcement function should satisfy the property φ. [6] Transparency means that the enforcement function should not change the order of events, and the delay between any two consecutive events cannot be reduced.

Optimality means that the output is produced as soon as possible: **Op1** means that if t is the first time instant at which there is a timed word that delays $\mathrm{obs}(\sigma, t)$ and satisfies φ, then, in the future at time $t' = t + \mathrm{time}(E(\sigma, t'))$, the enforcement monitor should have output exactly all the observed events until time t. **Op2**-1 means that if $E(\sigma,t) \neq \epsilon$ is released by the enforcement function, for the smallest prefix $E(\sigma,t)_{[\cdots n]}$ that satisfies φ, the total amount of time spent to trigger $E(\sigma,t)_{[\cdots n]}$ should be minimal.**Op2**-2 means that the delay between the remaining actions $\mathrm{obs}(\sigma,t)_{[n+1\cdots]}$ (i.e., when the property is satisfied) should not be changed. Similarly to safety properties, we expect the enforcement function to minimally alter the initial sequence: after correcting an incorrect prefix, the remainder of the sequence should be the same for events and delays between them.

Before presenting the definition of enforcement monitor, we introduce $\mathrm{update_c}$ as a function from $(\mathbb{R}_{\geq 0} \times \Sigma)^+ \to \mathbb{R}^+_{\geq 0} \times \mathbb{B}$ such that for $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^+$

$$\mathrm{update_c}(\sigma) \stackrel{\text{def}}{=} \begin{cases} \big((\delta_1, \ldots, \delta_{|\sigma|}), tt\big) \text{ s.t. } \sum_{i=1}^{|\sigma|} \delta_i = \min\{\gamma_t(\sigma)\}, \text{ if } \gamma(\sigma) \neq \emptyset \\ \big((\mathrm{del}(\sigma(1)), \ldots, \mathrm{del}(\sigma(|\sigma|))), \mathtt{ff}\big), \text{otherwise} \end{cases}$$

Definition 7 (Enforcement Monitor for co-safety properties). *An enforcement monitor EM for φ is a transition system $\langle C, C_0, \Gamma, \hookrightarrow\rangle$ s.t.:*

- $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B}$ *is the set of configurations and the initial configuration is $C_0 = \langle \varepsilon, 0, 0, \mathtt{ff}\rangle \in C$;*
- $\Gamma_{EM} = (\mathbb{R}_{\geq 0} \times \Sigma) \times Op \times (\mathbb{R}_{\geq 0} \times \Sigma)$ *is the "input-operation-output" alphabet, where $Op = \{\text{store-}\overline{\varphi}(\cdot), \text{store-}\varphi_{\text{init}}(\cdot), \text{store-}\varphi(\cdot), \mathrm{dump}(\cdot), \mathrm{delay}(\cdot)\}$;*
- $\hookrightarrow \subseteq C \times \Gamma_{EM} \times C$ *is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:*

1. store-$\overline{\varphi}$: $\langle \sigma_s, \delta, d, \mathtt{ff}\rangle \overset{(\delta,a)/\text{store}-\overline{\varphi}(\delta,a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta, a), 0, d, \mathtt{ff}\rangle$ *if* $\Pi_2\big(\mathrm{update_c}(\sigma_s \cdot (\delta, a))\big) = \mathtt{ff}$
2. store-φ_{init}: $\langle \sigma_s, \delta, d, \mathtt{ff}\rangle \overset{(\delta,a)/\text{store}-\varphi_{\text{init}}(\delta',a)/\epsilon}{\hookrightarrow} \langle \sigma'_s, 0, 0, \mathtt{tt}\rangle$ *if* $\Pi_2\big(\mathrm{update}_c(\sigma_s \cdot (\delta, a))\big) = \mathtt{tt}$ *with*
   - $\delta' = \Pi_1\big(\mathrm{update}_c(\sigma_s \cdot (\delta, a))\big)$
   - $\sigma'_s = \bigodot_{i \in [1, |\sigma_s|]} (\Pi_i(\delta'), \mathrm{act}(\sigma_s(i))) \cdot (\delta'_{|\sigma_s|+1}, a)$
3. store-φ: $\langle \sigma_s, \delta, d, \mathtt{tt}\rangle \overset{(\delta,a)/\text{store}-\varphi(\delta,a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta, a), 0, d, \mathtt{tt}\rangle$
4. *dump:* $\big\langle (\delta, a) \cdot \sigma_s, s, \delta, \mathtt{tt}\big\rangle \overset{\epsilon/\mathrm{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle \sigma_s, s, 0, \mathtt{tt}\rangle$
5. *delay:* $\langle \sigma_s, s, d, b\rangle \overset{\epsilon/\mathrm{delay}(\delta)/\epsilon}{\hookrightarrow} \langle \sigma_s, s + \delta, d + \delta, b\rangle$.

[6] As usual in runtime enforcement, either it is assumed that the empty sequence ϵ does belong to the property or the soundness constraint does not take ϵ into account.

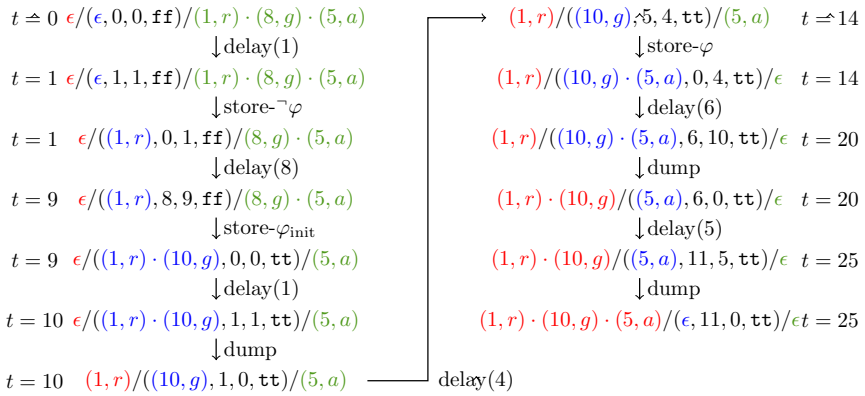


Fig. 4: Enforcer configuration evolution

The EM either lets time elapse when no event is read or released as output, or reads and stores (resp. dumps and outputs) an event after some delay. Semantic rules can be understood as follows:

- Upon reception of an event (δ, a), one of the three store rules is executed. The rule store-$\overline{\varphi}$ is executed if b `ff` and the property still remains unsatisfied after this new event (i.e., when the update_c function returns `ff`). If the update_c function returns `tt` (indicating that the φ can now be satisfied), then the rule store-φ_{init} is executed. When executing this rule, d is reset to 0, indicating that the enforcer can start outputting events. The rule store-φ is executed if the Boolean in the current configuration is `tt`, which indicates that the property is already satisfied by the inputs received earlier. So, in this case, it is not necessary to invoke the update_c function, and the event (δ, a) is appended to the memory.
- The *dump* rule is similar to the one of the enforcement of safety properties except that we wait that the Boolean indicating property satisfaction becomes `tt`.
- The *delay* rule adds the time elapsed to the current clock values s and d.

Note that, in this case, time measured in output starts elapsing upon property satisfaction by the memory content (contrarily to the safety case, where it starts with the enforcer).

As was the case in the previous section, from EM, we can define an enforcement function E as in Eq. (1), such that the following proposition holds:

Proposition 2. *Given an enforcement monitor EM for a co-safety property and E defined as in Eq (1), E is sound, transparent and optimal as per Definition 6.*

Example 3. Let us illustrate how these rules are applied to enforce φ_2 (Fig. 1b), with $\Sigma_2 = \{r, g, a\}$. Let us consider the input timed word $\sigma = (1,r)\cdot(8,g)\cdot(5,a)$. Figure 4 shows how semantic rules are applied, and the evolution of the configurations of the EM. The input is shown on the right of the configuration, and the output is presented on the left. The variable t describes global time. The resulting output is $E(\sigma) = (1,r)\cdot(10,g)\cdot(5,a)$, which satisfies the property φ presented in Fig. 1b.

6 Implementation

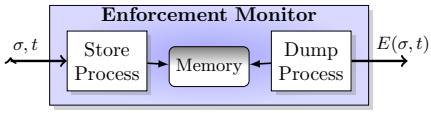


Fig. 5: Realizing an EM

Let us now provide the algorithms showing how enforcement monitors can be implemented. As shown in Fig. 5, the implementation of an enforcement monitor (EM) consists of two processes running concurrently (Store and Dump) and a memory. The Store process models the store rules. The memory contains the timed words σ_s. The Dump process reads events stored in the memory and releases them as output after the required amount of time. To define the enforcement monitors, the following algorithms assume a TA $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$.

Algorithm 1 DumpProcess$_{\mathsf{safety}}$

$d \leftarrow 0$
while `tt` **do**
  await $(|\sigma_s| \geq 1)$
  $(\delta, a) \leftarrow$ dequeue (σ_s)
  wait $(\delta - d)$
  dump (a)
  $d \leftarrow 0$
end while

Algorithm 2 StoreProcess$_{\mathsf{safety}}$

$(l, X) \leftarrow (l_0, [X \leftarrow 0])$
while `tt` **do**
  $(\delta, a) \leftarrow$ await $event$
  if $(\mathrm{post}(l, X, a, \delta) \notin G)$ **then**
    $\delta' \leftarrow \mathrm{update}(l, X, a, \delta)$
    if $\delta' = \infty$ **then**
      terminate StoreProcess
    end if
  else
    $\delta' \leftarrow \delta$
  end if
  $(l, X) \leftarrow \mathrm{post}(l, a, X, \delta')$
  enqueue (δ', a)
end while

Algorithm 3 DumpProcess$_{\mathsf{co-safety}}$

await $startDump$
$d \leftarrow 0$
while `tt` **do**
  await $(|\sigma_s| \geq 1)$
  $(\delta, a) \leftarrow$ dequeue (σ_s)
  wait $(\delta - d)$
  dump (a)
  $d \leftarrow 0$
end while

Algorithm 4 StoreProcess$_{\mathsf{co-safety}}$

$goalReached \leftarrow$ `ff`
while `tt` **do**
  $(\delta, a) \leftarrow$ await $(event)$
  enqueue(δ, a)
  if $goalReached =$ `ff` **then**
    $(newDelays, R) \leftarrow \mathrm{update}_c(\sigma_s)$
    if $R =$ `tt` **then**
      $modify\ delays$
      $goalReached \leftarrow$ `tt`
      notify $(startDump)$
    end if
  end if
end while

We now describe these processes for safety properties.

- The DumpProcess$_{\mathsf{safety}}$ algorithm (see Algorithm 1) is an infinite loop that scrutinizes the memory and proceeds as follows: Initially, d is set to 0. If the memory is empty ($|\sigma_s| = 0$), it waits until a new element (δ, a) is stored in memory, otherwise it proceeds with the first element in memory. Meanwhile, d keeps track of the time elapsed since the last dump operation. The DumpProcess$_{\mathsf{safety}}$ waits for $(\delta - d)$ time units before releasing the action a and resets d.
- The StoreProcess$_{\mathsf{safety}}$ algorithm (see Algorithm 2) is an infinite loop that scrutinizes the system for input events. It proceeds as follows. Let (l, X) be the state of the

property automaton, where l represents the location and X is the current clock values initialized to $(l_0, 0)$. The function post takes a state of the property automaton (l, X), an event (δ, a), and computes the state reached by the property automaton. The update function computes a new delay δ' such that the property automaton will reach an accepting state in an optimal way by triggering (δ', a).

We now describe these processes for co-safety properties.

- The $\mathsf{DumpProcess}_{\mathsf{co-safety}}$ algorithm for co-safety properties (see Algorithm 3) resembles the one of the safety case. The only difference is that the infinite loop starts only after receiving the $startDump$ notification from the $\mathsf{StoreProcess}_{\mathsf{co-safety}}$.
- In the $\mathsf{StoreProcess}_{\mathsf{co-safety}}$ algorithm (see Algorithm 4), $goalReached$ is a Boolean, used to indicate if the goal location is visited by the input events which were already processed. It is initialized to `ff`. The update_c function takes all events stored in the enforcer memory, and returns new delays and if the goal location is reachable. $startDump$ is a notification message sent to the $\mathsf{DumpProcess}_{\mathsf{co-safety}}$, to indicate that it can start dumping the events stored in the memory. Note that the update_c can be easily implemented using the optimal path routine of UPPAAL.

7 Evaluation

Enforcement monitors for safety and co-safety properties, based on the algorithms presented in the previous section, have been implemented in prototype tool of 500 LOC using Python. The tool also uses UPPAAL [12] as a library to implement the update function and the pyuppaal library to parse UPPAAL models written in XML.

We present some performance evaluation on a simulated system where the input timed trace is generated. As described in Sec. 6, enforcement monitors for safety and co-safety properties are implemented by two concurrent processes. The TA representing the property is a UPPAAL model, and is an input to the enforcement monitor. The UPPAAL model also contains another automaton representing the sequence of events received by the enforcement monitor. The update function of the $\mathsf{StoreProcess}$ uses UPPAAL. Experiments were conducted on an Intel Core i7-2720QM at 2.20GHz CPU, and 4 GB RAM running on Ubuntu 12.04 LTS. Note that the implementation is a prototype, and there is still scope for improving the performance.

Results of the performance analysis of our running example properties are presented in Tables 1a and 1b. The values are presented in seconds. Average values are computed over multiple runs. The length of the input trace is denoted by $|tr|$. The entry $\mathrm{t\_tr}$ represents the time taken by the system simulator process to generate the trace. The entry $\mathrm{t\_update}$ (resp. $\mathrm{t\_Post}$) indicates the time taken for one call to the update (resp. post) function when the last event of the input trace is received. The entry $\mathrm{t\_EM}$ presents the total time from the start of the simulation until the last event is dumped by the enforcer. The throughput shows how many events can be processed by the enforcer ($|\mathrm{tr}|/\mathrm{t\_EM}$).

We observe that the throughput decreases with the length of the input trace. This unexpected behavior stems from the external invocation of UPPAAL to realize post and update functions. Indeed, after each event, the length of the automaton representing the trace grows, and, as indicated in Table 1a, the time taken by update and post functions also increases, unnecessarily starting the computation from the initial location each time an event is received. Future implementations will avoid this by realizing the post and update functions online from the current state. Performance and throughput shall

Table 1: Performance analysis of enforcement monitors

(a) For φ_1

$\|tr\|$	t_update	t_post	t_tr	t_EM	throughput
100	0.0433	0.0383	0.00483	2.648	37
200	0.08196	0.07158	0.0087	9.135	21.89
300	0.121	0.1065	0.0118	19.42	15.46
400	0.1696	0.1525	0.0133	34.314	11.65
500	0.2148	0.1891	0.0142	53.110	9.41
600	0.2668	0.2334	0.0166	77.428	7.75
700	0.3164	0.2789	0.0178	107.61	6.50
800	0.3669	0.3289	0.0198	143.53	5.57
900	0.4256	0.3810	0.0237	181.06	4.97
1000	0.4878	0.4352	0.0259	229.12	4.36

(b) For φ_2

$\|tr\|$	t_update	t_tr	t_EM
100	0.063	0.0026	1.28
200	0.17	0.0065	8
300	0.33	0.0081	25
400	0.54	0.0115	58
500	0.79	0.0131	109
600	1.11	0.0157	186
700	1.50	0.0186	297
800	1.96	0.0209	462
900	2.40	0.0234	623
1000	2.84	0.0341	852

be independent from the trace length. Further experiments have been carried out on different examples similarly demonstrating feasibility and scalability.

For co-safety properties, regarding the total time $\mathrm{t\_EM}$, note that the most expensive operation update is called upon each event. Moreover, examining the column t_update in Table 1b, the time taken by the update function increases with the number of events. This behavior is expected for co-safety properties, as we check for an optimal output from the initial state after each event. Please note that in case of a co-safety property, once the property is satisfied (a good location is reached), it is not necessary to invoke the update function. From that point onwards, the increase in total time $\mathrm{t\_EM}$ per event will be very less (since we just add the received event to the output queue), and $\mathrm{t\_update}$ will be zero for the events received later on.

8 Related Work

This work is by no means the first to address monitoring of timed properties. Matteucci inspires from partial-model checking techniques to synthesize controller operations to enforce safety and information-flow properties using process-algebra [13]. Monitors are close to Schneider's security automata [7]. The approach targets discrete-time properties and systems are modelled as timed processes expressed in CCS. Compared to our approach, the description of enforcement mechanisms remains abstract, directly restricts the monitored system, and no description of monitor implementation is proposed.

Other research efforts aim to mainly runtime verify timed properties and we shall categorize them into i) rather theoretical efforts aiming at synthesizing monitors, and ii) tools for runtime monitoring of timed properties.

Synthesis of timed automata from timed logical formalisms Bauer et al. propose an approach to runtime verify timed-bounded properties expressed in a variant of Timed Linear Temporal Logic [4]. Contrarily to TLTL, the considered logic, TLTL_3, processes finite timed words and the truth-values of this logic are suitable for monitoring. After reading some timed word u, the monitor synthesized for a TLTL_3 formula φ state the verdict $\top$ (resp. $\bot$) when there is no infinite timed continuation w such that $u \cdot w$

satisfy (resp. does not satisfy) φ. Another variant of LTL in a timed context is the metric temporal logic (MTL), a dense extension of LTL. Nickovic et al. [14, 3] proposed a translation of MTL to timed automata. The translation is defined under the bounded variability assumption stating that, in a finite interval, a bounded number of events can arrive to the monitor. Still for MTL, Thati et al. propose an online monitoring algorithm by rewriting of the monitored formula and study its complexity [1]. Later, Basin et al. propose an improvement of this approach having a better complexity but considering only the past fragment of MTL [5].

Runtime enforcement of timed properties as presented in this paper is compatible with the previously described approaches. These approaches synthesize automata-based decision procedures for logical formalisms. Decision procedures synthesized for safety and co-safety properties could be used as input to our framework.

Tools for runtime monitoring of timed properties The Analog Monitoring Tool [10] is a tool for monitoring specifications over continuous signals. The input logic of AMT is STL/PSL where continuous signals are abstracted into propositions and operations are defined over signals. Input signal traces can be monitored in an offline or incremental fashion (i.e., online monitoring with periodic trace accumulation).

LARVA [15, 11] takes as input properties expressed in several notations, e.g., Lustre, duration calculus. Properties are translated to DATE (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. Transitions are augmented with code that modify the internal system state. DATE target only safety properties. In addition, LARVA is able to compute an upper-bound on the overhead induced on the target system. The authors also identify a subset of duration calculus, called counter-examples traces, where properties are insensitive to monitoring [16].

Our monitors not only differ by their objectives but also by how they are interfaced with the system. We propose a less restrictive framework where monitors asynchronously read the outputs of the target system. We do not assume our monitors to be able to modify the internal state of the target program. The objective of our monitors is rather to correct the timed sequence of output events before this sequence is released to the environment (i.e., outside the system augmented with a monitor).

9 Conclusion and Future Work

This paper introduces runtime enforcement for timed properties and provides a complete framework. We consider safety and co-safety properties described by timed automata. We propose adapted notions of enforcement monitors with the possibility to delay some input actions in order to satisfy the required property. For this purpose, the enforcement monitor can store some actions during a certain time period. We propose a set of enforcement rules ensuring that outputs not only satisfy the required property (if possible), but also with the "best" delay according to the current situation. We describe how to realize the enforcement monitor using concurrent processes, how it has been prototyped and experimented. This paper introduced the first steps to runtime enforcement of (continuous) timed properties. However, several research questions remain open. As this approach targets explicitly safety and co-safety properties, it seems desirable to investigate whether more expressive properties can be enforced, and if so, pro-

pose enforcement mechanisms for them. We expect to extend our approach to Boolean combinations of timed safety and co-safety properties, and more general properties. The question requires further investigation since the update function would have to be adapted. A precise characterization of *enforceable timed properties* would thus be possible, as was the case in the untimed setting [4, 17]. Also related to expressiveness is the question of how the set of timed enforceable properties is impacted when the underlying memory is limited and/or the primitives operations endowed to the monitor are modified. A more practical research perspective is to study the implementability of the approach proposed in this paper, e.g., using *robustness* of timed automata.

References

1. Thati, P., Rosu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theor. Comput. Sci. **113** (2005) 145–162
2. Chen, F., Rosu, G.: Parametric trace slicing and monitoring. In Kowalewski, S., Philippou, A., eds.: TACAS. Volume 5505 of LNCS., Springer (2009) 246–261
3. Nickovic, D., Piterman, N.: From MTL to deterministic timed automata. In Chatterjee, K., Henzinger, T.A., eds.: FORMATS. Volume 6246 of LNCS., Springer (2010) 152–167
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20** (2011) 14
5. Basin, D.A., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In Khurshid, S., Sen, K., eds.: RV. Volume 7186 of LNCS., Springer (2011) 260–275
6. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: FM 2012: 18th International symposium on Formal Methods. (2012) Accepted for publication. To appear.
7. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000)
8. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transaction Information System Security. **12** (2009)
9. Falcone, Y.: You should better enforce than verify. In: Runtime Verification. (2010) 89–105
10. Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Formal Modeling and Analysis of Timed Systems. (2007) 304–319
11. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM. (2009) 33–37
12. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) **1** (1997) 134–152
13. Matteucci, I.: Automated synthesis of enforcing mechanisms for security properties in a timed setting. Electron. Notes Theor. Comput. Sci. **186** (2007) 101–120
14. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In Asarin, E., Bouyer, P., eds.: FORMATS. Volume 4202 of LNCS., Springer (2006) 274–289
15. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: FMICS. (2008) 135–149
16. Colombo, C., Pace, G.J., Schneider, G.: Safe runtime verification of real-time properties. In: Formal Modeling and Analysis of Timed Systems, 7th International Conference (FORMATS). Volume 5813 of LNCS., Budapest, Hungary (2009) 103–117
17. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? STTT **14** (2012) 349–382