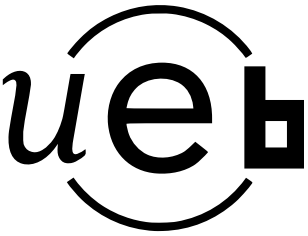
N° d'ordre : ANNÉE 2015



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale MATISSE

présentée par

Srinivas PINISETTY

préparée à l'unité de recherche IRISA-INRIA: UMR 6074
Institut de recherche en informatique et système aléatoires ISTIC

Enforcement à l'exécution de propriétés temporisées

Runtime Enforcement of Timed Properties

Thèse soutenue à Rennes le 23 janvier 2015

devant le jury composé de :

Martin LEUCKER
Professor, Université de Lübeck / *Rapporteur*

Didier LIME
HDR, Maître de Conférences, École Centrale de Nantes / *Rapporteur*

Sophie PINCHINAT
Professeure à Université de Rennes 1 / *Examinatrice*

Frédéric HERBRETEAU
Maître de conférence, ENSEIRB Bordeaux / *Examinateur*

Yliès FALCONE
Maître de conférence à Université de Grenoble 1 / *Invité*

Thierry JÉRON
Directeur de recherche à l'INRIA Rennes-Bretagne Atlantique / *Directeur de thèse*

Hervé MARCHAND
Chargé de recherche à l'INRIA Rennes-Bretagne Atlantique / *Co-directeur de thèse*

Acknowledgements

This thesis has been influenced by many people, and I would like to thank all of them. Firstly, I must thank my supervisors Thierry Jéron, Hervé Marchand, and Yliès Falcone for offering me a PhD position, providing a very encouraging research environment, and accompanying me in every step of this long journey. In spite of their busy schedules, they always gave me time to discuss, to answer my questions, and attended to my concerns from day one. Discussions with them was always a pleasure given their passion for research, immense knowledge, experience, enthusiasm, and patience. I had learnt a lot from them during these years, and could not have imagined having better advisors for my PhD study.

I should also thank Yliès Falcone for visiting us several times, and also for inviting me a couple of times to his lab to conduct this research work. During my PhD, I also had a chance to work with Antoine Rollet and Omer Nguena Timo. I thank them for collaborating and their contributions in this work.

Further, I thank other members of the thesis examining committee: Didier Lime, Martin Leucker, Sophie Pinchinat, and Frédéric Herbreteau. It is a honour for me that you dedicated your time to read my thesis, and to attend my thesis defense. Reporters Didier Lime, and Martin Leucker provided detailed feedback on my thesis, and improved the quality of my thesis.

I also thank all the VERTECS and SUMO team members for supporting me and providing necessary information and help, for any issue I had (related to work or others) during my stay in Rennes. I thank my friends Yogesh, Raghu ram, Sharat, Akshay, Ajay, Mani, Hrishikesh, Aswin, Deepak, all my other friends, and team members at INRIA in Rennes for making these years of PhD study a memorable one.

I should also thank all my friends, and teachers who always encouraged me and provided valuable advices when I needed. I thank my wife for being with me during last months of my PhD study, and for understanding, supporting and encouraging me. Finally, very special thanks to my parents, sister and family for always believing in me and for standing by me throughout the course of my study.

Srinivas PINISETTY
January 2015

Contents

Chapter 1

Résumé en Français

Les techniques de vérification formelle comme le model-checking [CE82, QS82, BK08] sont bien adaptées pour la vérification de systèmes complexes critiques. Toutefois, même si plusieurs modèles formels, techniques d'analyse et outils ont été développés ces dernières années, le passage à l'échelle de ces techniques constitue un goulet d'étranglement qui empêche leur utilisation à grande échelle dans l'industrie. Ce dernier point motive l'utilisation de techniques et d'outils qui permettent à un logiciel ou à un système de continuer à fonctionner même en présence de fautes ou de pannes. On peut dans ce contexte citer l'enforcement de propriétés à l'exécution [Sch00, Fal10] (runtime enforcement), qui suit l'exécution d'un système et qui contrôle le respect d'exigences définies formellement sur le système.

L'enforcement à l'exécution étend la vérification à l'exécution [BLS11] et peut se définir comme l'ensemble des théories, techniques et outils dont le but est d'assurer la conformité des exécutions d'un système vis à vis d'un ensemble de propriétés. L'utilisation d'un *moniteur d'enforcement* permet de transformer l'exécution du système (vue comme un séquence d'événements) pour satisfaire une propriété (p. ex. une propriété de sûreté). Ce moniteur d'enforcement est habituellement construit de manière à respecter deux contraintes :

- la séquence fournie en sortie par le moniteur d'enforcement doit satisfaire la propriété (*correction*), et
- si la séquence lue par le moniteur satisfait déjà la propriété, la sortie doit rester inchangée (*transparence*).

L'enforcement à l'exécution a été largement étudié ces dernières années pour des propriétés non-temporisées [Sch00, LBW09, FMFR11]. La notion de temps a déjà été prise en compte dans des approches d'enforcement à l'exécution dans [Mat07] pour des propriétés à temps discret, et dans [BJKZ13] où l'écoulement du temps est modélisé par une séquence d'événements incontrôlables ("ticks").

Dans cette thèse, nous nous intéressons aux *mécanismes d'enforcement pour des propriétés temporisées à temps dense*. Pour de telles propriétés (sur des séquences finies), non seulement l'ordre des événements est important (comme dans le cas non temporisé), mais le temps écoulé entre l'occurrence de deux événements influence également

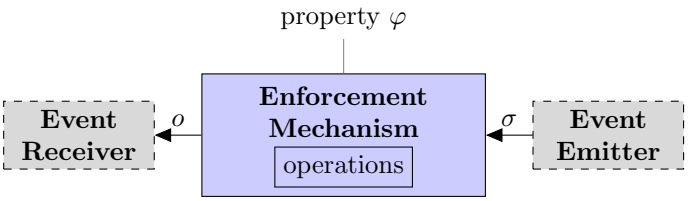


Figure 1.1: Mécanisme d'enforcement (enforcement mechanism).

l'évaluation de la propriété. Les propriétés temporisées permettent donc de décrire plus précisément les comportements désirés des systèmes dans la mesure où elles spécifient l'écoulement du temps attendu entre les événements. Ceci les rend particulièrement utiles pour certains domaines d'application [PFJM14a]. Par exemple, dans un contexte de sécurité, les moniteurs d'enforcement peuvent être utilisés comme des "*firewalls*" permettant d'éviter des attaques par déni de service en assurant un délai minimal entre des événements. Sur un réseau, les moniteurs d'enforcement peuvent être utilisés pour synchroniser des flux d'événements, ou pour assurer que les événements respectent les pré-conditions de services.

1.1 Contexte

Dans cette thèse, nous nous focalisons sur l'*enforcement à la volée de propriétés temporisées*. Le contexte général est décrit par la Figure 2.1. Plus spécifiquement, étant donnée une propriété temporisée φ, nous cherchons à synthétiser un mécanisme d'enforcement fonctionnant à l'exécution. Afin d' être le plus général possible, ce mécanisme d'enforcement est supposé être placé entre un émetteur et un récepteur d'événments qui s'exécutent de manière asynchrone. Cette architecture abstraite est suffisamment générique pour être instanciée à un grand nombre de cas concrets où l'émetteur et le récepteur peuvent être vus soit comme un programme soit comme l'environnement.

Le but d'un moniteur d'enforcement est de lire une séquence d'événements σ (potentiellement incorrecte) produite par l'émetteur et de la transformer en une séquence de sortie o correcte vis à vis d'une propriété φ, o étant alors l'entrée du récepteur. Nous supposons que le médium de communication entre l'émetteur et le récepteur, via le mécanisme d'enforcement, est sûr et que ce dernier n'induit pas de délai de communication. Dans notre cadre temporisé, les événements sont modélisés par des actions auxquelles sont associées leurs dates d'occurrence. Les séquences d'événements en entrée et sortie du mécanisme d'enforcement sont donc modélisées comme des mots temporisés et le mécanisme d'enforcement est modélisé par une fonction de transformation de mots temporisés.

– **Expressivité du formalisme des spécifications/propriétés.** Un concept central en vérification et enforcement à l'exécution est la génération de moniteurs à partir d'un langage de spécification de haut niveau permettant de décrire les propriétés. Nous avons choisi de nous baser sur le modèle des automates tem-

porisés qui est traditionnellement utilisé pour la modélisation et la vérification de systèmes temps-réels. Un automate temporisé (TA) [AD94] est un automate d'états finis étendu avec un ensemble fini de variables réelles appelées *horloges* servant à modéliser le temps (continu). Dans cette thèse, nous considérons 3 grandes classes de propriétés: Les propriétés de sûreté (*safety*) qui modélisent le fait que aucun comportement mauvais ne se produit; les *co-safety* qui modélisent le fait qu'un comportement désiré va inévitablement se produire après un temps fini et les propriétés *régulières* qui contiennent ces deux classes et regroupent toutes les propriétés modélisables par des TA. De plus, afin de considérer des spécifications encore plus expressives, nous avons également étendu les TA avec des variables entières et des paramètres.

– **Capacité du mécanisme d'enforcement**. Le mécanisme d'enforcement que nous considérons retarde le temps, c'est à dire que sa principale primitive d'enforcement permet d'augmenter la date d'occurrence des actions. Outre cet aspect, le moniteur peut également supprimer des actions quand il n'est plus possible de satisfaire une propriété en retardant les actions.
En résumé, étant donnés un propriété régulière φ et un mot temporisé σ en entrée, le mécanisme d'enforcement reçoit un mot temporisé σ et produit en sortie un mot temporisé o qui satisfait φ. Le mot temporisé o est obtenu en retardant les dates d'occurrence des actions d'une sous-séquence de σ.

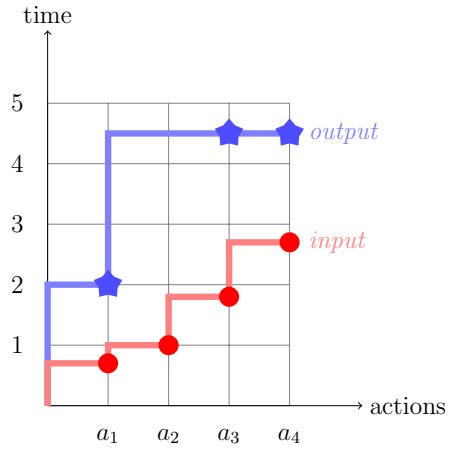


Figure 1.2: Comportement possible d'un mécanisme d'enforcement

La Figure 1.2 illustre le paradigme d'enforcement à l'exécution que nous considérons, c'est à dire comment un mécanisme d'enforcement se comporte afin de corriger une séquence d'entrée. La courbe rouge (resp. bleue) représente la séquence d'entrée (resp. de sortie) avec les actions en abscisse et les dates d'occurrence de celles-ci en ordonnée. En plus de la satisfaction de la propriété (non représentée sur la figure), le mécanisme d'enforcement ne doit pas changer l'ordre des actions, mais peut soit augmenter les dates d'occurrences de celles-ci,

soit supprimer les actions (p. ex. l'action a_2). À noter qu'il peut réduire le délai entre 2 actions (p. ex. a_3 et a_4). De plus, de manière à avoir un impact minimal sur la séquence d'entrée, le mécanisme d'enforcement doit fournir les actions en sortie le plus tôt possible.

Pour continuer notre illustration du paradigme d'enforcement, considérons le cas de 2 processus accédant à une ressource partagée et réalisant une action sur celle-ci. Chaque processus i (avec $i \in \{1, 2\}$) interagit avec cette ressource via 3 actions: acquisition (acq_i), release (rel_i), et une opération spécifique (op_i). De plus les 2 processus peuvent réaliser une action commune op. Le système démarre sur l'occurrence de l'action *init*. Dans la suite, la variable t permet de coder le passage du temps.

Considérons la spécification suivante: "*Les opérations op_1 et op_2 doivent s'exécuter de manière transactionnelle. Les 2 opérations doivent être exécutées sans ordre a priori et toute transaction doit contenir à la fois l'opération op_1 et op_2. Toute transaction doit se finir en moins de 10 unités de temps. Plusieurs occurrences de l'opération op peuvent se produire entre op_1 et op_2. Enfin chaque opération doit être espacée de la précédente d'au moins 2 unités de temps.*

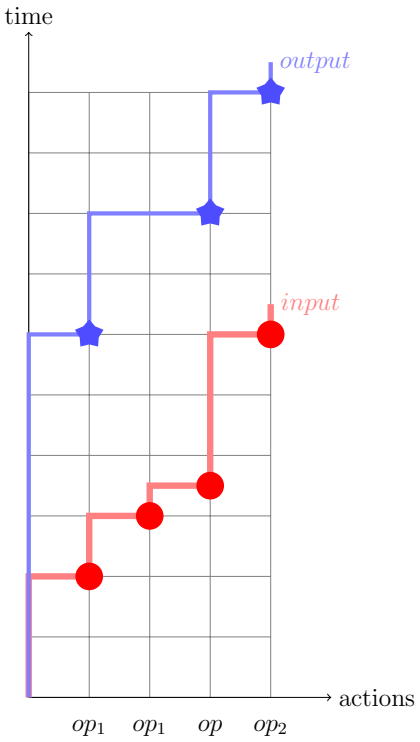


Figure 1.3: Exemple illustrant le mécanisme d'enforcement

Considérons la séquence d'entrée $\sigma = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. À $t = 2$, le moniteur d'enforcement ne peut produite op_1 en sortie car cette action ne satisfait pas la propriété (et le moniteur ne connaît pas encore les dates et occurrences des événements suivants). À $t = 3$, Le moniteur reçoit une nouvelle fois l'action op_1.

Clairement, il n'y a aucune manière de calculer de nouvelles dates pour ces 2 actions op_1 de manière à satisfaire la spécification. Le moniteur choisit donc de supprimer la deuxième occurrence de l'action op_1. À $t = 3.5$, quand le moniteur reçoit l'action op, la séquence d'entrée ne satisfait toujours pas la spécification, mais il existe encore des continuations possibles permettant la satisfiabilité de celle-ci. À $t = 6$, sur réception de l'action op_2, le moniteur peut calculer des délais appropriés entre les actions op_1 suivi de op et op_2 de manière à satisfaire la spécification. Ainsi, la date associée à op_1 est fixée à 6 (c'est à dire la plus petite date possible au moment de la décision), 8 pour l'action op (le délai minimal entre deux actions est fixé à 2), et 10 pour l'action op_2. Finalement, comme décrit par la Figure 1.3, la sortie du moniteur d'enforcement pour σ est $(6, op_1) \cdot (8, op) \cdot (10, op_2)$.

1.2 Résumé de l'approche

Ce paragraphe présente un résumé de l'approche utilisée pour la synthèse d'un moniteur d'enforcement pour une propriété temporisée.

À un niveau abstrait, un mécanisme d'enforcement d'une propriété φ peut être vu comme une fonction prenant en entrée un mot temporisé et fournissant en sortie un mot temporisé satisfiant φ (c.f. Figure 5.5).

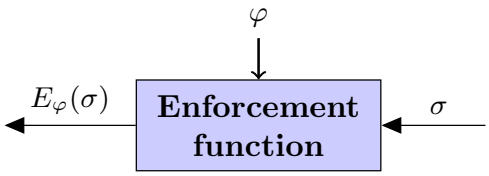


Figure 1.4: Fonction d'enforcement

Les contraintes que doivent satisfaire un mécanisme d'enforcement sont les suivantes:

- *Contraintes physiques*: toute événement produit en sortie par le moniteur d'enforcement ne peut être remis en cause par celui-ci.
- *Correction*: le moniteur doit produire uniquement des séquences qui satisfont la propriété φ
- *Transparence*: le moniteur peut seulement i) retarder les actions sans en changer l'ordre d'occurrence ou ii) supprimer des événements.

Afin de simplifier la conception et l'implémentation d'un mécanisme d'enforcement dans un contexte temporisé, nous avons choisi de le décrire à trois niveaux d'abstraction: la *fonction d'enforcement*, le *moniteur d'enforcement* , et les *algorithmes d'enforcement*.

Définition fonctionnelle. Une fonction d'enforcement décrit une transformation fonctionnelle d'un mot temporisé (en entrée) en un autre mot temporisé (en sortie). Son but est de décrire, à un niveau abstrait, pour chaque mot temporisé σ, le mot de sortie $E_\varphi(\sigma)$ attendu de manière à ce que cette fonction satisfasse les contraintes précédemment décrites.

Moniteur d'enforcement. Cette description fonctionnelle est par la suite raffinée en une vue plus concrète qui définit le comportement opérationnel d'un mécanisme d'enforcememnt en fonction du temps. Il est défini via un système de transitions étiquetées infini.

Implémentation et évaluation. Les algorithmes décrivent comment implémenter de manière concrète ces mécanismes d'enforcement pour toute propriété temporisée régulière spécifiée par un automate temporisé. L'implémentation est réalisée en Python et utilise des librairies UPPAAL [BY03]. Nos résultats expérimentaux nous ont permis d'avoir un premier retour sur expérience concernant la validité de notre approche et de montrer le caractère effectif des mécanismes d'enforcement dans un cadre temps-réel.

1.3 Résultats

Nous avons développé des mécanismes d'enforcement à l'exécution pour des spécifications avec contraintes temporelles fortes. Dans cette thèse, nous avons décrit comment synthétiser des moniteurs d'enforcements à partir d'une description formelle des propriétés en termes d'automates temporisés. Toutes les propriétés temporisées modélisables par des automates temporisés finis sont supportées dans ce cadre. Les mécanismes d'enforcement permettent de retarder les actions (tout en permettant de réduire l'intervalle de temps entre celles-ci) et de supprimer des actions dès lors qu'il n'est plus possible de satisfaire la propriété en retardant les actions, et ceci quelque soit le futur possible. Des algorithmes basés sur ce mécanisme ont également été implémentés et leur correction est prouvée formellement.

Ce travail a donné lieu à plusieurs publications dans des conférences internationales et journaux. Nous rappelons ici les résultats pour chacune d'entre elles. Ceci montre l'évolution de notre démarche et la manière dont nous avons généralisé (et simplifié) l'approche au fil du temps.

- **Runtime Enforcement of Timed Properties [PFJ+12].** [1]
  Dans [PFJ+12], nous avons introduit le concept d'enforcement à l'exécution pour des propriétés temporisées de *safety* et *co-safety* modélisées par des automates temporisées. Pour ce premier résultat la puissance des mécanismes d'enforcement se résumait à augmenter les délais entre chaque action. Nous avons proposé une notion d'optimalité du mécanisme d'enforcement qui devait, en fonction de la situation, calculer les plus petits délais entre chaque action afin de satisfaire la propriété (sous la contrainte que ceux-ci devaient être supérieurs aux délais initiaux). Ce travail a donné lieu à une première implémentation et à des expérimentations démontrant la validité de notre approche.
- **Runtime Enforcement of Regular Timed Properties [PFJM14b].**
  L'approche présentée dans [PFJ+12] se focalise sur l'enforcement de propriétés de safety et co-safety. Nous avons par la suite étendu cette approche en considérant la classe des propriétés régulières, ces dernières permettant d'exprimer

1. Les titres correspondent à ceux des articles.

notamment une certaine forme de comportement transactionnel. Une des difficultés à considérer est que ce type de propriétés n'est close ni par préfixe ni par suffixe. Pour un mot temporisé en entrée, le mot en sortie calculé par le moniteur d'enforcement alterne donc entre des préfixes qui satisfont la propriété et d'autres qui ne la satisfont pas, mais dont on est sur que son extension la satisfera. Le moniteur d'enforcement doit donc prendre en compte cet aspect lors du calcul des dates d'occurrence des actions et des éventuelles suppressions. Ce travail a également donné lieu à une nouvelle implémentation.

– **Runtime Enforcement of Regular Timed Properties [PFJ+14].**
Dans [PFJ+14], nous avons généralisé et simplifié les résultats de [PFJ+12, PFJM14b] et démontré formellement tous les résultats.

– **Runtime Enforcement of Regular Timed Properties by Suppressing and Delaying Events.**
Dans [PFJ+12, PFJM14b, PFJ+14] le moniteur ne pouvait qu'augmenter les délais entre les actions. Nous avons généralisé ce résultat dans un article soumis au journal Science of Computer Programming (SCP) en permettant au moniteur de supprimer des événements (seulement quand cela était nécessaire) et en lui permettant d'augmenter les dates d'occurrence d'événements (quitte à éventuellement réduire les délais entre événements).

– **Runtime Enforcement of Parametric Timed Properties with Practical Applications [PFJM14a].**
Pour divers domaines d'applications (sécurité des réseaux, protocoles de communications, etc), beaucoup de propriétés attendues du système comportent à la fois des contraintes sur le temps et sur les données. Dans [PFJM14a], nous avons donc étendu les résultats de [PFJ+12] en considérant un modèle de propriétés temporisées paramétrées avec variables (ces dernières pouvant être internes ou externes, c'est à dire portées par les événements). Un des paramètre permet d'instancier la propriété en fonction par exemple d'un numéro de session. Les événements reçus sont alors dispatchés sur des moniteurs en fonction de ce paramètre et ré-assemblés en sortie. Pour [PFJM14a], nous nous sommes focalisés sur des propriétés de safety et des moniteurs qui ne pouvaient qu'augmenter les délais entre les actions. Cette limitation a été levée dans le manuscrit de thèse: nous considérons maintenant tout type de propriétés et les possibilités du moniteur correspondent à celles décrites dans le papier soumis au journal SCP.

Chapter 2

Introduction

2.1 Motivations for Runtime Enforcement

Embedded and cyber-physical systems (CPS) are usually composed of multiple subsystems that are distributed and possibly developed using several programming languages. The behavior of such systems is affected by several external factors since they generally interact with many systems and users, and are integrated in networked environments. As the complexity of a system and its environment increase, ensuring that the system is error-free becomes a challenge. Failure of a system may result in severe financial losses, and in case of a safety-critical system, it may be even worse since it may also result in loss of human lives [LT93].

Using formal methods and model driven development approaches for designing and developing systems is a major area of research. Expressing requirements using a formal specification will make requirements clearer and remove ambiguities and inconsistencies. Formal languages are more easily amenable to automatic processing, by means of tools [Jan02]. Various formal verification techniques such as model checking [BK08] involve the formal modeling of computing systems and the verification of properties on the models, including safety and timing properties. These model-based techniques are well suited for verifying complex safety-critical systems, because they guarantee absence of errors.

Although several formal modeling and analysis techniques and tools based on them have been developed over the years, scalability of the techniques is a central issue that has prevented their widespread adoption. This motivates the necessity for techniques and tools that enable the system to tolerate failures and to continue operating in case of failures. An example is runtime enforcement techniques [Sch00, Fal10] that monitor the execution of system (at runtime) and control its compliance against the requirements of the system that are formally defined. Runtime enforcement extends runtime verification [BLS11] and refers to the theories, techniques, and tools aiming at ensuring the conformance of the executions of systems under scrutiny with respect to some desired property. Using an enforcement monitor, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence

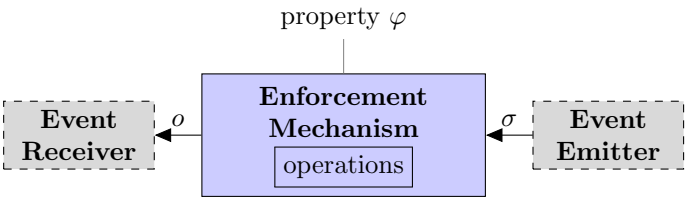


Figure 2.1: Enforcement mechanism.

that complies with a property (e.g., formalizing a safety requirement). It is based on two properties: the output sequence complies with the property (soundness) and if the input already complies with the property, it should be left unchanged (transparency).

Runtime enforcement has been extensively studied over the last decade in the context of untimed properties. According to how a monitor is allowed to correct the input sequence, several models of enforcement monitors have been proposed. Security automata [Sch00] focused on safety properties, and blocked the execution as soon as an illegal sequence of actions (not compliant with the property) is recognized. Later, several refinements have been proposed such as suppression automata [LBW09] that allowed to suppress events from the input sequence, insertion automata [LBW09] that allowed to insert events to the input sequence, and edit-automata [LBW09] or so-called generalized enforcement monitors [FMFR11] allowed to perform any of these primitives. The notion of time has been considered in previous runtime enforcement approaches such as in [Mat07] for discrete-time properties, and in [BJKZ13] which considers elapsing of time as a series of uncontrollable events ("ticks").

2.2 Problem Statement

Motivations for extending runtime enforcement to timed properties abound. First, timed properties are more precise to specify desired behaviors of systems since they allow to explicitly state how time should elapse between events. Thus, timed properties/specifications can be particularly useful in some application domains [PFJM14a]. 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.

Context and objectives. The general context is depicted in Fig. 2.1. We focus on *online enforcement of timed properties*. More specifically, given a timed property φ, we want to synthesize an enforcement mechanism that operates at runtime. To be as general as possible, an enforcement mechanism is supposed to be placed between an event emitter and an event receiver. The emitter and receiver execute asynchronously. This abstract architecture is generic and can be instantiated to many concrete ones where the emitter and receiver are considered to be e.g., a program or the environment.

An enforcement mechanism receives a sequence of timed events σ as input and transforms it into a sequence of timed events o. No constraint is required on σ, whereas the enforcement mechanism must ensure that o is correct with respect to property φ. Satisfaction of property φ by the output sequence is considered at the output of the enforcement mechanism and not at the input of the event receiver: we assume a reliable, without delay, and safe communication between the emitter and receiver. We do not consider any security, communication, nor reliability issue with events. The considered enforcement mechanisms are *time retardants*, i.e., their main enforcement primitive consists in delaying the received events.

To sum up, given some timed property φ and an input timed word σ, we aim to study mechanisms that input σ and output a sequence o that satisfies φ (soundness of the mechanism).

Various directions that we want to explore are:

- **Expressiveness of the supported specification formalism** A central concept in runtime verification and enforcement, is to generate monitors from some high-level specification of the property (which the monitor should verify or enforce). In our enforcement framework, we wanted to handle (formalize and synthesize enforcement monitors) all the specifications that we commonly encounter in real-life in various domains.
- **Power of the enforcement mechanism** Another interesting aspect is regarding the power of the enforcement mechanism. What can the enforcement mechanism do to the input sequence in order to correct it with respect to the property? How the notion of transparency can be adapted in our framework also depends on the power of the enforcement mechanism. Enforcement operations that require expensive computations may not be desirable in a timed context.
- **Implementability** We also wanted to investigate the feasibility of realizing the enforcement monitoring mechanism that we propose. We want to see whether it is feasible to synthesize enforcement monitors based on the proposed formal framework, and also to have a first assessment of performance.

2.3 Contributions

This work resulted in publications in international conferences and journals (See Appendix B for the list of publications). We now summarize key contributions and how our work evolved.

- **Runtime Enforcement of Timed Properties.**
  We initially started with limited properties, and considered runtime enforcement of timed safety and co-safety properties. Safety properties express that "something bad should never happen" and co-safety properties express that "something good should happen within a finite amount of time". The results related to enforcement of safety and co-safety properties were published in RV 2012 [PFJ+12], introducing runtime enforcement framework for timed properties. We showed how runtime enforcers can be synthesized for any safety or co-safety timed property

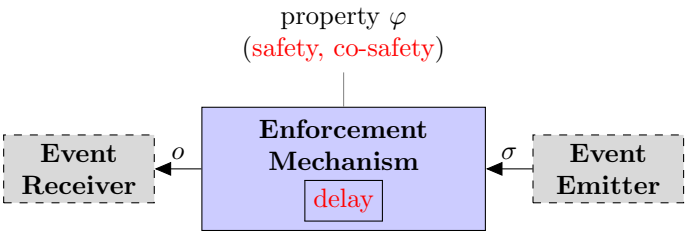


Figure 2.2: Supported properties and operations in [PFJ+12].

defined as timed automaton. Proposed runtime enforcers are time retardant: to correct an input sequence the enforcement mechanism has the power to introduce additional delays between the events of the input sequence. Output of the enforcement mechanism o satisfies φ (soundness of the mechanism), and has the same order of events as σ with possibly increased delays (transparency of the mechanism). Experiments have been performed on prototype monitors to show their effectiveness and the feasibility of our approach.

– **Runtime Enforcement of Regular Timed Properties.**

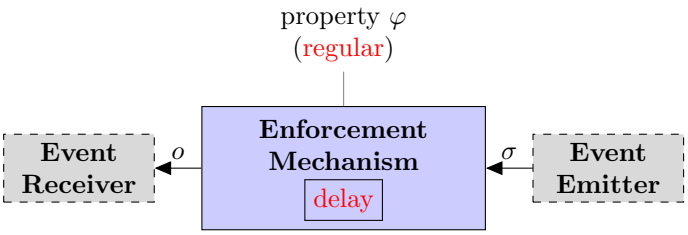


Figure 2.3: Supported properties and operations in [PFJM14b].

We later focused on supporting more properties in our enforcement framework introduced in [PFJ+12]. In the space of regular properties, some interesting properties of systems are neither safety nor co-safety properties. Indeed, some regular properties may express interesting properties of systems belonging to a larger class that allows to specify some form of transactional behavior. For example, the two following properties could specify the behavior of a server, and are neither safety nor co-safety properties.

– "Resource grants and releases alternate, starting with a grant, and every grant should be released within 15 to 20 time units (t.u.)".
– "Every 10 t.u., there should be a request for a resource followed by a grant. The request should occur within 5 t.u.".

The difficulty that arises when considering such properties is that the enforcement mechanisms should then consider the alternation between currently satisfying and not satisfying the property. The results in [PFJ+12] have further been extended to the enforcement of any regular timed property [PFJM14b]. We proposed enforcement monitor synthesis mechanisms for all regular timed properties. Also, for the enforcement of co-safety properties, the approach in [PFJ+12] assumes

that time elapses differently for input and output sequences (the sequences are desynchronized). More precisely, the delay of the first event of the output sequence is computed from the moment an enforcement mechanism detects that its input sequence can be corrected (that is, the mechanism has read a sequence that can be delayed into a correct sequence). Compared to [PFJ+12], the approach in [PFJM14b] is more realistic as it does not suffer from this "shift" problem.

– **Runtime Enforcement of Timed Properties revisited.**
We later revisited, combined the results in [PFJ+12] and [PFJM14b], simplified the formalizations, provided more explanations, examples and detailed formal proofs of these results which has been published in the Formal Methods in System Design (FMSD) journal [PFJ+14].
More specifically, this paper provides the following additional contributions:
  – proposes a more complete and revised theoretical framework for runtime enforcement of timed properties: we have re-visited the notations, unified and simplified the main definitions;
  – proposes a completely new implementation of our EMs that offers better performance (compared to the ones in [PFJ+12]);
  – synthesizes and evaluates EMs for more properties on longer executions;
  – includes correctness proofs of the proposed mechanisms.
– **Runtime Enforcement of Regular Timed Properties by Suppressing and Delaying Events.**

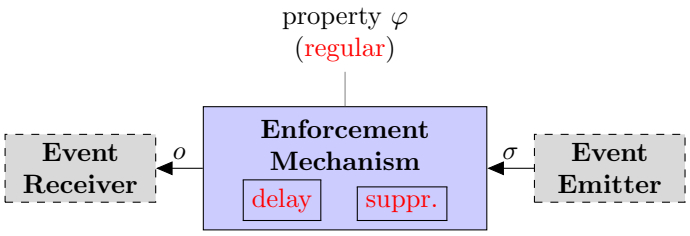


Figure 2.4: Supported properties and operations in [FJMP14].

In [PFJ+12, PFJM14b] the enforcement mechanism can only increase the delay between events. When it is no more possible to correct the input sequence anymore by introducing additional delay, the enforcement mechanism halts and cannot correct any more input events. So, later the focus was on increasing the power of the enforcement mechanism which can both delay events to match timing constraints, and suppress events when no delaying is appropriate, thus allowing the enforcement mechanisms and systems to continue executing. Also, in all our earlier works [PFJ+12, PFJM14b, PFJ+14], enforcement mechanisms receive sequences of events composed of actions and delays between them, and can only increase those delays to satisfy the desired timed property. Here, we consider events composed of actions with absolute occurrence dates, and we allow to increase the dates (while allowing to reduce delays between events) and to suppress events. This work has been submitted to the Science of Computer

Programming (SCP) journal [FJMP14].

– **Runtime Enforcement of Parametric Timed Properties with Practical Applications.**
When we considered requirements from some application domains such as network security, we noticed that some requirements have constraints both on time and data. Events also carry some data. To be able to formalize these requirements, that have constraints both on data and time, we felt the necessity to enrich Timed Automata (TA), the model we use to formally define requirements, with some additional features. 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 enforcement monitoring for parametric timed specifications. To handle expressive specifications in our framework, we introduce the model of Parameterized Timed Automata with Variables (PTAVs). To guide us in the choice of expressiveness features we considered requirements in several application domains. 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. These results were published in WODES 2014 [PFJM14a].

– **Prototype Implementations.**
To show the practical feasibility of our theoretical results, the proposed algorithms were also implemented in Python using some UPPAAL [BY03] libraries. The tool set developed is described in Chapter 6. In addition to enforcement monitor synthesis, the tool also provides other functionalities such as combining timed automata using Boolean operations, and determining the class of a given timed automaton.

2.4 Outline

This thesis is organized as follows:

Chapter 3 provides overview of various formal techniques related to checking the correctness of a system. It also describes and surveys related work on runtime enforcement techniques.

Chapter 4 provides the required background information. We describe all the preliminaries for formalizing timed properties and executions of a system. Timed automata are the formal model used to define properties. An enforcement monitor for a timed property is synthesized from a timed automaton defining the property. We describe the syntax and semantics of timed automata model, how timed automata can be combined using Boolean operations, and reachability analysis techniques for timed automata.

Chapter 5 presents runtime enforcement mechanisms for timed properties. We describe enforcement monitoring for systems where the physical time elapsing between actions matters. Executions are modeled as sequences of events composed of actions with dates (or timed words) and we consider runtime enforcement for timed specifi-

cations modeled as timed automata, in the general case of regular timed properties. The enforcement mechanisms have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus allowing the enforcement mechanisms and systems to continue executing. To ease their design and their correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input-output behavior in terms of transformations of timed words, constraints that should be satisfied by such functions, and enforcement monitors that describe the operational behavior of enforcement functions.

Chapter 6 presents enforcement algorithms describing the implementation of enforcement monitors described in Chapter 5. We validated the feasibility of enforcement monitoring for timed properties by prototyping the synthesis of enforcement monitors from timed automata. We describe the developed tool-chain, and discuss the experimental results in detail.

Chapter 7 presents runtime enforcement of parametric timed properties. We describe an extended model of timed automata, that can be used to formalize richer requirements (which have constraints both on time and data). We also extend our enforcement monitoring framework described in Chapter 5 to take in to account properties defined using the proposed extended model. Chapter 8 provides conclusions and outlines the future work.

Chapter 3

State of the Art

In this chapter, we first provide a short overview of various techniques, related to checking the correctness of a system (that is checking whether the system satisfies the requirements). Later, we describe runtime enforcement techniques, which is related to correcting the execution of a system (at runtime) according to the requirements.

3.1 Checking Correctness of a System

Generally, requirements are informal description of what we want a system to do. Requirements are the basis for the design, development and testing of a system.

The process of checking the correctness of a system (that is, whether the system conforms to the specification) is called as verification. The specification of the system, which is a set of requirements, is a description of the system (about what it should do/ how it should behave). Generally, verification is done using techniques such as testing.

The process of executing a program or a system with the objective of finding errors in the system is called as testing. Testing a system involves experimentally and systematically checking the correctness of the system. Generally, testing is performed by executing the system in a controlled environment, by providing some specific test data as input to the system, and making observations during the execution of the tests. From the observations made during the execution, verdicts about the correctness of the system (whether it satisfies the specification) is assigned.

For the testing activities, the specification of the system acts as the correctness criterion against which the system is to be tested. A common practice is to write the specification informally, in natural language. Such informal specification is the basis for any testing activity.

Drawbacks of the traditional testing process and using informal specification. Although informal specifications (written in some natural language such as English and French) are easy to read, they may be ambiguous. Informal description of requirements are often incomplete and liable to different and possibly inconsistent interpretations. Relying on unclear, imprecise, incomplete and ambiguous specifications

cause several problems in the testing process. Basing testing and verification activities on such informal specifications (which may have some inconsistencies) is not appropriate. Also, most of the testing activities are often incomplete, and they are used as a process to find errors, but cannot assure that the system is correct.

Formal specification and formal methods. A specification is said to be formal if it is expressed in a formal specification language which is a language having a precise syntax and semantics, and for which every statement in the language has a unique meaning mathematically. Such formal specification languages are often based on mathematical concepts such as mathematical logic. Advantage of such formal specifications and models is that they have a precise, unambiguous semantics, which enables the analysis of systems and the reasoning about them with mathematical precision.

Thus, using formal languages for specifying requirements will remove ambiguities and inconsistencies. Specifications described using formal languages, just as informal specifications, can be used in various development phases, and as contracts between developers and customers. Moreover, formal languages are more easily amenable to automatic processing, by means of tools. In contrast to informal specifications, formal specifications can be mathematically analysed since they are mathematically based. For example, methods (and tools) exist to automatically verify the absence of deadlock in the formal description of the design of the system. Formal specifications can also be used for generating test cases automatically.

3.2 Formal Verification Techniques

Verification is the act of checking whether a system/product satisfies (conforms to) its specification. When formal languages and techniques (based on mathematics) are used in the verification process it is called as formal verification. Formal verification techniques can be divided into two sub-categories called as static and dynamic verification techniques.

3.2.1 Static verification techniques

The static verification techniques are performed without actually executing the system being verified. Some of the static formal verification techniques are the following:

3.2.1.1 Model checking

Model checking [CE82, QS82, BK08] is an automated approach to verify that a formal model of the system (usually described as a state transition system), satisfies the formal specification, defining the requirements of the system. Model of the system is an abstraction of the system (omitting details irrelevant for checking the desired properties) that describes how the state of the system may evolve over time. Models are typically described as automata (or its extensions with time, probability, cost, data etc), describing the possible states, initial states, and the possible transitions

between states. Properties are usually expressed in some form of temporal logic [HR04] such as Linear Temporal Logic (LTL) [Pnu77, HR04], and Computational Tree Logic (CTL) [CE82, HR04], having constructs to express constraints on how the state of a system may evolve. Model checkers such as Spin [Hol97] and UPPAAL [LPY97], are some of the available tools that perform model checking.

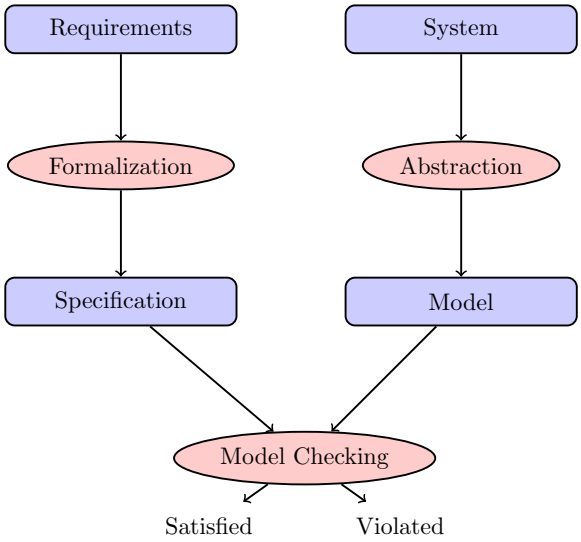


Figure 3.1: Model checking.

Figure 3.1 shows the process of applying model checking. To perform model checking on a system, a model of the system should be created, and the requirements (describing the desired properties of the system informally) should be formalized. The formalized requirements are called as properties (or property specification). The model checking algorithms (and tools), take the model and properties as input, and answer whether the model satisfies the properties (or not). The model checker explores all the possible states to decide whether the properties hold in all the (reachable) states. Incase if a property does not hold, then the model checker returns a counter example (which is a run of the model of the system, leading to a state in which the property does not hold).

3.2.1.2 Static analysis

Static analysis is a powerful technique that enables automatic verification of programs. Using static analysis approach, various properties such as type safety and resource consumption can be checked [HBB+11]. Abstract static analysis [DKW08] techniques which are mostly used for compiler optimization, are also used for program verification. These techniques focus on computing approximate *sound* guarantees efficiently. However, the information provided is not always precise since it is usually based on an approximation.

Generally, static analysis tools such as [EL02, HBB+11] parse the source code and build an abstraction (model) of the system such as a directed graph. By traversing the

model, it is checked whether certain properties hold in the model. In case a violation is found in the model, it can also be expected in the source code.

3.2.1.3 Theorem proving

In theorem proving approaches, for a mathematical statement to be true, a convening mathematical proof is constructed. If a proof cannot be found for a mathematical statement, then it cannot be concluded that the statement is true. In case a correct proof is found, the the statement is called a theorem. Several tools exist to help in the process of constructing formal proofs. For example interactive theorem provers such as COQ [BC04] are tools that can be use to construct a proof interactively. Proof checkers such as MetaMath [Met] can be used to check whether a proof is correct or not.

3.2.2 Dynamic verification techniques

Dynamic verification techniques are performed by executing the system in a particular environment, providing specific input data to the system and observing its behavior (or output).

3.2.2.1 Testing using formal methods

Several problems occur in the testing process since the specifications (which are written informally) are often incomplete or ambiguous. Basing testing activities on a formal specification is an advantage since formal specification is precise, complete, consistent and unambiguous. This is a first big advantage in contrast with traditional testing processes where such a basis for testing is often lacking.

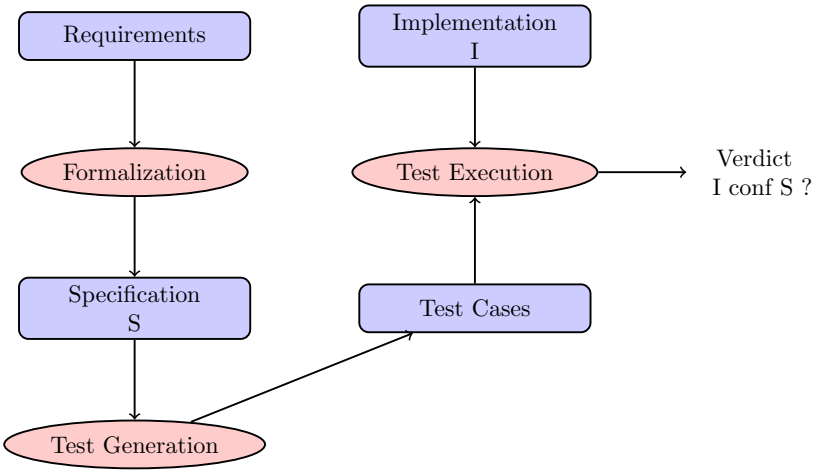


Figure 3.2: Conformance testing with formal methods.

Checking functional correctness of a black-box system under test by means of testing is known as conformance testing [Jan02, JJ05]. Formal specifications are also amenable

to automatic processing by tools, which is another advantage of formal specifications for testing. For conformance testing with formal methods, a formal specification is the starting point for the generation of test cases [Jan02]. Figure 3.2 gives an overview of the process. Algorithms (which have sound formal basis) have been developed for automatically generating test cases from a formal specification. This opens the way towards completely automating testing, where the system under test and its formal specification are the only required prerequisites. Moreover, they have been implemented in tools such as TGV [JJ05], STG [CJRZ02], and TorX [BB05], leading to automatic, faster and less error-prone generation of test cases. STG avoids enumerating the specification's state space, and uses symbolic generation techniques. Regarding conformance testing of real-time systems, there are tools such as TTG [KT09] based on the model of partially-observable, non-deterministic timed automata.

3.2.2.2 Runtime verification

Runtime verification [BLS11, FZ12] refers to the theories, techniques, and tools that allow checking whether a run of a system under scrutiny satisfies (or violates) a given correctness property. It is a formal verification technique, complementing the other formal verification techniques discussed so far such as model checking.

Runtime verification techniques do not influence the program execution, and deals only with detection of violation (or satisfaction) of properties. In runtime verification, checking whether a run of a system satisfies a property is performed using a monitor.

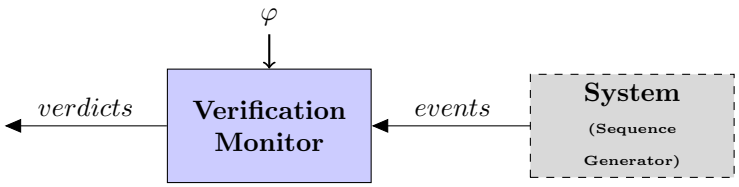


Figure 3.3: Verification monitor.

The first step of 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 verification monitor as shown in Figure 3.3 is a decision procedure emitting verdicts stating the correctness of the (partial) observed trace generated from the system execution. The system being monitored is regarded as a sequence generator. The set of verdicts may contain more than two different truth values such as $\{true, false, inconclusive\}$. The verdict provided by the monitor is *true* if the current input fulfills the property (irrespective of how the current input is extended), $false$ if a misbehavior is detected, and *inconclusive* otherwise. Formally, if φ denotes the property (the set of valid runs), then runtime verification is about checking whether a run of a system belongs to the property φ. 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/sequence of system events/actions. A central concept in runtime verification, is to generate monitors from some high-level specification of

the property which the monitor should check/verify. Monitors are generated from specifications such as Linear Temporal Logic (LTL) [BLS11].

Relation with other formal verification techniques In model checking, all executions of a given model of a system are examined, to answer whether the model satisfies the property, and in runtime verification, we check whether a particular execution of a system satisfies the property. Runtime verification deals only with observed executions of systems (generated by the real system), and does not require a model of the system. Runtime verification can be applied to black box system (that is, with knowledge only about the interfaces of the system, without the details of its internals). But, to perform model checking, we need to know all the internal details of a system in order to build an abstract model of the system. The model checking approach also suffers from the state explosion problem [BK08], since the whole state space of the model of the system (all possible executions) is generated. Runtime verification techniques are lightweight, avoiding problems such as state explosion, since only a single execution of the system is considered. Runtime verification can be considered as a form of passive testing, since it does not require to create any special test data, and require only to observe the results/behavior of a running system. Thus, runtime verification can be used in complement to other verification techniques when the system model is too big to handle with model checking, or when the system model is not available, or if the internal details of the system are unknown (that is, when the system is a black-box where only the outputs are observable).

In runtime verification, the complexity of generating the monitor from the specification is generally negligible (since the monitor is generated once, and offline). The complexity of the monitor (the memory it requires and the time it takes), for checking an execution is of interest, since runtime monitors are typically a part of an executing system, and should influence the system minimally.

A brief overview of some runtime verification frameworks 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 property may rule the ordering of function calls in a program. Monitoring such kind of properties has received a lot of attention [BLS11].

Parametric approaches have received a growing interest in the last five years. In this case, events in the property are augmented with formal parameters, instantiated at runtime [CR09, BFH+12].

In timed approaches, the observed time between events may influence the truth-value of the property. It turns out that monitoring of timed properties (where time is continuous) is a much harder problem because of (at least) two reasons. First, modeling timed requirements requires a more complex formalism involving time as a continuous parameter. Second, when monitoring a timed property, the problem that arises is that the overhead induced by the monitor (i.e., the time spent executing the

monitoring code) influences the truth-value of the monitored property. Consequently, without assumptions and limitations on the computation performed by monitors, not much information can be gained from the verdicts produced by the monitor.

Few attempts have been made on monitoring systems with respect to timed properties. Bauer et al. propose an approach to runtime verify timed-bounded properties expressed in a variant of Timed Linear Temporal Logic (TLTL) [BLS11]. 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 states verdict $\top$ (resp. $\bot$) when there is no infinite timed continuation w such that $u \cdot w$ satisfies (resp. does not satisfy) φ. Another variant of LTL in a timed context is Metric Temporal Logic (MTL), a dense extension of LTL. Nickovic et al. [MNP06, NP10] propose 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 which works by rewriting of the monitored formula and study its complexity [TR05]. Basin et al. propose an improvement of the aforementioned approach with a better complexity but considering only the past fragment of MTL [BKZ11]. Most of the other works related to runtime verification of timed properties are tools such as AMT [NM07] and LARVA [CPS09a].

One of the important features of runtime verification, compared to the other verification techniques is that it is performed at runtime (analysing a system during its execution). Thus, in addition to detecting errors, runtime verification techniques can be used/extended to act whenever an error/incorrect behavior of a system is detected. The techniques developed for runtime verification are the basis for other techniques dealing with correcting the execution of system at runtime (discussed in the later section).

3.3 Correcting Execution of a System at Runtime

Runtime enforcement [Sch00, LBW09, Fal10, FMFR11] extends runtime verification and refers to the theories, techniques, and tools aiming at ensuring the conformance of the executions of systems under scrutiny with respect to some desired property. Runtime enforcement is a powerful technique to ensure that a running system satisfies 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 with a property.

In runtime enforcement (as shown in Figure 3.4), an enforcement monitor (EM) transforms some (possibly) incorrect execution sequence into a correct sequence with respect to the property of interest. An enforcement monitor acts as a filter on some observable behavior of the system. The transformation performed by an EM should be *sound* and *transparent*. Soundness means that the resulting sequence obeys the property. Transparency (in an untimed setting) means that, the monitor should modify the input sequence in a minimal way (meaning that the input sequence should not be modified if it already conforms to the property).

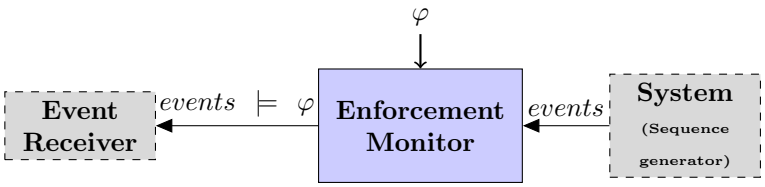


Figure 3.4: Enforcement mechanism.

Both verification and enforcement monitors (as shown in Figures 3.4, and 3.3) take the same data as input such as the property, and the input events from the system being monitored. They are plugged at the exit (or entrance) of the system, and they do not modify the behavior of the system. However, the goals of these techniques differ, where the aim of runtime verification techniques is to detect misbehaviors (or acknowledge desired behaviour), and thus a verification monitor outputs a verdicts providing this information. The main aim of runtime enforcement techniques is to prevent misbehaviors at runtime, and thus an enforcement monitor outputs a stream of events (eventually) satisfying the property.

3.3.1 Runtime enforcement of untimed properties

Over the last decade, runtime enforcement has been mainly studied in the context of untimed properties. Most of the work in runtime enforcement was dedicated to untimed properties (see [Fal10] for a short overview).

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 [Sch00, LBW09, Fal10, FMFR11].

An enforcement monitor that can definitely block the input sequence (as done by security automata [Sch00] introduced by Schneider), is the first runtime mechanism for enforcing safety properties. Then the set of enforceable properties was later refined by Schneider, Hamlen, and Morrisett by showing that security automata were actually restrained by the computational limits exhibited by Viswanathan and Kim [VK04]: the set of co-recursively enumerable safety properties is a strict upper limit of the power of (execution) enforcement monitors defined as security automata.

Enforcement mechanisms were proposed which can suppress events from the input sequence (as done by suppression automata [LBW09]), insert an event to the input sequence (as done by insertion automata [LBW09]).

Ligatti et al. [LBW09] later introduced edit-automata as enforcement monitors which can either insert a new action by replacing the current input, or suppress it. The set of properties enforced by edit-automata is called the set of infinite renewal properties: it is a super-set of safety properties and contains some liveness properties (but not all). Similar to edit-automata are generic enforcement monitors [FMFR11] which are able to enforce the set of (untimed) response regular properties in the safety-progress classification.

3.3.2 Runtime enforcement of timed properties

The notion of time has been considered in previous runtime enforcement approaches such as in [Mat07] for discrete-time properties, and in [BJKZ13] which considers elapsing of time as a series of uncontrollable events ("ticks").

Matteucci inspires from partial-model checking techniques to synthesize controller operations to enforce safety and information-flow properties using process-algebra [Mat07]. Monitors are close to Schneider's security automata [Sch00]. The approach targets discrete-time properties and systems are modeled 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.

Besides, in a general study, Rinard discusses monitoring and enforcement strategies for real-time systems [Rin03], and mentions the fact that enforcement mechanisms could delay individual input events in an input stream when they arrive too early w.r.t. the constraints of the system. In the same way, we consider in our work that an enforcer is time retardant. However, the work in [Rin03] remains at a high-level of abstraction and does not propose any detailed description of enforcement mechanisms.

More recently, Basin et al. [BJKZ13] proposed a general approach related to enforcement of security policies with controllable and uncontrollable events, investigating enforceability (with complexity results), and how to synthesize enforcement mechanisms for several specification formalisms (automata-based or logic-based). A monitor observes the system and terminates it to prevent violations. Timed properties described in MLTL logic are handled in this work. Discrete time is considered, clock ticks are used to determine the enforceability of an MLTL formula. In our approach, we consider dense time, using the expressiveness of timed automata and efficiency of UPPAAL. Moreover, our enforcement mechanisms may modify the execution of the observed system.

Most of the other works related to timed properties are either related to verification of timed properties (by synthesizing verification monitors from high level specifications), or tools for runtime monitoring of timed properties.

Tools for runtime monitoring of timed properties. The Analog Monitoring Tool [NM07] 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).

RT-MaC [SLS05] is a tool for verifying timed properties at runtime. RT-MaC allows to verify timeliness and reliability correctness. Using the time-bound temporal operators provided by the tool, one can specify a deadline after which a property must hold.

LARVA [CPS09a] 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 the duration calculus, called counter-examples traces, where properties are insensitive to monitoring [CPS09b].

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).

3.3.3 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 [CR09], 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.

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 [CR09]. Distinguishing parameters from external variables (parameters yield new instances of monitors while external variables get rebound) was first introduced in [BFH+12] with Quantified Event Automata (QEAs). QEAs feature a general use of quantifiers over parameters but do not consider time.

3.4 Summary

So far, we have briefly seen various formal verification techniques. We also saw what runtime enforcement means, and briefly the work done so far in this area. The rest of this thesis will focus only on runtime enforcement for timed properties. We present how enforcement monitors can be synthesized from properties defined as timed automata. In the next chapter we will see all the required background information about the formal model we use to express properties (timed automata), and summary of verification techniques for timed automata.

Chapter 4

Notations and Background

In this chapter, we first provide a brief description about requirements having time constraints (timed requirements) via some examples. Later, we present the preliminaries required to formally define timed requirements and executions (traces) of a system. Timed automata are the formal model used to define properties. An enforcement monitor for a timed property is synthesized from a timed automaton defining the property. We describe the syntax and semantics of timed automata model. Moreover, we also explain how timed automata can be combined using Boolean operations, and reachability analysis techniques for timed automata.

4.1 Timed Systems, and Requirements with Time Constraints

For real-time systems, the time at which the output is produced is significant. The correctness of a real-time systems depends not only on the logical result of the computation but also on the time at which the results are produced. If the correctness of a system is based both on the correctness of the outputs and the time at which the outputs are produced, then the system is called as a real-time system.

Real-time systems are categorized as soft real-time or hard real-time systems. In hard real-time systems, deadlines cannot be missed. In soft real-time systems, missing a deadline does not cause any harm, but may degrade quality or performance of the system. To specify the behavior of a real-time system, we need to take time into account.

Explicit timing constraints are present in several systems in real-life. Such systems have requirements with constraints over time (where we need to be precise about how time elapses between events). Considering time when specifying the behavior of systems brings some expressiveness that can be particularly useful in some application domains when, for instance, specifying usage of resources.

Examples of requirements with time constraints Let us consider the situation where two processes access to and operate on a common resource. Each process i

(with $i \in \{1, 2\}$) has three interactions with the resource: acquisition (acq_i), release (rel_i), and a specific operation (op_i). Both processes can also execute a common action op. System initialization is denoted by action $init$. The following requirements could specify the expected behavior of a server. [1]

S1 "*Each process should acquire first and then release the resource when performing operations on it. Each process should keep the resource for at least 10 time units (t.u). There should be at least 1 t.u. between any two operations.*"

S2 "*After system initialization, both processes should perform an operation (actions op_i) before 10 t.u. The operations of the different processes should be separated by 3 t.u.*"

S3 "*Operations op_1 and op_2 should execute in a transactional manner. Both actions should be executed, in any order, and any transaction should contain one occurrence of op_1 and op_2. Each transaction should complete within 10 t.u. Between operations op_1 and op_2, occurrences of operation op can occur. There is at least 2 t.u. between any two occurrences of any operation.*"

S4 "*Processes should behave in a transactional manner, where each transaction consists of an acquisition of the resource, at least one operation on it, and then a release of it. After the acquisition of the resource, the operations on the resource should be done within 10 t.u. The resource should not be released less than 10 t.u. after acquisition. There should be no more than 10 t.u. without any ongoing transaction.*"

Formalization of a requirement with time constraints is called as a timed property. A timed automaton [AD94] is a finite automaton extended with a finite set of real valued clocks. It is one of the most studied models for modeling verification of real-time systems. Timed properties can be defined as timed automata. Various algorithms and tools exist related to verification of timed automata, which is the main advantage of defining timed properties as timed automata.

Outline The rest of this chapter is organized as follows: Firstly, in Section 4.2.1 notations and preliminaries are introduced related to formalizing requirements in the untimed case, and later in Section 4.2.2 these notations are lifted to the timed case for defined timed traces and timed languages. In Section 4.3, timed automata model is explained describing its syntax and semantics, how timed properties can be defined using timed automata is explained via some examples, a classification of timed properties is described, and how timed automata can be combined using union and intersection operations is also presented. Finally, techniques and algorithms related to the verification of timed automata are discussed briefly.

4.2 Preliminaries and Notations

Firstly, basic preliminaries and notations related to formalizing requirements without timing constraints are presented in Section 4.2.1. Later in Section 4.2.2, the nota-

1. We shall see in Section 4.2.2 how to formalize and define these specifications by timed automata.

tions in the untimed case are lifted to the timed setting.

4.2.1 Untimed languages

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

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

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

Given two words u and v, $v^{-1} \cdot u$ is the *residual* of u by v and denotes the word w, such that $v \cdot w = u$, if this word exists, i.e., if v is a prefix of u. Intuitively, $v^{-1} \cdot u$ is the suffix of u after reading prefix v. By extension, for a language $\mathcal{L} \subseteq A^*$ and a word $v \in A^*$, the residual of $\mathcal{L}$ by u is the language $v^{-1} \cdot \mathcal{L} \stackrel{\text{def}}{=} \{w \in A^* \mid v \cdot w \in \mathcal{L}\}$. It is the set of suffixes of words that, concatenated to v, belong to $\mathcal{L}$. In other words, $v^{-1} \cdot \mathcal{L}$ is the set of suffixes of words in $\mathcal{L}$ after reading prefix v.

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

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

Given an n-tuple of symbols $e = (e_1, \ldots, e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$).

Example 4.1 *Let us illustrate these definitions via an example. Let $A = \{a, b, c\}$ be a set of actions. $w_1 = a \cdot c \cdot a$ is a finite word over A. The set of all words over elements of A which start with action b is a language over A. $w_{1[2]} = c$ is the second letter in w_1, and $w_{1[2 \cdots 3]} = c \cdot a$. Consider another finite word $w_2 = a \cdot c$. Concatenation of w_1 and w_2 is $w_1 \cdot w_2 = a \cdot c \cdot a \cdot a \cdot c$, w_2 is a prefix of word w_1, $w_2^{-1} \cdot w_1 = a$, and word $a \cdot a$ is a subsequence of w_1.*

4.2.2 Timed words and languages

In a timed setting, the occurrence time of actions is also important. For an enforcement monitor in a timed setting, input and output streams are seen as sequences of events composed of a date and an action, where the date is interpreted as the absolute

time when the action is received by the enforcement mechanism [2]. In what follows, input and output streams are formalized with timed words, and some related notions are defined.

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

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

The set of timed words over Σ is denoted by $\text{tw}(\Sigma)$. A *timed language* is any set $\mathcal{L} \subseteq \text{tw}(\Sigma)$. Note that even though the alphabet $(\mathbb{R}_{\geq 0} \times \Sigma)$ is infinite in this case, previous notions and notations defined in the untimed case (related to length, prefix, subword, subsequence etc) naturally extend to timed words.

Concatenation of timed words however requires more attention, as when concatenating two timed words, one should ensure that the result is a timed word, i.e., dates should be non-decreasing. This is ensured if the ending date of the first timed word does not exceed the starting date of the second one. Formally, let $\sigma = (t_1, a_1) \cdots (t_n, a_n)$ and $\sigma' = (t'_1, a'_1) \cdots (t'_m, a'_m)$ be two timed words with $\text{end}(\sigma) \leq \text{start}(\sigma')$, their concatenation is $\sigma \cdot \sigma' \stackrel{\text{def}}{=} (t_1, a_1) \cdots (t_n, a_n) \cdot (t'_1, a'_1) \cdots (t'_m, a'_m)$. By convention $\sigma \cdot \epsilon \stackrel{\text{def}}{=} \epsilon \cdot \sigma \stackrel{\text{def}}{=} \sigma$. Concatenation is undefined otherwise.

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

Example 4.2 *Let us understand these definitions further via an example. Consider a set of actions* $\Sigma = \{a, b, c\}$. $\sigma_1 = (1, a) \cdot (2.3, b) \cdot (3, a) \cdot (4, c)$ *is a timed word over* Σ. *Note that the occurrence dates of events in* σ *are increasing. Starting date of* σ_1, $\text{start}(\sigma_1) = 1$ *and, the ending date* $\text{end}(\sigma_1) = 4$. *The untimed projection of* σ_1, $\Pi_\Sigma(\sigma_1) = a \cdot b \cdot a \cdot c$. *Consider two more timed words over* Σ, $\sigma_2 = (2, b) \cdot (2.3, b) \cdot (3, a)$, *and* $\sigma_3 = (10, b) \cdot (12, a)$. *The concatenation* $\sigma_1 \cdot \sigma_2$ *is undefined since* $\text{start}(\sigma_2)$ *is lesser than* $\text{end}(\sigma_1)$. *The concatenation* $\sigma_1 \cdot \sigma_3 = (1, a) \cdot (2.3, b) \cdot (3, a) \cdot (4, c) \cdot (10, b) \cdot (12, a)$.

4.3 Timed Automata

A timed automaton [AD94] is a finite automaton extended with a finite set of real-valued clocks. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of *clocks*. A *clock valuation* for X is an element of $\mathbb{R}^X_{\geq 0}$, that is a function from X to $\mathbb{R}_{\geq 0}$. For $\chi \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\chi + \delta$ is the valuation assigning $\chi(x) + \delta$ to each clock x of X. Given a set of clocks $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the clock valuation χ where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of *guards*, i.e., clock constraints defined as Boolean combinations

2. Alternatively, input and output streams can be seen as sequence of events composed of a delay and an action, where the delay associated with each event indicates the time elapsed after the previous event or the system initialization for the first event.

of simple constraints of the form $x \bowtie c$ with $x \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and $\chi \in \mathbb{R}^X_{\geq 0}$, we write $\chi \models g$ when g holds according to χ.

4.3.1 Syntax and semantics

Before going into the formal definitions, we introduce timed automata on an example. The timed automaton in Figure 4.1 defines the requirement *"In every 10 time*

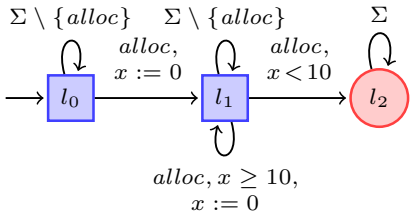


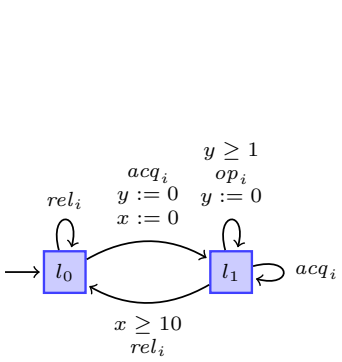
Figure 4.1: Timed automaton: Example

units, there cannot be more than 1 alloc action". The set of locations is $L = \{l_0, l_1, l_2\}$, and l_0 is the initial location. The set of actions is $\Sigma = \{alloc, rel\}$. There are transitions between locations upon actions. A finite set of real-valued clocks is used to model real-time behavior, the set $X = \{x\}$ in the example. On the transitions, there are i) guards with constraints on clock values (such as $x < 10$ on the transition between l_1 and l_2 in the considered example), and ii) assignment to clocks. Upon the first occurrence of action *alloc*, the automaton moves from l_0 to l_1, and the clock x is assigned to 0. In location l_1, if action *alloc* is received, and if $x \geq 10$, then the automaton remains in l_1, resetting the value of clock x to 0. It moves to location l_2 otherwise.

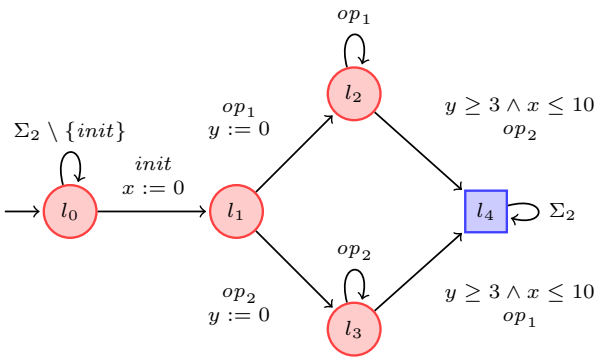
Definition 4.1 (Timed automata) *A* timed automaton *(TA) is a tuple* $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$*, such that* L *is a finite set of* locations *with* $l_0 \in L$ *the* initial location*,* X *is a finite set of* clocks*,* Σ *is a finite set of* actions*,* $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ *is the* transition relation*.* $F \subseteq L$ *is a set of* accepting locations*.*

The semantics of a timed automaton is defined as a transition system where each state consists of the current location and the current values of clocks. Since the possible values for a clock are infinite, a timed automaton has infinite states. The semantics of a TA is defined as follows.

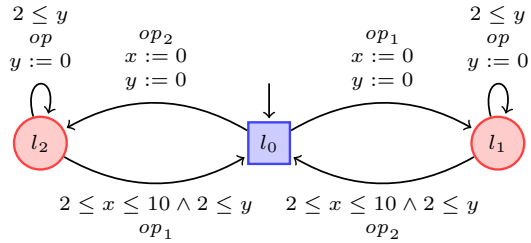
Definition 4.2 (Semantics of timed automata) *The* semantics *of a TA is a* timed transition system $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \rightarrow, Q_F)$ *where* $Q = L \times \mathbb{R}^X_{\geq 0}$ *is the (infinite) set of* states*,* $q_0 = (l_0, \chi_0)$ *is the* initial state *where* χ_0 *is the valuation that maps every clock in* X *to* 0*,* $Q_F = F \times \mathbb{R}^X_{\geq 0}$ *is the set of* accepting states*,* $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ *is the set of transition* labels*, i.e., pairs composed of a delay and an action. The* transition relation $\rightarrow \subseteq Q \times \Gamma \times Q$ *is a set of transitions of the form* $(l, \chi) \xrightarrow{(\delta, a)} (l', \chi')$ *with* $\chi' = (\chi + \delta)[Y \leftarrow 0]$ *whenever there exists* $(l, g, a, Y, l') \in \Delta$ *such that* $\chi + \delta \models g$ *for* $\delta \in \mathbb{R}_{\geq 0}$*.*



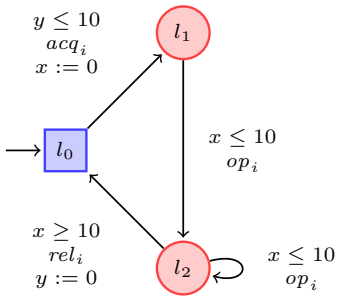
(a) A safety automaton for S_1.



(b) A co-safety automaton for S_2.



(c) A regular automaton for S_3.



(d) A regular automaton for S_4.

Figure 4.2: Some examples of timed automata.

In the following, we consider a timed automaton $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ with its semantics $[\![\mathcal{A}]\!]$. $\mathcal{A}$ is said to be *deterministic* whenever for any location l and any two distinct transitions (l, g_1, a, Y_1, l'_1) and (l, g_2, a, Y_2, l'_2) with source l in Δ, the conjunction of guards $g_1 \wedge g_2$ is unsatisfiable [3]. $\mathcal{A}$ is said *complete* whenever for any location $l \in L$ and any action $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labeled by a evaluates to *true* (i.e., it holds according to any valuation): $\forall l \in L, \forall a \in \Sigma : \bigvee_{(l,g,a,Y,l')} g = true$. In the remainder of this thesis, we shall consider only deterministic and complete timed automata, and automata refer to timed automata.

Example 4.3 (Timed automata) *Let us formalize the specifications introduced in Section 4.1 as TAs. The global alphabet of events is* $\Sigma \stackrel{\text{def}}{=} \{init, acq_1, rel_1, op_1, acq_2, rel_2, op_2, op\}$. *The specifications on the behavior of the processes introduced in Section 4.1 are defined with the TAs in Figure 4.2. Accepting locations are denoted by squares.*

S_1 The specification is defined by the automaton depicted in Figure 4.2a with al-

3. There is no valuation of clock variables that evaluate $g_1 \wedge g_2 = true$.

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

S_2 *The specification is defined by the automaton depicted in Figure 4.2b with alphabet* $\Sigma_2 \stackrel{\text{def}}{=} \{init, op_1, op_2\}$. *The automaton has two clocks, where clock* x *keeps track of the time elapsed since initialization, whereas clock* y *keeps track of the time elapsing between the operations of the two different processes.*

S_3 *The specification is defined by the automaton depicted in Figure 4.2c with alphabet* $\Sigma_3 \stackrel{\text{def}}{=} \{op, op_1, op_2\}$. *Clock* x *keeps track of the time elapsing since the beginning of the transaction, whereas clock* y *keeps track of the time elapsing between any two operations.*

S_4 *The specification is defined by the automaton depicted in Figure 4.2d with alphabet* $\Sigma_4^i \stackrel{\text{def}}{=} \{acq_i, op_i, rel_i\}$. *Clock* x *keeps track of the duration of a currently executing transaction, whereas clock* y *keeps track of the time elapsing between two transactions.*

Remark 4.1 (Completeness and determinism) *Although we restrict the presentation to deterministic TAs, results also hold for non-deterministic TAs, with slight adaptations required to the vocabulary and when synthesizing an enforcement monitor. Regarding completeness, for readability of TA examples, if no transition can be triggered upon the reception of an event, a TA implicitly moves to a non-accepting trap state (i.e., where all actions are looping with no timing constraint).*

A *run* ρ of $\mathcal{A}$ from a state $q \in Q$ triggered at time $t \in \mathbb{R}_{\geq 0}$ over a timed *trace* $w_t = (t_1, a_1) \cdot (t_2, a_2) \cdots (t_n, a_n)$ is a sequence of moves in $[\![\mathcal{A}]\!]$ denoted as $\rho_t = q \xrightarrow{(\delta_1, a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n, a_n)} q_n$, for some $n \in \mathbb{N}$, satisfying condition $t_1 = t + \delta_1$ and $\forall i \in [2, n], t_i = t_{i-1} + \delta_i$. To simplify notations, we note $q \xrightarrow{w_t} q_n$ in this case, and generalize it to $q \xrightarrow{w_t} P$ when $q_n \in P$ for a subset P of Q. The set of runs from the initial state $q_0 \in Q$, starting at $t = 0$ is denoted $\text{Run}(\mathcal{A})$ and $\text{Run}_{Q_F}(\mathcal{A})$ denotes the subset of those runs *accepted* by $\mathcal{A}$, i.e., ending in an accepting state $q_n \in Q_F$. We note $\mathcal{L}(\mathcal{A})$ the set of traces of $\text{Run}(\mathcal{A})$. We extend this notation to $\mathcal{L}_{Q_F}(\mathcal{A})$ as the traces of runs in $\text{Run}_{Q_F}(\mathcal{A})$. We thus say that a timed word is accepted by $\mathcal{A}$ if it is the trace of an accepted run.

Example 4.4 *Consider the automaton in Figure 4.2a. A run of this automaton triggered at* $t = 0$ *starting from the initial state* $(l_0, 0, 0)$ *over a timed trace* $w_t = (1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1)$ *is a sequence of moves* $(l_0, 0, 0) \xrightarrow{(1, acq_1)} (l_1, 0, 0) \xrightarrow{(2, op_1)} (l_0, 2, 0) \xrightarrow{(1, op_1)} (l_1, 3, 0) \xrightarrow{(0.5, acq_1)} (l_1, 3.5, 0.5) \xrightarrow{(0.5, op_1)} (l_1, 4, 0)$ *which is also concisely denoted as* $(l_0, 0, 0) \xrightarrow{w_t} (l_1, 4, 0)$.

4.3.2 Partition of states of timed automata

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

This partitioning is useful for enforcement monitoring. An enforcement monitor makes decisions by checking the reachable subsets. For example, if all the reachable states belong to the subset B, then there is no possibility to correct the input sequence anymore (in the future). If the current state belongs to the subset G, then any sequence will lead to a state belonging to the same subset and thus the monitor can be turned off.

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

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

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

It is well known that reachability of a set of locations is decidable using the classical zone (or region) symbolic representation (see [BY03]). As Q_F corresponds to all states with location F, the partition can then be symbolically computed on the zone graph. This partition will be useful to classify timed properties and for monitor synthesis.

Remark 4.2 *By definition, from good (resp. bad) states, one can only reach good (resp. bad) states. Consequently, a run of a TA traverses currently good and/or currently bad states, and may eventually reach a good state and remain in good states, or a bad state and remain in bad states.*

4.3.3 Classification of timed properties

Regular, safety, and co-safety timed properties In the sequel, a timed property is defined by a timed language $\varphi \subseteq \text{tw}(\Sigma)$ that can be recognized by a timed automaton. That is, the set of regular timed properties are considered. Given a timed word $\sigma \in \text{tw}(\Sigma)$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$.

Safety (resp. co-safety) timed properties are sub-classes of regular timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). In this thesis, the classes are characterized as follows:

Definition 4.3 (Regular, safety, and co-safety properties)

- *Regular properties are the properties that can be defined by languages accepted by a TA.*
- *Safety properties (a subset of regular properties) are the non-empty prefix-closed timed languages that can be accepted by a TA.*
- *Co-safety properties (a subset of regular properties) are the non-universal*[4] *extension-closed timed languages that can be accepted by a TA.*

Remark 4.3 *Usually, safety properties are defined as prefix-closed languages, and co-safety properties as extension-closed languages. With the usual definitions, properties* $\emptyset$ *and* $\mathrm{tw}(\Sigma)$ *(the empty and universal properties, respectively vacuously and universally satisfied) are both safety and co-safety properties, and are the only ones in the intersection. To simplify the presentation, and to avoid pathological cases, the two classes are separated, by considering that* $\mathrm{tw}(\Sigma)$ *is a safety (but not a co-safety) property, and* $\emptyset$ *is a co-safety (but not a safety) property.*

4.3.4 Defining timed properties as timed automata

In this subsection, details related to defining timed properties as timed automata are presented. Moreover, examples of some timed automata defining timed properties of different classes are presented. In the sequel, we shall only consider the properties that can be defined by a deterministic timed automaton. Note that some of these properties can be defined using a timed temporal logic such as a subclass of MTL, which can be transformed into timed automata using the technique described in [MNP06, NP10].

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

Definition 4.4 (Safety and co-safety TA) *Let* $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, F)$ *be a complete and deterministic TA, where* $F \subseteq L$ *is the set of accepting locations.* $\mathcal{A}$ *is said to be:*

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

It is then easy to check that safety (respectively co-safety) TAs define safety (respectively co-safety) properties. Intuitively, in a safety TA, runs start in an accepting location, but if they leave the set of accepting locations, it is definitive; thus a safety TA defines a prefix-closed language. Conversely, in a co-safety TA, a run starts in a non-accepting location, and once it reaches an accepting location, it is definitive; thus a co-safety TA defines an extension-closed language. [5]

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

5. As one can observe, these definitions of safety and co-safety TAs slightly differ from the usual ones by expressing constraints on the initial state. As a consequence of these constraints, consistently with Definition 4.3, the empty and universal properties are ruled out from the set of safety and co-safety properties, respectively.

Example 4.5 (Classes of timed automata) *Let us consider again the specifications introduced in Example 4.2. We formalize specification S_i as property φ_i, $i = 1, \ldots, 4$. Property φ_1 is a safety property specified by the safety TA in Figure 4.2a (leaving accepting locations is definitive). Property φ_2 is a co-safety property specified by the co-safety TA in Figure 4.2b (leaving non-accepting locations is definitive). Property φ_3 is specified by the TA in Figure 4.2c. Property φ_4 is specified by the TA in Figure 4.2d. Both properties φ_3 and φ_4 are regular, but neither safety nor co-safety properties. In the underlying automata, runs may alternate between accepting and non-accepting locations, thus the languages that they define are neither prefix nor extension-closed.*

Remark 4.4 *Alternatively, safety and co-safety timed automata could also be defined based on the semantics, using the partitioning of states defined in Section 4.3.2.*

4.3.5 Combining properties using boolean operations.

The next definition, as described in [AD94], allows to combine (complete and deterministic) timed automata and will be used in the sequel to define properties expressed as a Boolean combination of other properties.

Definition 4.5 (Operations on timed automata) *Given two properties φ_1 and φ_2 defined by TAs $\mathcal{A}_{\varphi_1} = (L_1, \ell_1^0, X_1, \Sigma, \Delta_1, G_1)$ and $\mathcal{A}_{\varphi_2} = (L_2, \ell_2^0, X_2, \Sigma, \Delta_2, G_2)$, respectively. The $\times_{op}$-product of $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$, where $op \in \{\cup, \cap\}$, is the timed automaton defined as $\mathcal{A}_{\varphi_1} \times_{op} \mathcal{A}_{\varphi_2} \stackrel{\text{def}}{=} (L, l_0, X, \Sigma, \Delta, G)$ where $L = L_1 \times L_2$, $l_0 = (l_0^1, l_0^2)$, $X = X_1 \cup X_2$ (disjoint union), $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the* transition relation, *where $((l_1, l_2), g_1 \wedge g_2, a, Y_1 \cup Y_2, (l_1', l_2')) \in \Delta$ iff $(l_1, g_1, a, Y_1, l_1') \in \Delta_1$ and $(l_2, g_2, a, Y_2, l_2') \in \Delta_2$. G_{op} is a set of accepting locations with:*
- $G_\cap = G_1 \times G_2$,
- $G_\cup = (L_1 \times G_2) \cup (G_1 \times L_2)$.

Definition 4.6 (Negation of a timed automaton) *Given a property φ defined by a TA $\mathcal{A}_\varphi = (L, \ell^0, X, \Sigma, \Delta, G)$ its negation is defined as*
$\neg\mathcal{A}_\varphi \stackrel{\text{def}}{=} (L, \ell^0, X, \Sigma, \Delta, L \setminus G)$.

The proposition below states that when performing the $\cap$-product (resp. $\cup$-product) between two TAs, it amounts to perform the intersection (resp. union) of the recognized languages.

Proposition 4.1 *Consider two properties φ_1 and φ_2 defined by TAs $\mathcal{A}_{\varphi_1} = (L_1, \ell_1^0, X_1, \Sigma, \Delta_1, G_1)$ and $\mathcal{A}_{\varphi_2} = (L_2, \ell_2^0, X_2, \Sigma, \Delta_2, G_2)$, respectively. The following facts hold:*
- $\mathcal{L}_{(\mathcal{A}_{\varphi_1} \times_\cap \mathcal{A}_{\varphi_2})(G_\cap)} = \varphi_1 \cap \varphi_2$,
- $\mathcal{L}_{(\mathcal{A}_{\varphi_1} \times_\cup \mathcal{A}_{\varphi_2})(G_\cup)} = \varphi_1 \cup \varphi_2$,
- $\mathcal{L}_{\neg\mathcal{A}_{\varphi_1}} = \text{tw}(\Sigma_1) \setminus \mathcal{L}_{(\mathcal{A}_{\varphi_1})}$,
- $\mathcal{L}_{\neg\mathcal{A}_{\varphi_2}} = \text{tw}(\Sigma_2) \setminus \mathcal{L}_{(\mathcal{A}_{\varphi_2})}$.

The above proposition entails that the classes of safety and co-safety properties are closed under union and intersection. However, the properties resulting of any other operation between safety, co-safety, and regular properties is a regular property. Finally, note that the negation of a safety (resp. co-safety) property is a co-safety (resp. safety) property. From the results shown in [AD94], the following proposition holds.

Proposition 4.2

- *Safety and co-safety properties are closed under (finite) union and intersection.*
- *The negation of a safety property is a co-safety property, and vice-versa.*
- *Regular properties are closed under Boolean operations.*

Example 4.6 (Intersection of two safety TAs) *Let us see an example of a resulting safety TA, obtained from combining two safety TA's using intersection operation. Figure 4.3 presents the two input TA's defining specifications $S1$ expressing that "There should be a delay of at least 5 time units between two 'a' actions", and $S2$ expressing that "There should be a delay of at least 6 time units between two 'b' actions". The set of actions of the two TA's is $\Sigma = \{a, b, c\}$.*

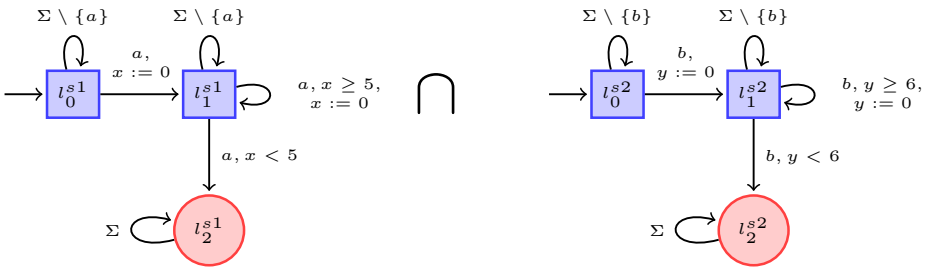


Figure 4.3: Example safety TAs

The resulting TA is shown in Figure 4.4 and defines the specification "There should be a delay of at least 5 time units between two 'a' actions and a delay of at least 6 time units between two 'b' actions". We can easily observe that this is a safety TA.

Example 4.7 (Union of two co-safety TAs) *An example of a co-safety TA obtained by performing union operation on two co-safety TA's is shown in Figure 4.6. Figure 4.5 presents the two input TA's defining specifications $CS1$ expressing that "A request r should be followed by a grant g within 10 time units" and $CS2$ expressing that "After an request r, if any other event other than g is observed, then the system should go into safe mode (perform an s action) within 10 time units". The set of actions of both TA's is $\Sigma = \{r, g, a, s\}$.*

The resulting TA is shown in Figure 4.6 expresses that "A request r should be followed by a grant g, otherwise the system should go into safe mode (perform an s action)". We can easily observe that the obtained TA is a co-safety TA.

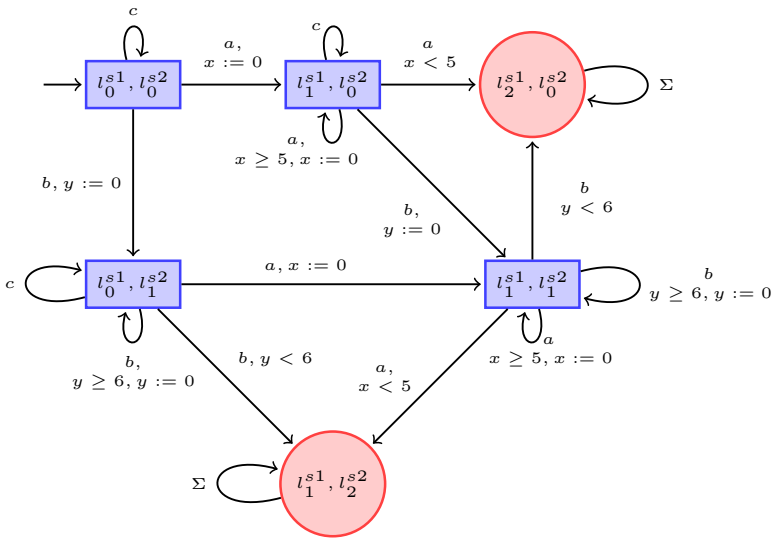


Figure 4.4: Intersection of two safety TAs

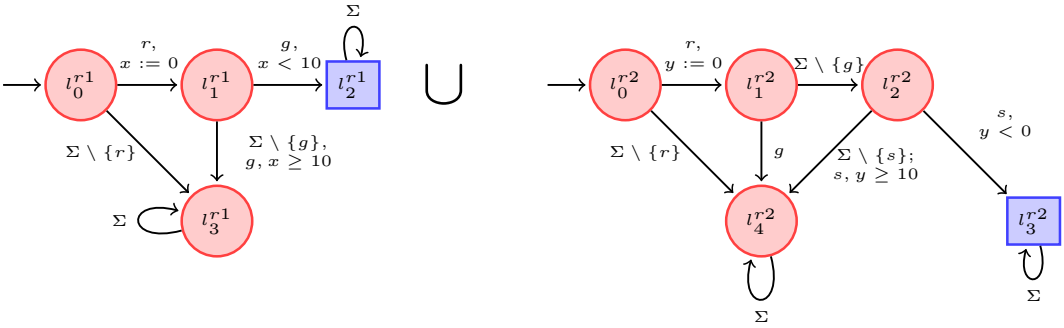


Figure 4.5: Example co-safety TAs

4.3.6 Verification of timed automata

For verification purposes, one should be able to verify reachability properties. That is, for timed automaton $\mathcal{A}$, we should be able to check *"Is state q (or a set of states) reachable?* For the class of finite-location automata, reachability is decidable [Bou, HMU03]. If we consider the semantics of a TA, a state of a TA q is a tuple containing the information about location and valuation of all the clocks. Thus, for a timed automaton, the number of states is infinite, which is problematic for reachability analysis.

Region abstraction and region automaton construction [AD94] is a way to abstract behaviours of timed automata, so that checking a reachability property in a timed automaton reduces to checking a reachability property in a finite automaton. The key idea is to partition the state-space into a finite number of equivalent classes, and to perform reachability analysis on finite abstract state-space. Though timed automata have an infinite number of states, using region automaton construction, checking reachability is decidable [AD94].

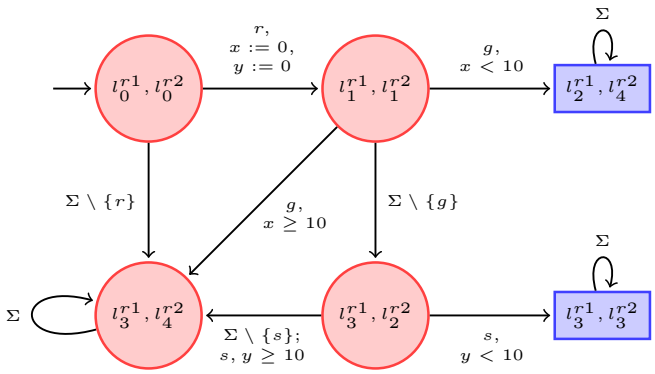


Figure 4.6: Resulting co-safety TA after performing union operation

Construction of region automaton, and proof that reachability is decidable for a timed automaton are well known results described in detail in [AD94]. The number of regions depends on the number of clocks and the maximal constant appearing in a guard or a invariant.

Algorithms and Implementation For reachability analysis of timed automata, region automaton construction is not used in practice. In tools such as UPPAAL [BY03] and Kronos [BDM+98], *on-the-fly* techniques are used for reachability analysis. Most of the tools use zone abstraction [BY03]. A zone is a convex union of regions and can be efficiently represented using difference bounded matrices (DBMs) [BY03]. Operations such as intersection of guards, reset of clocks and elapsing of time are all easily implementable in DBMs. More details about zones, and representing zones as DBMs are in [BY03].

Most of the algorithms for reachability analysis are based on either forward or backward analysis approaches [Bou09]. In forward analysis approaches, configurations that are reachable from the initial state in 1 step, 2 steps and so on are computed either until the desired (final) state is reached or until the computation terminates. The idea of backward analysis is to start from a final configuration, then compute the configurations which allow to reach an initial configuration in 1 step, 2 steps and so on until initial configurations are reached or until the computation terminates. These are generic approaches used not only for verification of timed automata, but also for many other models. Let us now see an example how reachability analysis can be performed on-the-fly using forward analysis, zone abstraction and operation on zones such as resetting clocks and intersection of guards.

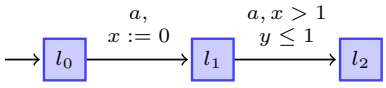


Figure 4.7: Example TA to demonstrate forward reachability analysis

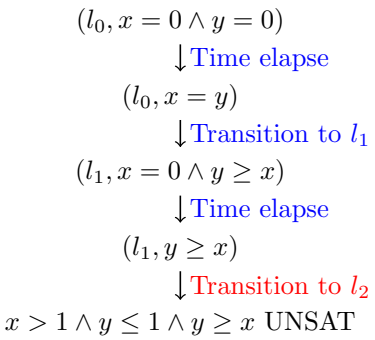


Figure 4.8: Forward reachability analysis

Example 4.8 (Forward reachability analysis) *Consider the TA shown in Figure 4.7. Using forward reachability analysis (see Figure 4.8), we can check that location l_2 is not reachable. The TA has two clocks x and y, which are initialized to 0. Initially, the TA is in location l_0. As time elapses, the location will remain unchanged, and the value of both the clocks change but will be equal (i.e., $x = y$). There is only one outgoing transition from location l_0 (to location l_1), and upon this transition, the clock x is reset to 0, and thus we have $x = 0 \wedge y \geq x$ upon this transition. As time elapses, we continue to remain in location l_1, value of clock x will no more be 0 and the condition $y \geq x$ still holds. There is a transition from l_1 to l_2, and the constraints on this transition when intersected with condition $y \geq x$ can not be satisfied. Thus, location l_2 is unreachable.*

4.4 Summary

In this chapter, we saw the required preliminaries to formally define requirements with time constraints, and an execution (trace) of a system. We also saw the syntax and semantics of timed automata, how timed properties can be defined as timed automata, how to combine timed automata, and techniques for reachability analysis for timed automata. In the next chapter, enforcement mechanism for timed properties will be explained in detail. We will see how to synthesize enforcement monitors from timed properties defined as timed automata.

Chapter 5

Runtime Enforcement of Timed Properties

In this chapter, we describe enforcement monitoring mechanisms for systems where the physical time elapsing between actions matters. Executions are thus modeled as sequences of events composed of actions with dates (or timed words). We consider runtime enforcement for timed specifications modeled as timed automata, in the general case of regular timed properties. The proposed enforcement mechanism has the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus allowing the enforcement mechanisms and systems to continue executing. To ease their design and correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input-output behavior in terms of transformations of timed words, constraints that should be satisfied by such functions, enforcement monitors that describe the operational behavior of enforcement functions, and enforcement algorithms that describe the implementation of enforcement monitors. The feasibility of enforcement monitoring for timed properties is validated by prototyping the synthesis of enforcement monitors from timed automata.

5.1 General Principles and Motivating Examples

In this section, the context, capabilities of an enforcement monitor, and the expected behavior of an enforcement monitor are described informally to understand the concepts before proceeding into the formalizations and details in the later sections.

5.1.1 General principles of enforcement monitoring in a timed context

As illustrated in Figure 5.1, the purpose of enforcement monitoring is to read some (possibly incorrect) input sequence of events σ produced by a system, referred to as the event emitter, and to transform it into an output sequence of events o that is correct w.r.t. a specification formalized by a property φ. This output sequence o is then provided as input to an event receiver. In our timed setting, events are actions with

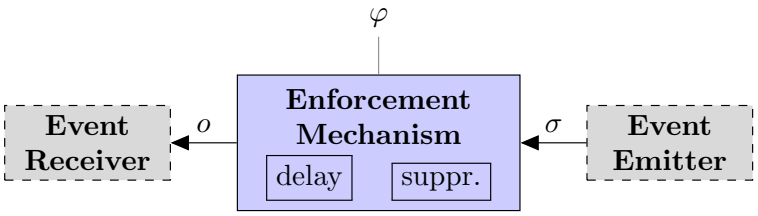


Figure 5.1: Illustration of the principle of enforcement monitoring.

their occurrence dates. Input and output sequences of events are formalized by timed words and enforcement mechanisms can be seen as transformers of timed words.

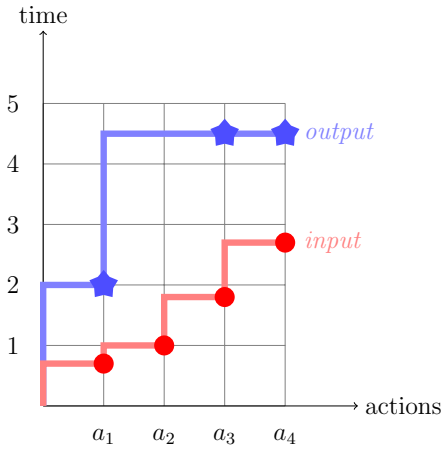


Figure 5.2: Behavior of enforcement monitors.

Figure 5.2 illustrates how an enforcement mechanism behaves, and what it can do to correct an input sequence. The red and blue curves respectively represent input and output sequences of events (actions in abscissa and occurrence dates in ordinate). In addition to the satisfaction of the property (not represented in the figure), the enforcement mechanism cannot change the order of actions, but can increase their dates or suppress some actions. For example, action a_2 is suppressed. Notice that the enforcement mechanism is allowed to reduce delays between events. For example, action a_4 occurs 0.9 time units after action a_3, but, both of these actions are released as output at the same time. Moreover, the actions should be released as output as soon as possible. Several application domains have requirements, where the required timing constraints can be satisfied by increasing dates of some actions [PFJM14a]. 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.

5.1.2 Some motivating examples

In this subsection, some intuition on the expected behavior of enforcement mechanisms is presented, considering again the requirements related to the usage of resources by some processes introduced in Section 4.1. Moreover, some issues arising in the timed context are also pointed out, and their relation to the expected constraints on enforcement mechanisms are discussed.

Let us consider the situation where two processes access to and operate on a common resource. Each process i (with $i \in \{1, 2\}$) has three interactions with the resource: acquisition (acq_i), release (rel_i), and a specific operation (op_i). Both processes can also execute a common action op. System initialization is denoted by action $init$. In the following, variable t keeps track of evolution of time. Figure 5.4 illustrates the behavior of enforcement mechanisms for some specifications on the behavior of the processes and for particular input sequences. [1]

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

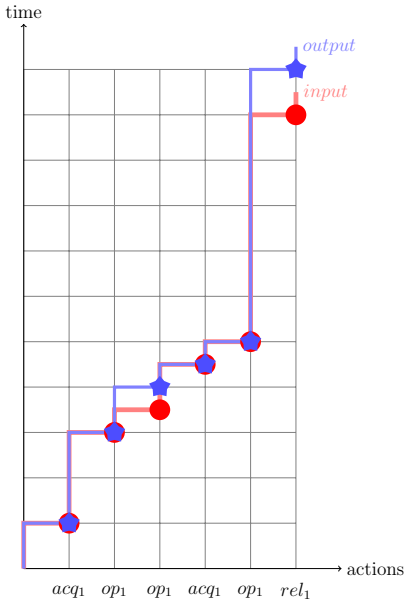
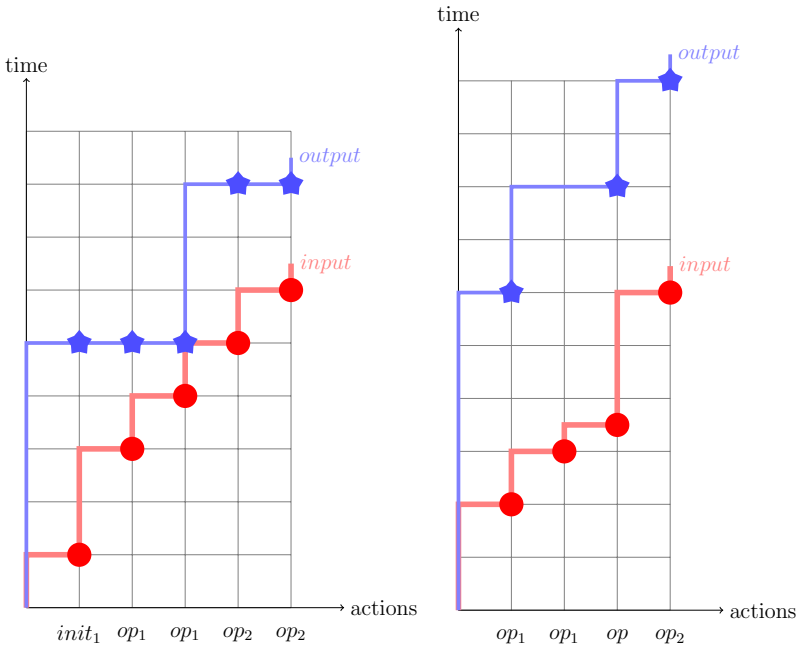


Figure 5.3: Specification $S1$, input σ_1.

Let us consider the input sequence $\sigma_1 = (1, acq_1) \cdot (3, op_1) \cdot (3.5, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (10, rel_1)$ (where each event is composed of an action associated with a date, indicating the absolute time at which the action is received as input). The monitor receives the first action acq_1 at $t = 1$, followed by op_1 at $t = 3$, etc. At $t = 1$ (resp.

1. In Section 4.2.2 we have seen how to formalize these specifications by timed automata.

$t = 3$), the monitor can output action acq_1 (resp. op_1) because this event (resp. the sequence $(1, acq_1) \cdot (3, op_1)$) satisfies specification S_1. At $t = 3.5$, when the second action op_1 is input, the enforcer determines that this action should be delayed by 0.5 t.u. to ensure the constraint that 1 t.u. should elapse between occurrences of op_1 actions. Hence, the second action op_1 is released at $t = 4$. At $t = 4.5$, when action acq_1 is received, the enforcer releases it immediately since this action is allowed by the specification with no time constraint. Similarly, at $t = 5$, an op_1 action is received and is released immediately because at least 1 t.u. elapsed since the previous op_1 action was released as output. At $t = 10$, when action rel_1 is received, it is delayed by 1 t.u. to ensure that the resource is kept for at least 10 t.u. (the first acq_1 action was released at $t = 1$). Henceforth, as shown in Figure 5.3, the output of the enforcement mechanism for σ_1 is $(1, acq_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (4.5, acq_1) \cdot (5, op_1) \cdot (11, rel_1)$.



(a) Specification $S2$, input σ_2. (b) Specification $S3$, input σ_3.

Figure 5.4: Some examples illustrating behavior of enforcement mechanism.

Specification S_2 The specification states that "*After system initialization, both processes should perform an operation (actions* op_i*) before 10 t.u. The operations of the different processes should be separated by 3 t.u.*"

Let us consider the input sequence $\sigma_2 = (1, init_1) \cdot (3, op_1) \cdot (4, op_1) \cdot (5, op_2) \cdot (6, op_2)$. At $t = 1, 3, 4$, when the enforcement mechanism receives the actions, it cannot release them as output but memorizes them since, upon each reception, the sequence of actions it received so far cannot be delayed so that a known continuation may satisfy specification S_2. At $t = 5$, upon the reception of action op_2, the sequence received so

far can be delayed to satisfy specification S_2. Action $init_1$ is released at $t = 5$ because it is the earliest possible date: a smaller date would be already elapsed. The two op_1 actions are also released at $t = 5$, because there are no timing constraints on them. The first op_2 action is released at $t = 8$ to ensure a delay of at least 3 t.u. with the first op_1 action. The second op_2 action is also released at $t = 8$, since it does not need to be delayed. Henceforth, as shown in Figure 5.4a, the output of the enforcement mechanism for σ_2 is $(5, init_1) \cdot (5, op_1) \cdot (5, op_1) \cdot (8, op_2) \cdot (8, op_2)$.

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

Let us consider the input sequence $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. At $t = 2$, the monitor can not output action op_1 because this action alone does not satisfy the specification (and the monitor does not yet know the next events i.e., actions and dates). If the next action was op_2, then, at the date of its reception, the monitor could output action op_1 followed by op_2, as it could choose dates for both actions in order to satisfy the timing constraints. At $t = 3$ the monitor receives a second op_1 action. Clearly, there is no possible date for these two op_1 actions to satisfy specification S_3, and no continuation could solve the situation. The monitor thus suppresses the second op_1 action, since this action is the one that prevents satisfiability in the future. At $t = 3.5$, when the monitor receives action op, the input sequence still does not satisfy the specification, but there exists an appropriate delaying of such action so that with future events, the specification can be satisfied. At $t = 6$, the monitor receives action op_2, it can decide that action op_1 followed by op and op_2 can be released as output with appropriate delaying. Thus, the date associated with the first op_1 action is set to 6 (the earliest possible date, since this decision is taken at $t = 6$), 8 for action op (since 2 is the minimal delay between those actions satisfying the timing constraint), and 10 for action op_2. Henceforth, as shown in Figure 5.4b, the output of the enforcer for σ_3 is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$.

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

Let us consider the input sequence $\sigma_4 = (1, acq_i) \cdot (2, op_i) \cdot (3, rel_i)$. At $t = 3$, when the monitor receives rel_i, it can decide that the three events acq_i, op_i, and rel_i can be released as output with appropriate delaying. Thus, the date associated with the first action acq_i is set to 3 . The output of the enforcer for σ_4 is $(3, acq_i) \cdot (3, op_i) \cdot (13, rel_i)$. To satisfy the timing constraint on release actions after acquisitions, the date associated to the last event rel_i is set to 13.

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

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

5.2 Preliminaries to Runtime Enforcement

In this section, some notations and partial orders on timed words which will be useful for enforcement are defined.

Definition 5.1 (Observation of a timed word at time t) *Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in \mathrm{tw}(\Sigma)$, the* observation of σ at time t *is the prefix of σ that can be observed at date t. It is defined as the maximal prefix of σ whose ending date is lower than t:*

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

For example, let $\sigma = (2, a) \cdot (3.6, b) \cdot (10, a)$. We get $\mathrm{obs}(\sigma, 5) = (2, a) \cdot (3.6, b)$.

Definition 5.2 (Delaying order $\succcurlyeq_d$) *For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, σ' delays σ (noted $\sigma' \succcurlyeq_d \sigma$) iff they have the same untimed projections, and the date of each event in σ' is greater than or equal to the date of the corresponding event in σ. Formally:*

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

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

For example, let $\sigma = (3,a)\cdot(5,b)\cdot(8,c)$ and $\sigma' = (3,a)\cdot(7,b)\cdot(9,c)$. $\sigma' \succcurlyeq_d \sigma$. Note that, absolute dates can be increased (but cannot be decreased), but duration between events may be decreased, e.g., the delay between b and c in σ is 3 time units and in σ' it is 2 time units).

Definition 5.3 (Delaying subsequence order $\lhd_d$) *For $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$, σ'* is a delayed subsequence of *σ (noted $\sigma' \lhd_d \sigma$) iff there exists a subsequence $\sigma"$ of σ such that σ' delays $\sigma"$. Formally:*

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

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

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

Definition 5.4 (Lexical order $\preceq_{\mathrm{lex}}$) *This order is useful to choose a unique timed word among some with same untimed projection. For two timed words σ, σ' with same untimed projection (i.e., $\Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$), the order $\preceq_{\mathrm{lex}}$ is defined inductively as follows: $\epsilon \preceq_{\mathrm{lex}} \epsilon$, and for two events with identical actions (t,a) and (t',a), $(t,a)\cdot\sigma \preceq_{\mathrm{lex}} (t',a)\cdot\sigma'$ if $t \leq t' \vee (t = t' \wedge \sigma \preceq_{\mathrm{lex}} \sigma')$.*

For example $(3,a)\cdot(5,b)\cdot(8,c)\cdot(11,d) \preceq_{\mathrm{lex}} (3,a)\cdot(5,b)\cdot(9,c)\cdot(10,d)$.

Definition 5.5 (Choosing a unique timed word with minimal duration) *Given a set of timed words with same untimed projection, $\min_{\preceq_{\mathrm{lex}},\mathrm{end}}$ selects the minimal timed word w.r.t the lexical order among timed words with minimal ending date: first the set of timed words with minimal ending date are considered, and then, from these timed words, the (unique) minimal one is selected w.r.t the lexical order. Formally, for a set $E \subseteq \mathrm{tw}(\Sigma)$ such that $\exists w \in \Sigma^*, \forall \sigma \in E : \Pi_\Sigma(\sigma) = w$, we have $\min_{\preceq_{\mathrm{lex}},\mathrm{end}}(E) = \min_{\preceq_{\mathrm{lex}}}(\min_{\preceq_{\mathrm{end}}}(E))$ where $\sigma \preceq_{\mathrm{end}} \sigma'$ if $\mathrm{end}(\sigma) \leq \mathrm{end}(\sigma')$, for $\sigma, \sigma' \in \mathrm{tw}(\Sigma)$.*

5.3 Enforcement Monitoring in a Timed Context

This section first introduces the enforcement monitoring framework (Section 5.3.1), and the constraints that should be satisfied by enforcement mechanisms (Section 5.3.2) are specified.

5.3.1 General principles

To ease the design and implementation of enforcement monitoring mechanisms in a timed context, enforcement mechanisms are described at three levels of abstraction: *enforcement functions*, *enforcement monitors*, and *enforcement algorithms*. An enforcement function describes the transformation of an input timed word into an output timed word. In this section the constraints required on enforcement functions are formalized. In Section 5.4 such enforcement functions are defined, and it is proved that the enforcement functions satisfy the constraints. An enforcement monitor is a more concrete view and defines the operational behavior of the enforcement mechanism over time. In Section 5.5 it is defined as an extended transition system and it is also proved that, for a given property φ, the associated enforcement monitor implements the corresponding enforcement function. In other words, an enforcement function serves as an abstract description (black-box view) of an enforcement monitor, and an enforcement monitor is the operational description of an enforcement function.

5.3.2 Constraints on an enforcement mechanism

At an abstract level, an enforcement mechanism for a given property φ can be seen as a function which takes as input a timed word and outputs a timed word. This is schematized in Figure 5.5 and defined in Definition 5.6.

Definition 5.6 (Enforcement function) *For a timed property φ, an enforcement mechanism behaves as a function, called* enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$.

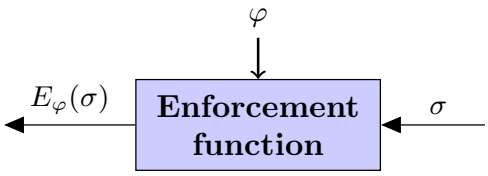


Figure 5.5: Enforcement function

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

Before providing the actual definition of enforcement function in Section 5.4, the constraints that should be satisfied by an enforcement mechanism dedicated to some property φ, are defined. The following constraints can serve as a specification of the expected behavior of enforcement mechanisms for timed properties, that can delay and suppress events.

An enforcement mechanism should first satisfy some *physical constraint* reflecting the streaming of events: the output stream can only be modified by appending new events to its tail. Second, it should be *sound*, which means that it should correct input words according to φ if possible, and otherwise produce an empty output. Third, it

should be *transparent*, which means that it is only allowed to shift events in time while keeping their order (such behavior is referred to as time retardants) and to suppress some events. These constraints are formalized in the following definition:

Definition 5.7 (Constraints on an enforcement mechanism) *Given a timed property φ, an enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, should satisfy the following constraints:*

- **Physical constraint:**

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

- **Soundness:**

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

- **Transparency:**

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

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

Remark 5.1 *Other constraints. Notice that for any input σ, releasing ϵ as output would satisfy the soundness, transparency and physical constraint. We want to suppress an event or to introduce additional delay only when necessary. In addition to the soundness, transparency and physical constraint, our enforcement mechanisms in a timed context also satisfies the following optimality constraints:*

Op1 ***Streaming behavior and deciding to output as soon as possible.*** *An enforcement function does not have the entire input sequence, and also the length of the input is unknown. For efficiency reasons, the output should be built incrementally in a streaming fashion. Enforcement mechanisms should take decision to release input events as soon as possible. Only when there is no possibility to correct the input, the enforcement mechanism should wait to receive more events. For example, in case if the property φ is a safety property[2], soon after an event is received as input, the date at which it should be released as output should be decided.*

2. Safety properties are prefix closed.

Op2 ***Optimal suppression.*** *The transparency constraint allows to suppress some events. Suppression should occur only when necessary, i.e., when, upon the reception of a new event, there is no possibility to satisfy the property, whatever is the continuation of the input.*

Op3 ***Optimal dates.*** *The transparency constraint also allows to increase the dates of events. The enforcement mechanism should also choose dates that are optimal with respect to the current situation, releasing events as output as soon as possible.*

It can be easily checked on the examples in Section 5.1 that the output sequences satisfy the constraints of enforcement mechanisms.

5.4 Enforcement Functions: Input/Output Description of Enforcement Mechanisms

In this section, an enforcement function dedicated to a desired property φ is defined. Its purpose is to define, at an abstract level, for any input word σ, the output word $E_\varphi(\sigma)$ expected from an enforcement mechanism.

Firstly, some preliminaries are discussed (Section 5.4.1). Then, the enforcement function itself is defined, and in Section 5.4.2 we prove that this functional definition satisfies the physical, soundness, and transparency constraints. Finally, in Section 5.4.3, we explain how the enforcement function behaves over time (how a given input sequence is consumed over time, and how the output is released in an incremental fashion).

5.4.1 Preliminaries to the definition of the enforcement function

An enforcement mechanism needs to memorize events since, for some properties (typically co-safety properties), upon the reception of some input timed word, the property might not be yet satisfiable by delaying, but a continuation of the input may allow satisfaction. For more general properties (which are neither safety nor co-safety properties), there may exist some prefix for which the property is satisfiable by delaying the input, thus dates can be chosen for these events. Enforcement mechanisms take decisions on dates as soon as possible, and the output is built in a fashion that is as incremental as possible. Suppression occurs only when necessary, i.e., when, upon the reception of a new event, there is no possibility to satisfy the property, whatever is the continuation of the input.

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

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

$\mathrm{CanD}(\sigma)$ is the set of timed words w that delay σ, and start at or after the ending date of σ (which is the date of the last event of σ). As we shall see, the first conjunct stems from the fact that we consider enforcement mechanisms as time retardants, while the second one means that the eligible timed words should not start before the date

end(σ) (i.e., the date of its last event), as illustrated informally with specification S_3 in Section 5.1 and further discussed in Section 5.4.3.

5.4.2 Definition of the enforcement function

The enforcement function E_φ for a property φ defines how an input stream σ is transformed. For a property φ, the enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ is defined as $E_\varphi(\sigma) = \Pi_1(\mathrm{store}(\sigma))$ where $\mathrm{store}(\sigma) = (\sigma_s, \sigma_c)$ describes how the input stream is transformed. Sequence σ_s is a delayed subsequence of the input that is to be released as output. σ_c is a suffix of the input stream for which the dates at which these events can be released as output cannot be computed. For example, in case if the property is a safety property, the date at which an event should be released as output can be computed immediately and thus σ_c will be always ϵ. For co-safety and other regular properties (since they are not prefix closed), for some events, decision to output them also depends on the subsequent input events. Let us consider again the examples in Section 5.1. Consider specification S_2, and the input sequence $\sigma_2 = (1, \mathit{init}_1) \cdot (3, \mathit{op}_1) \cdot (4, \mathit{op}_1) \cdot (5, \mathit{op}_2) \cdot (6, \mathit{op}_2)$. Only upon receiving op_2 at $t = 5$, the sequence received so far can be delayed to satisfy specification S_2. So, the enforcement function stores the first 3 events in σ_c until it processes op_2.

Let us now see the definition of the enforcement function E_φ in detail.

Definition 5.8 (Enforcement function) *The enforcement function for a property φ is the function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ defined as:*

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

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

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

where

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

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

- σ_s is a delayed subsequence of the input σ, in fact of its prefix of maximal length for which the absolute dates have been computed to satisfy property φ;

– σ_c is a subsequence of the remaining suffix of σ for which the releasing dates of events, still have to be computed. It is a subsequence (and not the complete suffix) since some events may have been suppressed when no delaying allowed to satisfy φ, whatever is the continuation of σ, if any.

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

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

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

- If $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$ (and thus $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma'_c) \neq \emptyset$), it is possible to choose appropriate dates for the timed word $\sigma'_c = \sigma_c \cdot (t, a)$ to satisfy φ. Adding any timed word belonging to κ_φ to σ_s soon after $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$ satisfies the optimality constraint related to deciding to output events as soon as possible (Op1). The minimal timed word in $\kappa_\varphi(\sigma_s, \sigma'_c)$ w.r.t the lexicographic order is chosen among those with minimal ending date, and appended to σ_s; the second element of the pair is set to ϵ since all events memorized in $\sigma_c \cdot (t, a)$ are corrected and appended to σ_s. Notice also that choosing a timed word from κ_φ with minimal ending date satisfies the optimality constraint related to outputting events as soon as possible (Op3).
- If $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma'_c) = \emptyset$ (and thus $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$), it means that, whatever is the continuation of the current input $\sigma \cdot (t, a)$, there is no chance to find a correct delaying for (t, a). Thus, event (t, a) should be suppressed, leaving σ_c and σ_s unmodified. Only in this case, when we know that there is no possibility to correct the input anymore, the last read event (t, a) is suppressed, satisfying the optimality constraint related to minimal suppression (Op2).
- Otherwise, i.e., when $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma'_c) \neq \emptyset$ but $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$, it means that it is not yet possible to choose appropriate dates for $\sigma'_c = \sigma_c \cdot (t, a)$ to satisfy φ, but there is still a chance to do it in the future, depending on the continuation of the input, if

any. Thus σ_c is modified into $\sigma'_c = \sigma_c \cdot (t, a)$ in memory, but σ_s is left unmodified.

Remark 5.2 (Alternative strategies to suppress events) *When there is no possibility to continue correcting the input sequence (i.e., $\kappa_{\text{pref}(\varphi)} = \emptyset$), we choose to erase only the last received event (t, a), since it is the last one that causes this impossibility. However, other policies to suppress events could be chosen. In fact, one could choose to suppress any events in $\sigma_c \cdot (t, a)$, since dates of these events have not yet been chosen. This would then require to choose among all subsequences of $\sigma_c \cdot (t, a)$, and thus to define an appropriate order on those subsequences, which may be rather complex to define, and, more importantly, computationally expensive.*

Proposition 5.1 *Given some property φ, its enforcement function E_φ as per Definition 5.8 satisfies the physical **(Phy)**, soundness* **(Snd)**, *and transparency* **(Tr)** *constraints as per Definition 5.6.*

Proof 5.1 (of Proposition 5.1 - sketch only) *The proof of the physical constraint is a direct consequence of the definition of* store. *The proofs of soundness and transparency follow the same pattern: they rely on an induction on the length of the input word σ. The induction steps use a case analysis, depending on whether the last input subsequence (i.e, the events in $\sigma_c \cdot (t, a)$) can be corrected or not. The two possible cases are:*

Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$ In this case, using the induction hypothesis, we can deduce that $E_\varphi(\sigma) = E_\varphi(\sigma \cdot (t, a))$, and conclude that $E_\varphi(\sigma \cdot (t, a))$ satisfies soundness and transparency.

Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) \neq \emptyset$ In this case, the argument is based on the induction hypothesis and how κ_φ is defined.

The complete proofs are given in Appendix A.1.

In addition to the physical, soundness, and transparency constraints, as we already saw, the functional definition also satisfies optimality constraints. The optimality constraint (Op3) that each subsequence is released as output as soon as possible, is expressed by the following proposition.

Proposition 5.2 (Optimality of the enforcement function) *Given some property φ, its enforcement function E_φ as per Definition 5.8 satisfies the following optimality constraint:*

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

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

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

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

The optimality constraint expresses that, among those sequences w' that could have been chosen (see below), w is the minimal one in terms of ending date, and lexical order (this second minimality ensures uniqueness). The "sequences that could have been chosen" are those such that $m \cdot w'$ satisfies the property, have the same events (thus can be produced by suppressing the same events), are delayed subsequences of the input σ, and have a starting date greater than or equal to $\text{time}(\sigma)$, since $\text{time}(\sigma)$ is the date at which w' is appended to the output, and thus a smaller date would be in the past of the output event.

Example 5.1 *Consider Specification S3 introduced in Section 5.1. Let the input sequence be* $\sigma = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (4, op_2)$. *This input sequence can be corrected only after the last event* $(4, op_2)$ *is observed. Some possible output sequences are:* $o_1 = (6, op_1) \cdot (8, op) \cdot (10, op_2)$, $o_2 = (4, op_1) \cdot (7, op) \cdot (9, op_2)$, *and* $o_3 = (4, op_1) \cdot (6, op) \cdot (8, op_2)$. *All these three sequences are valid outputs (they satisfy physical, soundness and transparency constraints), but only* o_3 *satisfies optimality.*

Proof 5.2 (of Proposition 5.2 - sketch only) *The proofs rely on an induction on the length of the input word* σ. *The induction step uses a case analysis, depending on whether the last input subsequence (i.e, the events in* $\sigma_c \cdot (t, a)$*) can be corrected or not. The two possible cases are:*

Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) = \emptyset$ *In this case, using the induction hypothesis, we can deduce that* $E_\varphi(\sigma) = E_\varphi(\sigma \cdot (t, a))$, *and conclude that* $E_\varphi(\sigma \cdot (t, a))$ *satisfies optimality.*

Case $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a)) \neq \emptyset$ *In this case, we show that what is computed for* w *in the definition of optimality is equal to the corrected subsequence appended to* σ_s *by the enforcement function.*

The complete proof is given in Appendix A.2 (p. 126).

Remark 5.3 (Fixing the events suppressed in the optimality condition) *Note that the condition* $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ *in Proposition 5.2 stems from the strategy chosen to suppress events (see Remark 5.2). If an enforcement function is defined, such that it is allowed to suppress any event in* $\sigma_c \cdot (t, a)$, *then the condition* $\Pi_\Sigma(w') = \Pi_\Sigma(m^{-1} \cdot E_\varphi(\sigma))$ *in optimality should be removed.*

Remark 5.4 (Simplified enforcement function for safety properties) *Because of the characteristics of safety properties, the enforcement function for such a property* φ *can be simplified. We denote it as* $\text{store}^{\text{sa}}_\varphi$. *The output of function* $\text{store}^{\text{sa}}_\varphi$ *is a timed word instead of a pair of timed words (*$\text{store}^{\text{sa}}_\varphi : \text{tw}(\Sigma) \to \text{tw}(\Sigma)$*). One can notice that, by a simple induction using the definition of function* store *in Section 5.4.2,*

the second component in the output (σ_c) remains always empty. Indeed, σ_c is initially empty. Now, suppose that $\text{store}(\sigma) = (\sigma_s, \epsilon)$ *for some* $\sigma, \sigma_s \in \text{tw}(\Sigma)$. *Since φ is a safety property,* $\text{pref}(\varphi) = \varphi$, *thus,* $\kappa_{\text{pref}(\varphi)} = \kappa_\varphi$, *and thus the third case in the definition of function* store *never happens. Moreover, in the two remaining cases, the second argument remains ϵ. Thus, the internal memory can be removed from the definition. Additionally, in the first case, the first argument of the output can be simplified as it is always called with the last read event (t, a) (see below).*

Function $\text{store}^{\text{sa}}_\varphi$ *can be defined as follows:*

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

where $K\left(\sigma,(t,a)\right) \stackrel{\text{def}}{=} \{t' \in \mathbb{R}_{\geq 0} \mid t' \geq t \wedge \text{store}^{\text{sa}}_\varphi(\sigma)\cdot(t',a) \lhd_d \sigma\cdot(t,a) \wedge \text{store}^{\text{sa}}_\varphi(\sigma)\cdot(t',a) \in \varphi\}$ *is the set of dates $t' \geq t$ that can be associated to action "a", such that the extension* $\text{store}^{\text{sa}}_\varphi(\sigma)\cdot(t',a)$ *of* $\text{store}^{\text{sa}}_\varphi(\sigma)$ *is a delayed subsequence of $\sigma\cdot(t,a)$ and still satisfies property φ. It can then be proved that, for safety properties,* store *can be replaced by* $\text{store}^{\text{sa}}_\varphi$, *resulting in the same enforcement function E_φ.*

5.4.3 Behavior of the enforcement function over time

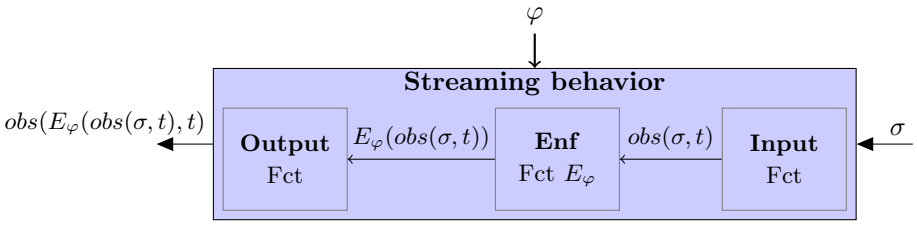


Figure 5.6: Behavior of the enforcement function over time.

The functional definition in Section 5.4.2, presents an abstract view, describing how to transform an input timed word σ according to a property φ. We abstract away from time, and show how a given complete input sequence σ is transformed. However, we want an enforcement mechanism to work in an online fashion, correcting partially known input, and releasing those events as output as soon as possible. So, now we present how the functional definition in Section 5.4.2 can compute output in an online fashion from partially observed input. Figure 5.6 provides an overview of how the enforcement function computes output incrementally over time using the input and output functions. The input and output functions as depicted in Figure 5.6 build the partial input and output of the enforcement function at any time instant. Both the input and output functions are realized using the obs function (see Definition 5.1).

At time t, the input and output functions compute the following:

– The input function computes what can be effectively observed from σ, which is $\text{obs}(\sigma, t)$ provided as input to the enforcement function;

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

Figure 5.7: Evolution of the enforcement function for property φ_3.

- The enforcement function computes the output corresponding to the input $\text{obs}(\sigma, t)$ which is $E_\varphi(\text{obs}(\sigma, t))$.
- Now, $E_\varphi(\text{obs}(\sigma, t))$ is a timed word where dates attached to events model the date when they should be released as output. Thus, only its prefix $\text{obs}(E_\varphi(\text{obs}(\sigma, t)), t)$ can be effectively released as output at date t, computed by the output function.

According to the physical constraint **(Phy)**, it is legal to build output incrementally since, $\text{obs}(\sigma, t)$ is the prefix of σ observed at time t, ($\text{obs}(\sigma, t) \preccurlyeq \sigma$), thus $E_\varphi(\text{obs}(\sigma, t))$ is a prefix of the complete output $E_\varphi(\sigma)$. Also, notice that, E_φ behaves as a time retardant (dates attached to output events exceed dates of corresponding input events). So, we also get $\text{obs}(E_\varphi(\text{obs}(\sigma, t)), t) = \text{obs}(E_\varphi(\sigma), t)$.

Thus, we can conclude that at date t, the released output is $\text{obs}(E_\varphi(\sigma), t)$, and what is ready to be released (but not released) is the residual of $E_\varphi(\text{obs}(\sigma, t))$ after releasing $\text{obs}(E_\varphi(\sigma), t)$ which is $\text{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\text{obs}(\sigma, t))$.

The enforcement monitor described in the next section, which concretizes the enforcement function, and will explicitly take care of this temporal behavior of our enforcement mechanism.

Example 5.2 (Enforcement function) *We illustrate how Definition 5.8 is applied to enforce specification S_3 (see Section 5.1), formalized by property φ_3, recognized by the automaton depicted in Figure 4.2c with Σ_3 ($= \{op_1, op_2, op\}$), and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Figure 5.7 shows the evolution of the observed input timed word $\text{obs}(\sigma_3, t)$, the output of the* store *function when the input timed word is $\text{obs}(\sigma_3, t)$, and E_{φ_3}. Variable t keeps track of physical time. When $t < 6$, the observed output is empty (since $E_{\varphi_3}(\text{obs}(\sigma_3, t)) = \epsilon$). When $t \geq 6$, the observed output, is $\text{obs}((6, op_1) \cdot (8, op) \cdot (10, op_2), t)$ (since $E_{\varphi_3}(\text{obs}(\sigma_3, t)) = (6, op_1) \cdot (8, op) \cdot (10, op_2)$).*

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

Figure 5.8: Evolution of the enforcement function for property φ_4 (a non-enforceable property).

Example 5.3 (Enforcement function: A non-enforceable property)
Consider specification S4 formalized by property φ_4, recognized by the automaton depicted in Figure 4.2d with $\Sigma_4^i \stackrel{\text{def}}{=} \{acq_i, op_i, rel_i\}$, and the input timed word $\sigma_4 = (3, acq_i) \cdot (7, op_i) \cdot (12, rel_i)$. Figure 5.8 shows the evolution of the observed input timed word $\text{obs}(\sigma_4, t)$, the output of the store *function when the input timed word is $\text{obs}(\sigma_4, t)$, and E_{φ_4}. The output of the enforcement function is ϵ at any time instant.*

5.5 Enforcement Monitors: Operational Description of Enforcement Mechanisms

The enforcement function definition in Section 5.4 is a functional view of our enforcement mechanism. The enforcement function is defined recursively which transforms an input stream of events according to property φ. The functional definition can be implemented using functional programming constructs such as recursion, and lazy evaluation.

However, a concern is that the computation of dates of new events are also dependent on the events that are already processed (and corrected). The functions κ_φ and $\kappa_{\text{pref}(\varphi)}$ also take σ_s as input. This requires to store all the corrected events in the memory of the enforcer. The memory σ_s grows over time and is never emptied.

An enforcement monitor is an alternative view which is based on the functional definition. It makes use of semantics of TA defining the property φ, is defined as a transition system $\mathcal{E}$ which provides a more operational view of the enforcement mechanism.

An enforcement function E_φ for a property φ specified by a TA $\mathcal{A}_\varphi$, is implemented by an enforcement monitor (EM), which provides a more operational view of the enforcement mechanism. An EM is defined as a transition system $\mathcal{E}$ and has explicit

state information. It keeps track of information such as time elapsed, and state of the underlying TA. Using the semantics of the TA defining the property φ, and keeping track of its current state (encoding the state reached upon σ_s), we no more require σ_s to compute the dates of the following input events in σ_c. The dates for the new events (that are in σ_c) can be computed by exploring all the paths from the current state of the underlying TA. An EM also releases all the corrected events over time (events in memory σ_s are removed and released as output at appropriate time instants).

5.5.1 Preliminaries to the definition of enforcement monitors

In contrast with an enforcement function which, at an abstract level, takes as input a timed word and produces a timed word as output, an enforcement monitor $\mathcal{E}$ also needs to take into account physical time (which is kept track of by clock t), the current observation $\text{obs}(\sigma, t)$ of the input stream σ at time t, the dumping of events to the environment which is characterized by $\text{obs}(E_\varphi(\sigma), t)$, and the residual of $E_\varphi(\text{obs}(\sigma, t))$ after releasing $\text{obs}(E_\varphi(\sigma), t)$.

An EM $\mathcal{E}$ is thus equipped with:

- a clock which tracks the current date t;
- it also uses two memories and a set of enforcement operations used to store and dump some timed events to and from the memories, respectively. The memories of an EM are basically queues, each of them containing a timed word:
  - σ_{ms} is the output queue which manages $E_\varphi(\sigma)$, but at time t, since only the prefix $\text{obs}(\sigma, t)$ has been observed, and $\text{obs}(E_\varphi(\sigma), t)$ has already been released, σ_{ms} contains the residual $\text{obs}(E_\varphi(\sigma), t)^{-1} \cdot E_\varphi(\text{obs}(\sigma, t))$, i.e., the timed word that is ready to be released but not yet released;
  - the other queue σ_{mc} manages the input, more precisely the subsequence of the input $\text{obs}(\sigma, t)$ composed of non-suppressed events for which dates could not yet be chosen to satisfy the property. This exactly corresponds to the timed word σ_c in the store function (see Definition 5.8).
- An EM also keeps track of the current state of the underlying LTS of the TA $\mathcal{A}_\varphi$ that encodes property φ. The current state is the one reached at time t after reading the timed word $E_\varphi(\text{obs}(\sigma, t))$ (that also corresponds to σ_s in the definition of E_φ) which is the output that can be computed from the current observation $\text{obs}(\sigma, t)$.

5.5.2 Update function

Before presenting the definition of enforcement monitor, we introduce function update that updates the current state in the underlying LTS of the TA, and mimics the computation of the sets CanD, κ_φ, and $\kappa_{\text{pref}(\varphi)}$ in function store. It also outputs a marker used by $\mathcal{E}$ to take decisions. Function update takes as input the current state ($q \in Q$) of $[\![\mathcal{A}_\varphi]\!]$ reached after reading sequence $E_\varphi(\text{obs}(\sigma, t))$ (it thus encodes the term $\sigma_s^{-1} \cdot \varphi$ in the definition of set $\kappa_\varphi(\sigma_s, \sigma'_c)$), sequence of events $\sigma_{\text{mc}} \in \text{tw}(\Sigma)$ (same as σ_c in the definition of store), and the last received event (t, a).

- If the new input event (t, a), appended to σ_{mc}, allows to satisfy the property by delaying, it outputs a state q' [3], a timed word w, and the marker `ok` where w is the sequence with minimal ending date delaying $\sigma_{\text{mc}} \cdot (t, a)$, and q' is the state reached from q by reading w.
- If for any delaying sequence, the state that is reached from q is in B (bad state), it returns the same state q, the sequence σ_{mc} (thus (t, a) is suppressed) and a marker `bad` (indicating that no further input event can allow to satisfy the property).
- Otherwise, if for some delaying sequence, some currently bad state B^C is reachable from q, and no delaying sequence from q leads to an accepting state Q_F, it does not modify q, and returns $\sigma_{\text{mc}} \cdot (t, a)$ and a marker `c_bad` (indicating that some further event from the input can allow to satisfy the property).

Definition 5.9 (Update function) update *is a function from* $Q \times \text{tw}(\Sigma) \times (\mathbb{R}_{\geq 0} \times \Sigma)$ *to* $Q \times \text{tw}(\Sigma) \times \{\texttt{ok}, \texttt{c\_bad}, \texttt{bad}\}$. *defined as follows:*

$$\text{update}(q, \sigma_{\text{mc}}, (t, a)) \stackrel{\text{def}}{=} \begin{cases} (q', w, \texttt{ok}) & \text{if } \Delta(q, \sigma_{\text{mc}} \cdot (t, a)) \neq \emptyset \wedge q \xrightarrow{w} q', \\ (q, \sigma_{\text{mc}}, \texttt{bad}) & \text{if } \forall w' \in \text{tw}(\Sigma) : w' \succcurlyeq_d \sigma_{\text{mc}} \cdot (t, a) \\ & \qquad \wedge \text{start}(w') \geq t \implies q \xrightarrow{w'} B, \\ (q, \sigma_{\text{mc}} \cdot (t, a), \texttt{c\_bad}) & \text{otherwise,} \end{cases}$$

where $w = \min_{\preceq_{\text{lex}}, \text{end}} \Delta(q, \sigma_{\text{mc}} \cdot (t, a))$ *with* $\Delta : Q \times \text{tw}(\Sigma) \to 2^{\text{tw}(\Sigma)}$ *defined as:*

$$\Delta(q, \sigma) = \left\{ w' \in \text{tw}(\Sigma) \mid w' \succcurlyeq_d \sigma \wedge \text{start}(w') \geq \text{end}(\sigma) \wedge q \xrightarrow{w'} Q_F \right\}.$$

$\Delta(q, \sigma_{\text{mc}} \cdot (t, a))$ is the set of timed words w' delaying $\sigma_{\text{mc}} \cdot (t, a)$, starting at a date greater than or equal to the (current) date t, and reaching an accepting state $q' \in Q_F$. Since q encodes $\sigma_s^{-1} \cdot \varphi$, $\Delta(q, \sigma_{\text{mc}} \cdot (t, a))$ exactly encodes set $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t, a))$.

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

- In the first case, $\Delta(q, \sigma_{\text{mc}} \cdot (t, a))$ is not empty, i.e., appropriate delaying dates can be chosen for the events in $\sigma_{\text{mc}} \cdot (t, a)$ such that an accepting state $q' \in Q_F$ is reached from q. [4] In this case, function update returns the minimal word w w.r.t the lexical order among those timed words of minimal ending date in $\Delta(q, \sigma_{\text{mc}} \cdot (t, a))$, the state $q' \in Q_F$ reached from q by w, and marker `ok` indicating that Q_F is reached.
- In the second case, it is impossible to correct $\sigma_{\text{mc}} \cdot (t, a)$ in the future since all the candidate sequences delaying $\sigma_{\text{mc}} \cdot (t, a)$ lead to states in B, i.e., non-accepting states from which no path leads to an accepting state. This reflects the fact that $\kappa_{\text{pref}(\varphi)}(\sigma_s, \sigma_c \cdot (t, a))$ is empty. In this case, function update lets state q and timed word σ_{mc} unmodified, indicating that event (t, a) is suppressed, and marker `bad`

3. Note that q' is necessarily an accepting state ($q' \in Q_F$).

4. This case could be further split in two cases depending on whether $q' \in G$ or $q' \in G^c$. In the first case, one could then simplify enforcement since now all reachable states from q' are in G, thus only the condition $\text{start}(w') \geq \text{end}(\sigma)$ of Δ has to be checked in subsequent calls to update.

indicates that no accepting state could be reached in the future if (t, a) was retained in memory.
- In the third case, function update lets state q unmodified, but returns the timed word $\sigma_{\text{mc}} \cdot (t, a)$, and a marker `c_bad`. This indicates that $\sigma_{\text{mc}} \cdot (t, a)$ can not be delayed to reach an accepting state, but there is still a chance to reach a new accepting state after observing more events in the future.

5.5.3 Definition of enforcement monitors

We define the notion of enforcement monitor.

Definition 5.10 (Enforcement Monitor) *Let us consider a TA $\mathcal{A}_\varphi$, with semantics $(Q, q_0, \Gamma, \rightarrow, Q_F)$, recognizing some property φ. The enforcement monitor $\mathcal{E}$ for φ is the transition system $(C^{\mathcal{E}}, c_0^{\mathcal{E}}, \Gamma^{\mathcal{E}}, \hookrightarrow_{\mathcal{E}})$ s.t.:*
- *$C^{\mathcal{E}} = \text{tw}(\Sigma) \times \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \times Q$ is the set of configurations of the form $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q)$, where σ_{ms}, σ_{mc} are timed words to memorize events, t is a positive real number to keep track of time, and q is a state in the semantics of the TA,*
- *$c_0^{\mathcal{E}} = (\varepsilon, \varepsilon, 0, q_0) \in C^{\mathcal{E}}$ is the initial configuration,*
- *$\Gamma^{\mathcal{E}} = ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}) \times Op \times ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})$ is the alphabet, i.e., the set of triples comprised of an optional input event, an operation, and an optional output event, where the set of possible operations is $Op = \{\text{store-}\overline{\varphi}(\cdot), \text{store}_{\text{sup}}\text{-}\overline{\varphi}(\cdot), \text{store-}\varphi(\cdot), \text{dump}(\cdot), \text{idle}(\cdot)\}$;*
- *$\hookrightarrow_{\mathcal{E}} \subseteq C \times \Gamma_{\mathcal{E}} \times C$ is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:*
  - **1.** ***store-φ:***
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q) \overset{(t,a)/\text{store}-\varphi(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}}} (\sigma_{\text{ms}} \cdot w, \epsilon, t, q')$
    if $\text{update}(q, \sigma_{\text{mc}}, (t, a))) = (q', w, \texttt{ok})$
  - **2.** ***store$_{\text{sup}}$-$\overline{\varphi}$:***
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q) \overset{(t,a)/\text{store}_{\text{sup}}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}}} (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q)$
    if $\text{update}(q, \sigma_{\text{mc}}, (t, a)) = (q, \sigma_{\text{mc}}, \texttt{bad})$
  - **3.** ***store-$\overline{\varphi}$:***
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q) \overset{(t,a)/\text{store}-\overline{\varphi}(t,a)/\epsilon}{\hookrightarrow_{\mathcal{E}}} (\sigma_{\text{ms}}, \sigma_{\text{mc}} \cdot (t, a), t, q)$
    if $\text{update}(q, \sigma_{\text{mc}}, (t, a)) = (q, \sigma_{\text{mc}} \cdot (t, a), \texttt{c\_bad})$
  - **4.** ***dump:***
    $((t, a) \cdot \sigma'_{\text{ms}}, \sigma_{\text{mc}}, t, q) \overset{\epsilon/\,\text{dump}(t,a)/(t,a)}{\hookrightarrow_{\mathcal{E}}} (\sigma'_{\text{ms}}, \sigma_{\text{mc}}, t, q)$
  - **5.** ***idle:***
    $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q) \overset{\epsilon/\,\text{idle}(\delta)/\epsilon}{\hookrightarrow_{\mathcal{E}}} (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t + \delta, q)$.

A configuration $(\sigma_{\text{ms}}, \sigma_{\text{mc}}, t, q)$ of the EM consists of the current stored sequences (i.e., the content of the two memories) σ_{ms} and σ_{mc}. The sequence which is corrected and can be released as output is denoted by σ_{ms}. Sequence σ_{mc} is a sort of internal memory: this is the input sequence read by the EM, but yet to be corrected, except for events

that are suppressed. The configuration also contains a clock t that indicates the current time instant. The last element q is the current state of $[\![\mathcal{A}_\varphi]\!]$ reached after processing the sequence already released followed by the timed word in memory σ_{ms}, i.e., $E_\varphi(\text{obs}(\sigma, t))$.

Semantic rules can be understood as follows:

- Upon the reception of an event (t, a) (i.e., when t is the date in the configuration and (t, a) is read), one of the following store rules is executed. Notice that their conditions are exclusive of each others.
  - The ***store-***φ rule is executed if function update returns marker `ok`, indicating that φ can be satisfied by the sequence already released as output, followed by σ_{ms}, and followed by w which minimally delays $\sigma_{\text{mc}} \cdot (t, a)$. When executing the rule, sequence w is appended to the content of output memory σ_s.
  - The ***store***$_{\text{sup}}$***-***$\overline{\varphi}$ rule is executed if the update function returns marker `bad` (indicating that $\sigma_{\text{mc}} \cdot (t, a)$ followed by any sequence cannot be corrected). Event (t, a) is then suppressed, and the configuration remains unchanged.
  - The ***store-***$\overline{\varphi}$ rule is executed if the update function returns marker `c_bad` (indicating that $\sigma_{\text{mc}} \cdot (t, a)$ cannot be corrected yet). The event (t, a) is then appended to the internal memory σ_{mc}.
- The ***dump*** rule is executed if the current time t is equal to the date corresponding to the first event of the timed word $\sigma_{\text{ms}} = (t, a) \cdot \sigma'_{\text{ms}}$ in the memory. The event is released as output and removed from σ_{ms} in the resulting configuration.
- The ***idle*** rule adds the time elapsed δ (where $\delta \in \mathbb{R}_{>0}$) to the current value of t when no store nor dump operation is possible.

Note, all rules except the ***idle*** rule execute in zero time [5].

Example 5.4 (Execution of an enforcement monitor) *We illustrate how the rules of Definition 5.10 are applied to enforce property φ_3 (see Section 5.1), recognized by the automaton depicted in Figure 4.2c with $\Sigma_3 = \{op_1, op_2, op\}$, and the input timed word $\sigma_3 = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. Figure 5.9 shows how semantic rules are applied according to the current date t, and the evolution of the configurations of the enforcement monitor, together with input and output. More precisely, each line follows the syntax $O/c/I$, where O is the sequence of released events, c is a configuration, and I is the residual of the input σ after its observation at time t.*

*The resulting (final) output is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$, which satisfies property φ_3. After $t = 10$, only the **idle** rule can be applied.*

Remark 5.5 (Simplified EM definition for safety properties)
Because of the characteristics of safety properties, following the simplifications in the functional definition (see Remark 5.4), enforcement monitor definition can be also simplified. Only one timed word is needed in the configuration, and the internal memory

5. 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 the other operations, and the semantic rules of these operations can be modified accordingly (incrementing the value of clock "t" with some delay in the resulting configuration).

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

Figure 5.9: Execution of an enforcement monitor for φ_3.

σ_{mc} *is not necessary. Also, one of the store rules store-*$\overline{\varphi}$ *will never be applicable since the set of currently bad states* B^c *for a safety TA will be always empty, and the third case of the* update *function (Definition 5.9) can never occur.*

The simplified update *function for safety properties* update_s *is a function from* $Q \times (\mathbb{R}_{\geq 0} \times \Sigma)$ *to* $Q \times (\mathbb{R}_{\geq 0} \times \Sigma) \times \{\texttt{ok}, \texttt{bad}\}$. *defined as follows:*

$$
\text{update}_s(q,(t,a)) \stackrel{\text{def}}{=} \begin{cases} (q',(t',a),\texttt{ok}) & \text{if } \Delta(q,(t,a)) \neq \emptyset \wedge q \overset{(t',a)}{\rightarrow} q', \\ (q,(t,a),\texttt{bad}) & \text{if } \Delta(q,(t,a)) = \emptyset, \end{cases}
$$

where $w = min\Delta(q,(t,a))$ *with* $\Delta : Q \times (\mathbb{R}_{\geq 0} \times \Sigma) \to 2^{\mathbb{R}_{\geq 0}}$ *defined as:*

$$
\Delta(q,(t,a)) = \left\{ t' \in \mathbb{R}_{\geq 0} \mid t' \geq t \wedge q \overset{(t',a)}{\rightarrow} Q_F \right\}.
$$

$\Delta(q,(t,a))$ *is the set of dates* t' *greater than or equal to* t*, and an accepting state* $q' \in Q_F$ *is reachable from* q *upon* (t',a)*.*

In the first case, $\Delta(q,(t,a))$ *is not empty, appropriate date* t' *can be chosen for the event* (t,a)*. The* update_s *function returns a state* q'*,* (t',a) *and a marker* `ok` *indicating that the event can be released as output at time* t'*. In the second case,* $\Delta(q,(t,a))$ *is empty, meaning that there is no appropriate date, and the event* (t,a) *cannot be released as output. In this case, the* update_s *function returns back the same state* q*, same event* (t,a) *and a marker* `bad` *indicating that the event* (t,a) *should be suppressed.*

5.5.4 Relating enforcement functions and enforcement monitors

An enforcement function (see Section 5.4) describes at an abstract level, for any input word σ, the output word $E_\varphi(\sigma)$ expected from an enforcement mechanism. An enforcement monitor $\mathcal{E}$ (see Section 5.5) is defined as a transition system, providing a more concrete view and describes the operational behavior of the enforcement mechanism over time. We present how the definitions of enforcement function and enforcement monitor can be related: given a property φ, any input sequence σ, at any time instant t, the output of the associated enforcement function and the output-behavior of the associated enforcement monitor are equal. In Section 5.4, we already proved that the enforcement functions satisfy the constraints.

By relating enforcement functions and enforcement monitors, and showing that for any property and for any input, they compute the same output, we can also conclude that the enforcement monitor also satisfies the constraints.

Preliminaries We first describe how an enforcement monitor reacts to an input sequence. In the remainder of this section, we consider an enforcement monitor $\mathcal{E} = (C^{\mathcal{E}}, c_0^{\mathcal{E}}, \Gamma^{\mathcal{E}}, \hookrightarrow_{\mathcal{E}})$. According to Definition 5.10, the ***store*** rules which read an input event have priority over the ***dump*** rule which reads ϵ as input but produces an event as output. The ***idle*** operation neither reads an input event, nor produces an output event, and will always be the transition with least priority.

Enforcement monitors, described in Section 5.5, are deterministic. By determinism, we mean that, given an input sequence, the observable output sequence is unique. Moreover, given $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, how an enforcement monitor reads σ until time t is unique if consecutive ***idle*** operations are merged: it goes through a unique sequence of configurations However, given an input sequence σ and a time instant t, because of the ***idle*** rule which does not read any input event nor produce any output event there is possibly an infinite set of corresponding sequences over the *input-operation-output* alphabet (as in Definition 5.10). All these sequences are equivalent: they involve the same configurations if consecutive ***idle*** operations are merged into a single ***idle*** operation for the enforcement monitor and produce the same output sequence.

More formally, let us define $\mathcal{E}_{\text{ioo}}(\sigma,t) \in (\Gamma^{\mathcal{E}})^*$ to be the unique sequence of transitions (triples comprised of an optional input event, an operation, and an optional

output event) that is "triggered" from the initial configuration, when the enforcement monitor reads σ until time t:

Definition 5.11 (Input-Operation-Output sequence) *Given an input sequence $\sigma \in \mathrm{tw}(\Sigma)$ and some time instant $t \in \mathbb{R}_{\geq 0}$, we define the input-operation-output sequence, denoted as $\mathcal{E}_{\mathrm{ioo}}(\sigma, t)$, as the unique sequence of $(\Gamma^{\mathcal{E}})^*$ such that:*

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

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

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

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

Relating enforcement functions and enforcement monitors We now relate the enforcement function and the enforcement monitor, for a property φ. Seen from the outside, an enforcement monitor $\mathcal{E}$ behaves as a device reading and producing timed words. Overloading notations, this input/output behavior can be characterized as a function $\mathcal{E} : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \mathrm{tw}(\Sigma)$ defined as:

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

The corresponding output timed word $\mathcal{E}(\sigma, t)$, at any time t, is the concatenation of all the output events produced by the enforcement monitor over various steps of the enforcement monitor (erasing ϵ's). In the following, we do not make the distinction between an enforcement monitor and the function that characterizes its behavior.

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

Definition 5.12 (Implementation relation) *Given an enforcement function E_φ (as per Definition 5.8) and an enforcement monitor (as per Definition 5.10) whose behavior is characterized by a function $\mathcal{E}$, we say that $\mathcal{E}$* implements *E_φ iff:*

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

Proposition 5.3 (Relation between E_φ and $\mathcal{E}$) *Given a property φ, its enforcement function E_φ (as per Definition 5.8), and its enforcement monitor $\mathcal{E}$ (as per Definition 5.10), $\mathcal{E}$ implements E_φ in the sense of Definition 5.12.*

Proof 5.3 (of Proposition 5.3 - sketch only) *A complete proof of this proposition is given in Appendix A.3.4, p. 133. The proof relies on an induction on the length of the input word σ. The induction step uses a case analysis, depending on whether the input is completely observed or not at time t, whether the input can be delayed into a correct output or not, and whether the memory content (σ_{ms}) is completely dumped or not at time t. The proof also uses several intermediate lemmas that characterize some special configurations (e.g., value of the clock variable, content of the memory σ_{ms}) of an enforcement monitor.*

5.6 Summary

This chapter presented a general enforcement monitoring framework for systems with timing requirements. We showed how to synthesize enforcement mechanisms for any regular timed property (modeled by a timed automaton). The proposed enforcement mechanisms are more powerful than the ones in our previous research endeavors [PFJ+12, PFJM14b]. In particular, proposed enforcement mechanisms allow to delay the events of the observed input (while being allowed to shorten the delay between some events), and also to suppress events. An event is suppressed if it is not possible to satisfy the property by delaying, whatever are the future continuations of the input sequence (i.e., the underlying TA can only reach non-accepting states from which no accepting state can be reached). Enforcement mechanisms are described at different levels of abstraction (enforcement function, and monitor), thus facilitating the design and implementation of such mechanisms.

An enforcement monitor is still an abstract view of a real enforcement mechanism and needs to be further concretized. In Chapter 6, algorithms are described showing how enforcement monitors can be implemented. Furthermore, a prototype is implemented based on the proposed algorithms. In Chapter 6, the prototype implementation, experimental framework, and the experiments demonstrating the feasibility of enforcement monitoring for timed properties are described in detail.

In some application domains, we also encounter requirements with constraints both on time and data. Also, in the context of a client-server scenario, we may have to treat messages sent to the server from different clients independently. In Chapter 7, we introduce a model called as Parametrized Timed Automata with Variables (PTAV) which is an extension of timed automata, allowing to express much richer requirements. We also present how the enforcement mechanism presented in this chapter can be extended to enforce properties expressed as PTAVs.

Chapter 6

Implementation and Evaluation

In this chapter, we show how the abstract view of enforcement functions and enforcement monitors described in chapter 5 can be further concretized into an implementation. Firstly, we present the enforcement algorithms, describing how to realize enforcement monitors. Enforcement algorithms are also implemented in Python. We describe the experimental framework developed to evaluate the performance of enforcement monitors, and finally discuss the evaluation results.

6.1 Enforcement Algorithms

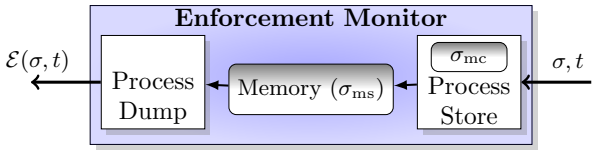


Figure 6.1: Realizing an EM.

The implementation of an enforcement monitor consists of two processes running concurrently (Store and Dump), and a shared memory, as shown in Figure 6.1. Process Store implements the store rules of the enforcement monitor. The memory contains the timed word σ_{ms}: the corrected sequence that can be released as output. The memory σ_{ms} is realized as a queue, shared by the Store and Dump processes, where the Store process adds events, that are processed and corrected, to this queue. Process Dump reads the events stored in the memory σ_{ms} and releases the action corresponding to each event as output, when time reaches the date associated to the event. Process Store also makes use of another internal buffer σ_{mc} (not shared with any other process), to store the events which are read, but can not be corrected yet, to satisfy the property. In the algorithms, primitive `await` is used to wait for a trigger event from another process or to wait until some condition becomes true. Primitive `wait` is used by a process to wait for some amount of time, which is determined by the process itself.

Algorithm 1 StoreProcess

$(t, q) \leftarrow (0, q_0)$
$\sigma_{\text{ms}} \leftarrow \epsilon$
$\sigma_{\text{mc}} \leftarrow \epsilon$
while `tt` **do**
  $(t, a) \leftarrow$ await $(event)$
  $(q', \sigma'_{\text{mc}}, isPath) \leftarrow \text{update}(q, \sigma_{\text{mc}}, (t, a))$
  if $isPath =$ `ok` **then**
    $\sigma_{\text{ms}} \leftarrow \sigma_{\text{ms}} \cdot \sigma'_{\text{mc}}$
    $\sigma_{\text{mc}} \leftarrow \epsilon$
    $q \leftarrow q'$
  else
    $\sigma_{\text{mc}} \leftarrow \sigma'_{\text{mc}}$
  end if
end while

Algorithm StoreProcess Algorithm StoreProcess (see Algorithm 3) is an infinite loop that scrutinizes the system for input events. In the algorithm, q memorizes the state of the property automaton, initialized to q_0. Function update implements the function of Definition 5.9. It takes the current state, the events stored in the internal memory σ_{mc} of the process Store, and the new event (t, a), and returns a new state q', a timed word σ'_{mc}, and a marker $isPath$ from the set $\{\texttt{ok}, \texttt{c\_bad}, \texttt{bad}\}$, indicating whether $\sigma_{\text{mc}} \cdot (t, a)$ can be delayed to satisfy φ.

The algorithm proceeds as follows. Process Store initially sets its clock t to 0, initializes q to q_0 and the two memories σ_{ms} and σ_{mc} to ϵ. It then enters an infinite loop where it waits for an input event. When receiving an action a at date t, it stores the event (t, a). It then invokes function update with the current state q, the events stored in σ_{mc} and the new event (t, a). The function returns a new state q', a timed word σ'_{mc} and the marker $isPath$. If the marker indicates $isPath =$ `ok`, it means that $\sigma_{\text{mc}} \cdot (t, a)$ can be corrected into the timed word σ'_{mc} computed by update and this word leads from state q to state q' in the underlying semantics of the timed automaton. Then, the timed word σ'_{mc} is appended to shared memory σ_{ms} (since it is now correct with respect to the property, and can be released as output), the internal memory σ_{mc} is cleared, and state q is updated to q'. In all other cases, σ_{mc} is set to σ'_{mc}, the result of update, which is either σ_{mc} if $isPath =$ `bad` (it is impossible to correct the input sequence σ_{mc} whatever are the future events) or $\sigma_{\text{mc}} \cdot (t, a)$ if $isPath =$ `c_bad`. In both cases, state q and σ_{ms} are unmodified.

Algorithm DumpProcess Algorithm DumpProcess (see Algorithm 2) is an infinite loop that scrutinizes memory σ_{ms} and proceeds as follows: Initially, clock d, which keeps track of the time elapsed is set to 0. Process Dump waits until the memory is not empty ($\sigma_{\text{ms}} \neq \epsilon$). Using operation dequeue, the first element stored in the memory is removed, and is stored as (t, a). Since d time units elapsed, process DumpProcess

Algorithm 2 DumpProcess

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

waits for $(t - d)$ time units before performing operation dump(a), releasing action a as output at date t (which amounts to appending (t, a) to the output of the enforcement monitor).

Remark 6.1 (Using non-deterministic TAs to define properties) *In this thesis, the presentation considers only deterministic TAs. The same mechanisms and algorithms remain valid to also enforce properties defined with non-deterministic TAs. The* update *function first computes all accepting paths from the current state, for the given input subsequence. And from this set of all accepting paths, a unique solution with minimal duration is chosen using the lexical order. Even if the underlying TA is non-deterministic, the policy used to choose a unique solution to define the* update *function (which computes optimal dates) will work. Only in situations when there is more than one accepting path, where the dates computed in each of these paths are exactly the same, but they end in different locations, some additional policy (such as picking one solution randomly) is necessary to choose a unique path among them. Note that the dates computed from all such paths are equal.*

Remark 6.2 (Simplified algorithms) *For safety properties, following the simplified functional and enforcement monitor definitions (see Remarks 5.4 and 5.5), the algorithm for the* StoreProcess *can also be simplified. In particular, in the algorithm for safety properties, the content of the memory can be maintained by a single sequence of events. Note that σ_{mc} will always be ϵ, and thus it is not necessary. Also, in case of safety properties, the* update *function is always invoked with a single event as input, instead of a sequence of events. The* StoreProcess *algorithm for safety properties can be simplified as follows:*

Note that this simplified algorithm uses the update_s *function (see Remark 5.5) to compute optimal date for each event. For each event (t, a), if the* update_s *function returns* `ok`*, then the event with optimal date returned by the* update_s *function is appended to the memory σ_{ms}, and the state information is updated to q'. Otherwise, we proceed with the next event, meaning that the event is suppressed (this corresponds to the case when* update_s *returns* `bad`*).*

Remark 6.3 (Enforcing several properties) *So far, we described how any single regular properties can be enforced. When a boolean combination of properties has to be enforced on a system, we can combine the properties into a single one and synthesize one*

Algorithm 3 StoreProcess$_{saf}$

$(t, q) \leftarrow (0, q_0)$
$\sigma_{\text{ms}} \leftarrow \epsilon$
while `tt` **do**
  $(t, a) \leftarrow$ await $(event)$
  $(q', (t', a), isPath) \leftarrow \text{update}_s(q, (t, a))$
  if $isPath =$ `ok` **then**
    $\sigma_{\text{ms}} \leftarrow \sigma_{\text{ms}} \cdot (t', a)$
    $q \leftarrow q'$
  end if
end while

enforcement monitor for the resulting property. Definition 4.5 in Section 4.3.5 describes how two properties defined by complete and deterministic TAs can be combined using Boolean operations (e.g., union, intersection and negation).

6.2 Overview and Architecture of TIPEX

In this section, we present TIPEX, a tool chain for Timed Property Enforcement during eXecution. Some of the modules of the TIPEX also have interest irrespective of enforcement.

The TIPEX tool consists of two modules (see Figure 6.2). The Timed Automata Generator (GTA) module consists of features that are not specific to enforcement monitors, and provides functionalities such as generating timed automata, and combining timed automata. The Enforcement Monitor Evaluation (EME) module consists of functionalities to synthesize enforcement monitors from a TA (implementation of the enforcement algorithms presented earlier in this chapter), and other functionalities such as a trace generator required to evaluate the performance of enforcement monitors. The EME module uses the GTA module to get an input timed automaton (defining the requirement), and other information such as the class to which the timed automaton belongs. Let us now see the features of both these modules in detail.

6.2.1 GTA module

There are a lot of ongoing research efforts related to the verification of timed systems by means of e.g., model-checking, testing, and runtime verification of timed systems. Central to these verification techniques is the use of timed automata (TA) [AD94] as a formalism to model requirements or systems. UPPAAL [BY03, LPY97] is a successful tool for the verification of realtime systems. UPPAAL is based on the theory of TA and has defined a somewhat standard syntax format for the definition of TA. Enforcement monitoring algorithms (EME module) are also implemented using some UPPAAL libraries and the input TA (defining the property) is a UPPAAL model.

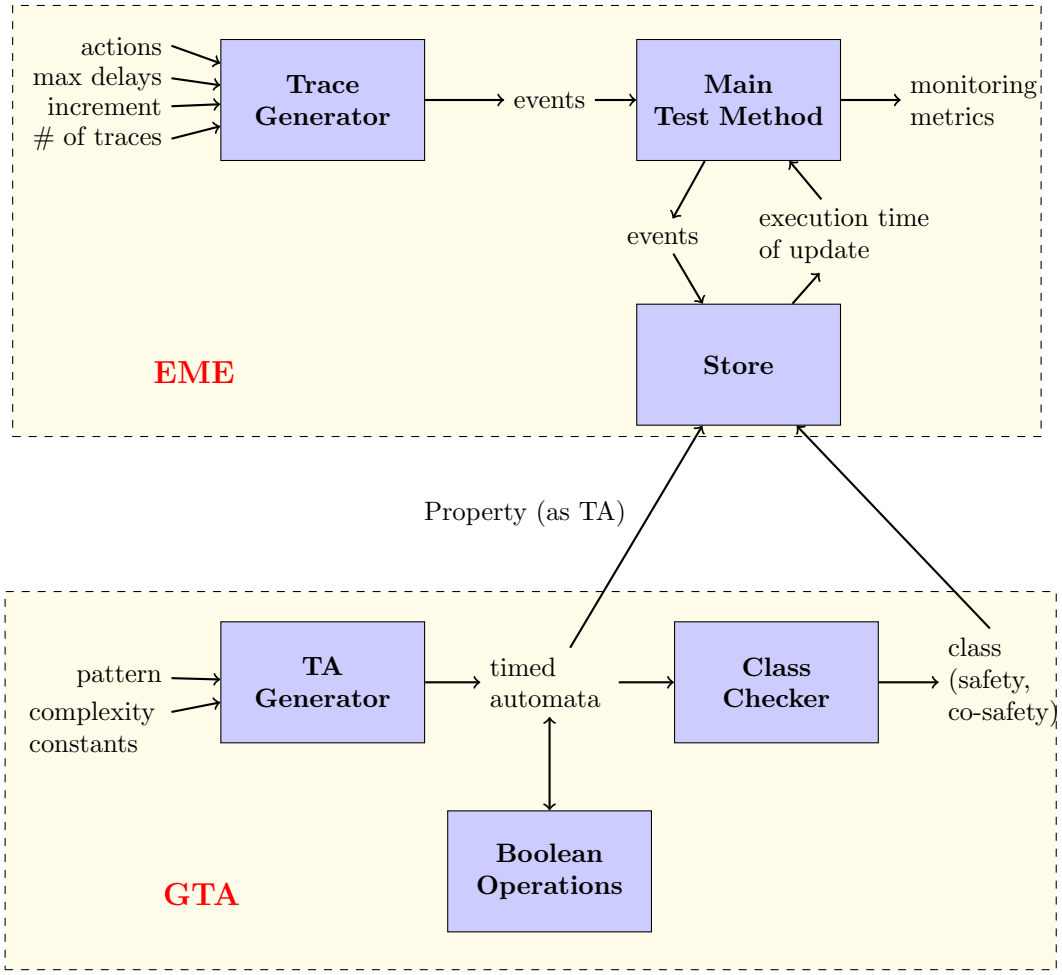


Figure 6.2: Overview of TIPEX tool.

To write formal specifications, specification patterns [DAC99] are particularly useful to guide non-experts as they provide a coverage of the various types of specifications that one may want to specify on a system [CMP93]. In [GL06], a set of specification patterns, that can be used to specify realtime requirements, is proposed.

GTA allows to:

- generate timed automata based on some user-inputs such as a pattern,
- to combine TA using Boolean operations, and
- to check the class to which a given TA belongs.

GTA generates timed automata in UPPAAL syntax, stored as an `.xml` file.

Motivations The main motivation behind GTA is to allow non-experts (such as engineers and developers who may not have the required experience and knowledge with formal specifications) to translate informal requirements into formal models eas-

ily. Moreover, when considering some complex requirements, automating the process and generating the model may be less erroneous compared to modeling such requirements manually (either in a graphical or a textual editor). Algorithms proposed in research papers need to be tested rigourously, considering several (and large) input test models. Using GTA, one can generate several (meaningful) timed automata, with varying complexity and belonging to different patterns, which can be used as input models for testing the proposed algorithms.

Let us consider the requirement *"There cannot be more than 100 requests in every 10 seconds"*. One can imagine that the timed automaton defining this requirement will have more than 100 locations and several transitions, and manually modeling it (for example using a graphical editor) is tedious and time consuming. Using GTA, the timed automaton defining this requirement can be obtained almost instantly, just by providing some required input data such as pattern, time constraint, and actions.

Definition 4.3.5 in Chapter 4 describes how two timed automata can be combined using Boolean operations. GTA also implements this, allowing to combine simple properties and to obtain one TA for which an enforcement monitor can be synthesized.

Figure 6.2 presents the architecture of GTA, with its modules, and their inputs and outputs. Note that the modules inside GTA are loosely coupled: each module can be used independently, and can be easily extended. GTA consists of the following sub-modules:

- **TA Generator:** This module constructs TAs by taking data such as a pattern, a time constraint constant, and complexity constant as input from the user.
- **Boolean Operations:** This module combines two TAs using Boolean operations such as Union, Intersection, and Negation.
- **Class Checker:** This module checks whether a TA belongs to a subclass (safety or co-safety) of regular properties.

GTA is implemented in Python. Further details about these modules, their features, required inputs, and their outputs are described in the following subsections.

6.2.1.1 Generating basic timed automata

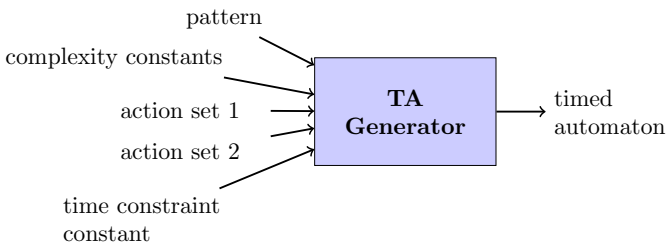


Figure 6.3: TA generator.

The *TA Generator* module generates timed automata. Generation of TA is based on some input data, shown in Figure 6.3.

The input parameter *pattern* indicates the desired pattern of the generated time

automaton. The parameter *complexity constant* is a natural number. The number of transitions and locations in the generated automaton will vary based on the value of this parameter. Two sets of input actions (*action set 1* and *action set 2*) should be provided as input: these are the actions used in the generated automaton. The parameter *time constraint constant* is a natural number that will be used in some guards in the generated TA to impose the required time constraints.

The tool generates a timed automaton as output in UPPAAL modeling language syntax stored as an `.xml` file. The generated automaton can be directly used as input TA for enforcement monitor synthesis in the EME module. Moreover, it can also be directly used in the UPPAAL tool.

Supported patterns. Currently, the tool supports generation of automata for the requirements of the following forms:

- In every consecutive time interval of "*time constraint constant*" time units, there are no more than "*complexity constant*" actions belonging to "*action set 1*". The properties modeling requirements of such form belong to the "absence" pattern.
- A sequence of "*complexity constant*" actions from *action set 1* enable actions belonging to *action set 2* after a delay of at least "*time constraint constant*" time units. The properties modeling requirements of such form belong to the "precedence" (more specifically called as precedence with a delay) pattern.
- There should be "*complexity constant*" consecutive actions belonging to *Action set 1* which should be immediately followed by an action from *action set 2* within "*time constraint constant*" time units. The properties modeling requirements of such form belong to the "existence" (more specifically called as timed bounded existence) pattern.

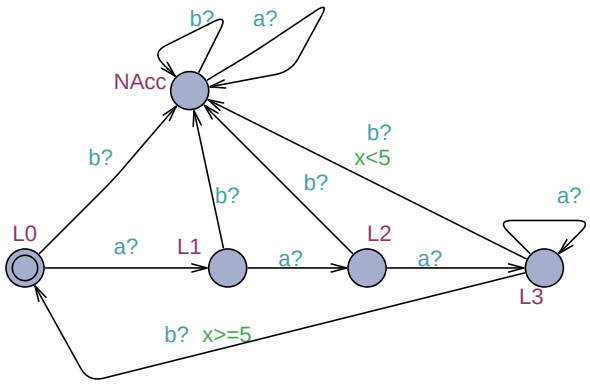


Figure 6.4: Automaton belonging to the precedence pattern.

Examples We now present some TAs generated using the *TA Generator* functionality of the GTA tool.

Figures 6.4 and 6.5 show how the TAs look when viewed in the UPPAAL tool.

– The TA in Figure 6.5 defines the requirement *"In any time interval of "10" t.u., there are no more than "3" "a" actions"*. The values of the input parameters provided to the tool, to generate the TA are the following: *pattern*= absence, *complexity constant*=3, *time constraint constant*=10, *action set 1*= {a} and *action set 2*= { b}.
– The TA in Figure 6.4 defines the requirement *"A sequence of "3" "a" actions enables action "b" after a delay of at least "5" time units "*. The values of the input parameters provided to the tool, to generate the TA are the following: *pattern*= precedence, *complexity constant*=3, *time constraint constant*=5, *action set 1*= {a} and *action set 2* ={b}.

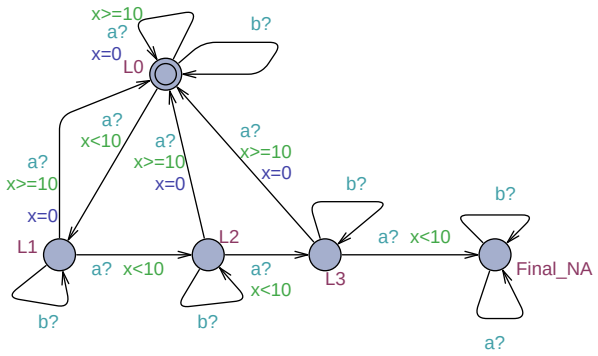


Figure 6.5: Automaton belonging to the absence pattern.

6.2.1.2 Combining timed automata

The *Boolean Operations* module builds a timed automaton by combining two timed automata provided as input, using Boolean operations. The required inputs and the

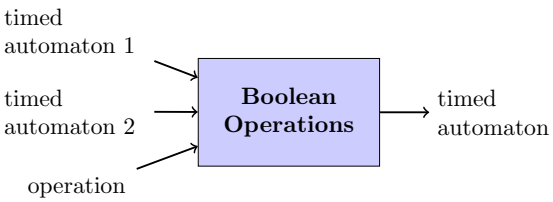
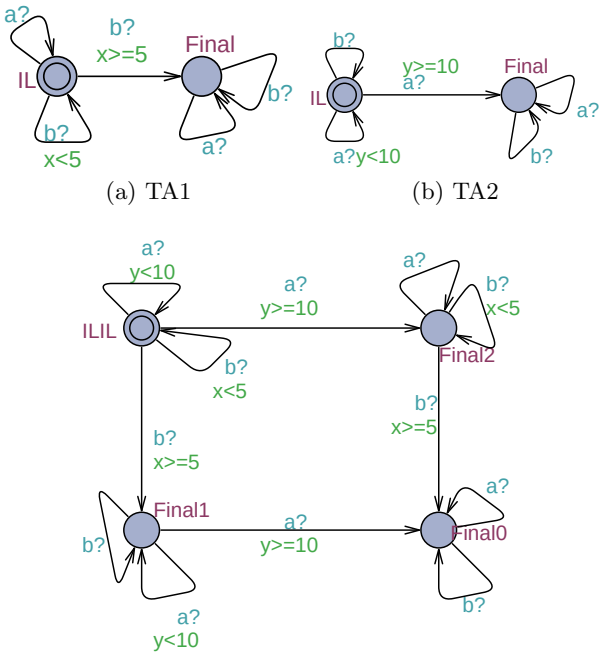


Figure 6.6: Boolean operations.

output of the *Boolean Operations* module is shown in Figure 6.7. *Timed automaton 1* and *Timed automaton 2* are the two (complete and deterministic) timed automata which we want to combine, and *operation* is the operation we want to perform on the input timed automata. The input and output timed automata are UPPAAL models stored as .xml.

The *Union* and *Intersection* operations are supported. The *Union* and *Intersection* operations are performed by building the synchronous product of the two input timed automata, where each location in the resulting automata is a pair (with one location



(a) TA1 (b) TA2

(c) Union of TA1 and TA2

Figure 6.7: Example: Combining TAs using Boolean operations.

from each input automaton), and the guards in the resulting automaton are obtained by intersection of the guards in the input automata.

For *Union* operation, accepting locations are the pairs where at least one location is accepting in the input automata, and for *Intersection* operation, both the locations in the corresponding input automata are accepting.

For formal definitions and more details on performing *Union* and *Intersection* operations on complete and deterministic TA, see Chapter 4, Definition 4.3.5.

Example Let us now see an example of the resulting TA obtained after combining two TAs using the *Boolean Operations* functionality.

The two input TAs are shown in Figure 6.7a and Figure 6.7b. Figure 6.7c shows the resulting TA after combining the two input TAs using Union operation.

6.2.2 Identifying the class of a timed automaton

As described in Chapter 4 (see Definition 4.3), a *safety* property informally means that nothing bad will ever happen, and a *co-safety* (or guarantee) property means that something good will eventually happen within a finite amount of time. For *safety* properties, whenever a sequence satisfies a property, all its prefixes should satisfy the property. If a sequence satisfies a *co-safety* property, then any possible extension of this sequence should satisfy the property. A TA defining a safety (co-safety)property is said to be a safety (co-safety) TA.

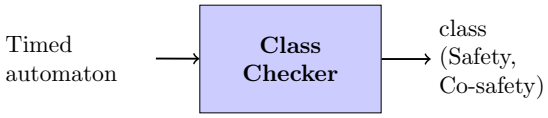


Figure 6.8: Class checker.

In a safety TA, transitions are not allowed from non-accepting to accepting locations. In a co-safety TA, transitions are not allowed from accepting to non-accepting locations. For more explanation and formal definitions of safety and co-safety properties (TAs), see Definition 4.4.

The *Class Checker* takes as input a timed automaton (described using UPPAAL syntax) and checks the class of the property defined by the TA. It answers *safety* (if the constraints of a safety TA are satisfied), *co-safety* (if the constraints of a co-safety TA are satisfied) and "*Other*" if neither safety TA nor co-safety TA constraints are satisfied.

6.2.3 EME module

The EME module consists of the implementation of the enforcement algorithms described in Section 6.1 and an experimentation framework in order to:

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

The EME module supports all regular properties defined by deterministic timed automata. Suppression of events is not taken into account in the tool, since the implementation of the update function follows the update function proposed in [PFJM14b]. To handle suppression, the update function should be adapted following the definition in Section 5.5.2. When an accepting state is not reachable from the current state upon the given input subsequence, the update function should check whether all the reachable state are `bad` or not.

The architecture of the EME module is depicted in Figure 6.2. Module Main Test Method uses module Trace Generator that provides a set of input traces to test the module Store. It sends each sequence to module Store, and keeps track of the result returned by the Store module for each trace. Module Trace Generator takes as input the alphabet of actions, the range of possible delays between actions, the desired number of traces, and the increment in length per trace. For example, if the number of traces is 5 and the increment in length per trace is 100, then 5 traces will be generated, where the first trace is of length 100 and the second trace of length 200 and so on. For each event, module Trace Generator picks an action (from the set of possible actions), and a random delay (from the set of possible delays) using methods from the Python random module.

Module Store takes as input a property (defined as a TA) and one trace, and returns the total execution time of the update function to process the given input trace. The TA defining the property is a UPPAAL [LPY97] model written in XML. Module Store uses the pyuppaal library to parse the UPPAAL model (input property), and the UPPAAL DBM library to implement the update function. [1]

6.3 Performance Evaluation of Function update

Examining the algorithms, one can observe that the steps in the algorithm of the DumpProcess of monitors are algorithmically simple and lightweight from a computational point of view. Regarding the StoreProcess function, their most computationally intensive step is the call to function update. We thus focus on this function in the evaluation.

Experiments were conducted on an Intel Core i7-2720QM at 2.20GHz CPU, with 4 GB RAM, and running on Ubuntu 12.04 LTS. The reported numbers are mean values over 10 runs and are represented in seconds.

Now, the properties used in our experiments are described and the results of the performance analysis are discussed. The properties follow different patterns [GL06], and belong to different classes. They are inspired from the properties introduced in Example 4.3. They are recognized by one-clock timed automata since this is a limitation of our current implementation. We however expect the trends exposed in the following to be similar when the complexity of automata grows.

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

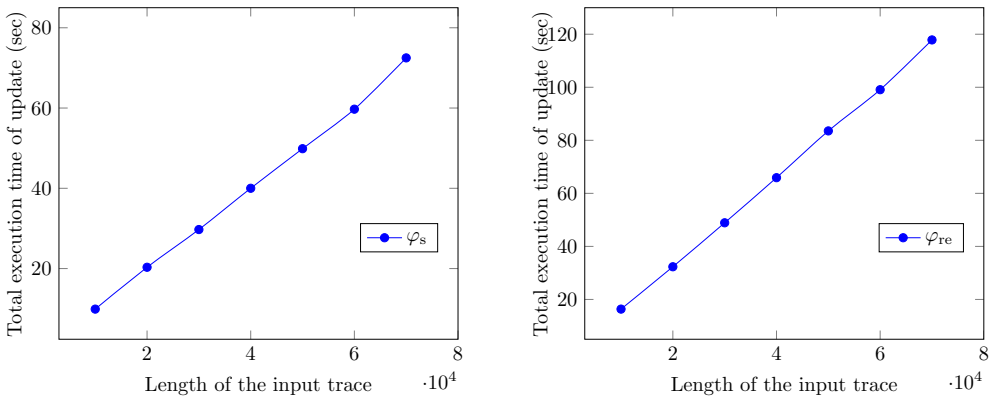
Results and analysis Results of the performance analysis for our example properties are reported in Table 6.1. The entry t_update indicates the total execution time of function update.

From the results presented in Table 6.1, as expected for the safety property (φ_{s}), we can observe that t_update increases linearly with the length of the input trace. This behavior is also clearly shown by the curve in Figure 6.9a showing the total time taken by the update function versus the length of the input trace. Moreover, the time taken per call to update (which is $^{t\_\text{update}}/_{|\text{tr}|}$) does not depend on the length of the trace. This behavior is as expected for a safety property: function update is always called with only one event which is read as input (the internal buffer σ_{mc} remains empty).

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

Table 6.1: Performance analysis of enforcement monitors

φ_{s}		φ_{re}		φ_{cs}	
|tr|	t_update	|tr|	t_update	|tr|	t_update
10,000	9.895	10,000	16.354	100	3.402
20,000	20.323	20,000	32.323	200	13.583
30,000	29.722	30,000	48.902	300	29.846
40,000	40.007	40,000	65.908	400	53.192
50,000	49.869	50,000	83.545	500	82.342
60,000	59.713	60,000	99.088	600	120.931
70,000	72.494	70,000	117.852	700	169.233



(a) For φ_{s}. (b) For φ_{re}.

Figure 6.9: Length of the input trace (Vs) total execution time of update.

Consequently, the state of the TA is updated after each event, and after receiving a new event, the possible transitions leading to a good state from the current state are explored.

For the co-safety property (φ_{cs}), the considered input traces are generated in such a way that they can be corrected only upon the last event. From the results presented in Table 6.1, also clearly shown by the curve in Figure 6.10, notice that t_update is now not linear. Moreover, the average time-per-call to function update (which is $t\text{\_update}/|\text{tr}|$) increases with |tr|. For the considered input traces, this behavior is as expected for a co-safety property because the length of the internal buffer σ_{mc} increases after each event, and thus function update is invoked with a growing sequence.

For the regular property (φ_{re}), the considered input traces are generated in such a way that it can be corrected every two events. Consequently, function update is invoked with either one or two events. For the considered input traces, the time taken per call to update (which is $t\text{\_update}/|\text{tr}|$) does not depend on the length of the trace. For

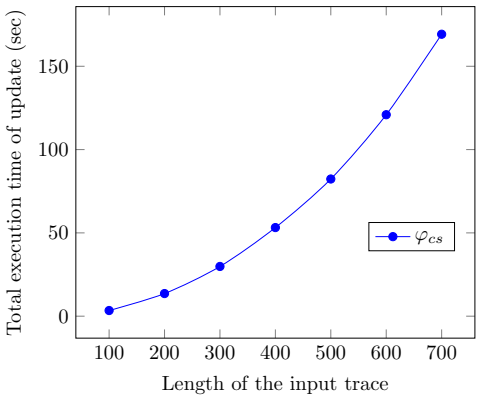


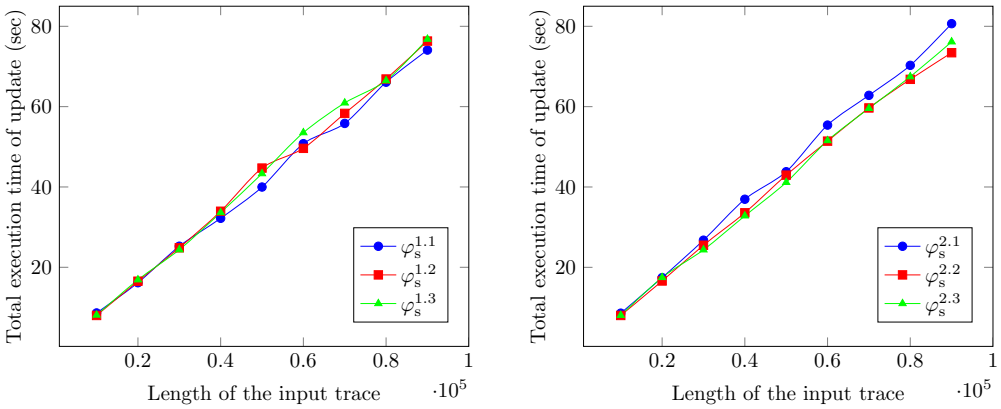
Figure 6.10: Length of the input trace (Vs) total execution time of update for φ_{cs}.

input traces of same length, the value of t_update is higher for φ_{re} than the value of t_update for φ_{s}, which can be noticed from Figures 6.9aand 6.11b. This stems from the fact that, for a safety property, function update is invoked only with one event.

6.4 Implementation of Simplified Algorithms for Safety Properties

The algorithms presented in this chapter are general algorithms for all regular properties. In Section 5.4, we saw that when only safety properties are considered, the functional definition can be simplified, and consequently the enforcement monitor and algorithms (see Remark 6.2) can be also simplified. By exploring the structure of a given TA, it is possible to determine whether property defined by the TA belongs to a subclass of regular properties (either safety or co-safety) properties. It has been also implemented in the GTA module, discussed earlier in this chapter in Section 6.2.3. In addition to the input TA, as shown in Figure 6.2, class information can also be provided as input to the EME module, and using this information, we can invoke either the simplified implementation (in case if the input TA is safety) or the general one.

The simplified enforcement algorithm for safety properties is also implemented, and experiments were conduced using several safety properties. We again focus on benchmarking the update function of the StoreProcess using the same experimental setup described in Sections 6.2.3 and 6.3. We have chosen to compute the average values over 10 runs because, for all metrics, with 95% confidence, the measurement error was less than 1%. For example, referring to Table 6.2, for safety property $\varphi_{\text{s}}^{1.1}$, the mean value of the total execution time of the update function is 8.6306 seconds, and the error is 0.018 seconds. Thus with 95% confidence, the execution time of update for input trace of length 10000 lies within the interval $[8.6126, 8.6486]$ (seconds). For the average time per call, as shown in Table 6.2, $\varphi_{\text{s}}^{1.1}$, the mean value is 0.863 milliseconds, and the error is 0.005 milliseconds. Thus, with 95% confidence, the average time per call to update, for input trace of length 10000 lies within the interval $[0.858, 0.868]$



(a) For φ_s^1. (b) For φ_s^2.

Figure 6.11: Length of the input trace (Vs) total execution time of update.

(milliseconds).

Performance evaluation of the update **function for safety properties.** We describe the properties used in our experiments and discuss the results of the performance analysis. The considered safety properties follow different patterns [GL06].

Property φ_s^1 belongs to the absence pattern. It expresses that *"There cannot be n or more a-actions in every k time units"*, where n is a parameter of the pattern. Following this pattern, the considered properties are $\varphi_s^{1.1}$, $\varphi_s^{1.2}$ and $\varphi_s^{1.3}$, each varying in the value of n: $n = 2$ for $\varphi_s^{1.1}$, $n = 10$ for $\varphi_s^{1.2}$, $n = 20$ for $\varphi_s^{1.3}$. Property φ_s^2 belongs to the precedence pattern. It expresses that *"A sequence of n a-actions enables action b after a delay of k time units"*. Following this pattern, the considered properties are $\varphi_s^{2.1}$, $\varphi_s^{2.2}$ and $\varphi_s^{2.3}$, each varying in the value of n: $n = 1$ for $\varphi_s^{2.1}$, $n = 5$ for $\varphi_s^{2.2}$, and $n = 10$ for $\varphi_s^{2.3}$.

Results and analysis. Results of the performance analysis of our running example properties are reported in Table 6.2. The entry t_update indicates the total execution time of the update function, and the entry t_avg is the average time per call. From the results presented in Table 6.2, as expected for safety properties, we can observe that the time taken per call to update is independent on the length of the trace. This behavior is as expected: since we update the state of the TA after each event, and after receiving a new event, we explore the possible transitions leading to a good state from the current state. Moreover, from the curves shown in Figure 6.11, notice that, for a given trace length, the execution time of update is similar for the two patterns and their variants in size.

From the results presented in Tables 6.1 and 6.2 we can notice that for enforcing safety properties, using the simplified algorithm gives better performance. The

Table 6.2: Performance analysis of enforcement monitors for safety properties

	$\varphi_s^{1.1}$		$\varphi_s^{1.2}$		$\varphi_s^{1.3}$	
$\|tr\|$	t_update	t_avg	t_update	t_avg	t_update	t_avg
10,000	8.6306	0.000863	8.008	0.00080	8.106	0.000810
20,000	16.157	0.000807	16.538	0.000828	16.887	0.000844
30,000	25.251	0.000841	24.855	0.000828	24.3794	0.00812
40,000	32.199	0.000804	33.947	0.000848	33.619	0.000840
50,000	39.982	0.000799	44.704	0.000854	43.314	0.000866
60,000	50.785	0.000846	49.616	0.000826	53.521	0.000892
70,000	55.821	0.000797	58.317	0.000833	60.928	0.000870
80,000	66.080	0.000826	66.876	0.000835	66.461	0.000830
90,000	74.082	0.000823	76.327	0.000848	76.807	0.000853
	$\varphi_s^{2.1}$		$\varphi_s^{2.2}$		$\varphi_s^{2.3}$	
$\|tr\|$	t_update	t_avg	t_update	t_avg	t_update	t_avg
10,000	8.589	0.000858	8.019	0.000801	8.050	0.000805
20,000	17.435	0.000871	16.603	0.000830	17.472	0.000873
30,000	26.760	0.000892	25.507	0.000850	24.353	0.000811
40,000	36.956	0.000923	33.576	0.000839	32.811	0.000820
50,000	43.806	0.000876	42.955	0.000859	41.141	0.000822
60,000	55.410	0.000923	51.417	0.000856	51.550	0.000859
70,000	62.816	0.000897	59.677	0.000852	59.572	0.000851
80,000	70.282	0.000878	66.800	0.000835	67.450	0.000843
90,000	80.659	0.000896	73.423	0.000815	76.137	0.000845

time taken per call to update reduces by around 0.2 milliseconds using the simplified algorithm.

6.5 Discussion and Summary

On precision. In theory, the delays between actions and the optimal delay computed by the update function are real numbers. In the implementation, in order to compute optimal delay, we need to set precision.

We use UPPAAL [LPY97] to model the input TA, and some UPPAAL libraries to realize the algorithms. In UPPAAL, only integers can be used to compare the values of clocks in the guards. But, in practice, we may have to use real-numbers to express requirements and timing constraints. This issue can be handled by setting the precision of real-numbers, and representing values on guards with equivalent integers. For example, if we set the precision with four digits after the decimal point, 0.0024

can be represented as 24, and 5.0012 can be represented as 50012. Note that having a large integer value on a guard such as in $x > 50000$ is not an issue with region and zone computations, as computation is done on-the-fly. After each event, we check for possible paths from the current state.

Summary. In this chapter, algorithms illustrating how enforcement mechanisms described in Chapter 5 can be implemented, are explained. The proposed algorithms are also implemented, demonstrating the practical feasibility of our theoretical results.

A tool-chain called as TIPEX developed in order to evaluate the performance of enforcement monitors is described. The Enforcement Monitor Evaluation (EME) module consists of functionalities to synthesize enforcement monitors from a TA, and other functionalities such as a trace generator required to evaluate the performance of enforcement monitors.

As far as we know, there is no available tool, which helps in formalizing real-time requirements. The Timed Automata Generator (GTA) module consists of features to automatically generate timed automata from some input data such as the actions, pattern, and time constraint constant. As shown in some examples, the input data which GTA requires to generate the property can be easily noticed from the informal textual description of the requirement we want to formalize. Moreover, GTA support other additional features such as combining TAs using Boolean operations (which facilitates to construct complex requirements from basic ones), and identifying the class of a given TA.

Assessing the performance of runtime enforcement monitors is crucial in a timed context as the time when a action happens influences satisfaction of the property. We also evaluated the performance of enforcement monitors for several properties, and considering very long input executions. As our experiments in Sections 6.3 and 6.4 show, the computation time of the monitor upon the reception of an event is relatively low. Moreover, given some average computation time per event and a property, one can determine easily whether the computation time is negligible or not. For example, for safety properties, one can see that, on the used experimental setup, the computation time of the update function is below 1ms. By taking guards with constraints using integers above 0.1s, one can see that the computation time can be negligible in some sense as the impact on the guard is below 1%, and makes the overhead of enforcement monitoring acceptable.

In Chapter 7, we introduce a model called as Parametrized Timed Automata with Variables (PTAV) which is an extension of timed automata, allowing to express much richer requirements. We also present how the enforcement mechanism presented in Chapter 5 can be extended to enforce properties expressed as PTAVs. Moreover, we also discuss how the algorithms and the implementation discussed in the chapter can be extended to PTAVs.

Chapter 7

Runtime Enforcement of Parametric Timed Properties

7.1 Overview

Many theoretical frameworks have been proposed for the runtime enforcement of high-level specifications on systems (see [FMFR11] or [Fal10] for an overview). In these enforcement frameworks, a specification is formalized as a propositional property (i.e., a set of words over a propositional alphabet) and an execution is a word over the considered alphabet. In Chapter 5, we also saw an enforcement framework for propositional timed properties. 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 network security, Runtime Enforcement (RE) monitors can detect and prevent Denial-of-Service (DoS) attacks. In resource allocation, RE monitors can ensure fairness. In addition to timing constraints, specifications in these domains also express data-constraints over the received events.

In timed specifications, the absolute time of occurrence of events matters, i.e., it influences satisfiability of the specifications by the system (or an execution of the system). In Chapter 5, we saw how enforcement monitors can be synthesized for requirements with timing constraints. But, requirements with both time and data constraints cannot be specified using timed automata. 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. [CR09, BFH+12]). 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.*

In parametric case, actions also carry some data. For example, let the set of actions $\Sigma = \{req, error, ok\}$, and $(2, req(1)) \cdot (5.3, req(3)) \cdot (6.7, req(1))$ is a sequence of events over Σ where each event is a tuple containing an action and its occurrence date. In this example, *req* action carries client identifier information. So, the first and third *req* actions are from client 1, and the second *req* action is from client 3.

To express these properties, we need features to express constraints over time and data. Moreover, the enforcement monitor running on 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 chapter, we make one step towards practical runtime enforcement by considering event-based specifications where i) absolute time of events matters and ii) events carry data values from the monitored system. We refer to this problem as *enforcement monitoring for parametric timed specifications*.

Similar to the enforcement of timed properties described in Chapter 5, because of the timing feature, we consider enforcement monitors as *time retardants* which have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate. Following the compositional approach (the so-called "plugin approach") introduced in [CR09], for a parametric timed specification, the input trace is sliced according to the parameter value of events, and redirected to the appropriate monitor instance. 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 chapter extends runtime enforcement for *non*-parametric (i.e., propositional) timed properties, presented in Chapter 5. To formalize richer requirements (with both time and data constraints), we introduce Parameterized Timed Automata with Variables (PTAVs), an extension of Timed Automata (TAs) with internal and external variables. Internal variables are used for internal computation, such as to keep track of the number of actions received. External variables are used to model transfer of data along with events from the monitored system. The expressiveness features of PTAVs 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. We then extend enforcement for TAs to enforcement for PTAVs. Furthermore, we show how these application domains can benefit from using runtime enforcement monitors synthesized from requirements formalized as PTAVs (Section 7.5). The enforcement monitors synthesized from PTAVs are able to ensure several requirements in the considered application domains. Our experiments validate our choice on the expressiveness of PTAVs and assess the efficiency of obtained enforcement monitors.

The results that we present in this chapter summarizes results in [PFJM14a], but we adapt the presentation with elements from [FJMP14]. For example, timed words are formalized with delays instead of dates in [PFJM14a], but here we use dates similar to the presentation in [FJMP14], and in our previous chapters.

7.2 Preliminaries to Runtime Enforcement of Parametric Timed Properties

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers, and Σ a finite alphabet of *actions*. An *event* is a pair (t, a), where $\text{date}((t, a)) \stackrel{\text{def}}{=} t \in \mathbb{R}_{\geq 0}$ is the absolute time at which the action $\text{act}((t, a)) \stackrel{\text{def}}{=} a \in \Sigma$ occurs. As in Chapter 5, monitors input and output timed words.

Until Chapter 5, a *timed word* over a finite alphabet Σ is a finite sequence of events $\sigma = e_1 \cdot e_2 \cdots e_n$, where $(t_i)_{i \in [1,n]}$ is a non-decreasing sequence in $\mathbb{R}_{\geq 0}$. Similarly, in the parametric case, an event is still a tuple with an action and a date where dates in a timed word are non-decreasing, but actions carry data, that is $e_i = (t_i, a_i(\pi_i, \eta_i))$ where 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 ranging over a countable set $\mathcal{D}_p$, identifying the monitor instance to which the action should be fed as input, and $\eta_i \in \mathcal{D}_V$ is a vector of values of a tuple of variables V. We denote by $\text{date}\left((t, a\,(\pi, \eta))\right) \stackrel{\text{def}}{=} t$ the projection on date, by $\text{act}\left((t, a\,(\pi, \eta))\right) \stackrel{\text{def}}{=} a$ the projection on action, by $\sigma_{[i]}$, the i^{th} event, and by $\text{time}(\sigma)$ the total duration of σ, i.e., the date of the last event in σ. $|\sigma|$ 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. For example, if $\sigma = (0.5, a(1, \eta_1)) \cdot (1.2, a(2, \eta_2)) \cdot (2, a(1, \eta_3)) \cdot (2.4, a(2, \eta_4))$, then $\sigma \downarrow_1 = (0.5, a(1, \eta_1)) \cdot (2, a(1, \eta_3))$ and $\sigma \downarrow_2 = (1.2, a(2, \eta_2)) \cdot (2.4, a(2, \eta_4))$.

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:

- $\text{merge}\{\epsilon\} = \epsilon$;
- $\text{merge}\{\sigma_1, \ldots, \sigma_n\} = \text{merge}\{\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n\}$ if $\sigma_i = \epsilon$;
- $\text{merge}\left\{(t_1, a_1(1)) \cdot \sigma_1, \ldots, (t_n, a_n(n)) \cdot \sigma_n\right\} =$
  let i s.t. $t_i = \min\{t_j \mid j \in [1, n]\}$ in
  $(t_i, a_i(i)) \cdot \text{merge}\left\{(t_1, a_1(1)) \cdot \sigma_1, \ldots, \sigma_i, \ldots, (t_n, a_n(n)) \cdot \sigma_n\right\}$.

The merge of a set containing only 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 date (say the i-th sequence), and, the remainder of the sequence is the merge of the remainder of the i-th sequence with the previous sequences.

For example, if $\sigma_1 = (0.5, a(1, \eta_{1.1})) \cdot (1.5, a(1, \eta_{1.2}))$ and $\sigma_2 = (0.8, a(2, \eta_{2.1})) \cdot (1.2, a(2, \eta_{2.2}))$ then $\text{merge}\{\sigma_1, \sigma_2\} = (0.5, a(1, \eta_{1.1})) \cdot (0.8, a(2, \eta_{2.1})) \cdot (1.2, a(2, \eta_{2.2})) \cdot (1.5, a(1, \eta_{1.2}))$.

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 definitions of observation of a timed word σ at time t, delaying order $\succcurlyeq_d$, and delaying subsequence order $\lhd_d$ defined in Section 5.2 are adapted in a straightforward manner.

7.3 Parametric 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 (PTAV). PTAVs are partly inspired from Input-Output Symbolic Transition Systems (IOSTS) of [RBJ00], Timed Input-Output Symbolic Transition Systems (TIOSTS) of [AMJM11], parametric trace slicing of [CR09] and Quantified Event Automata of [BFH+12]. The features of PTAVs have been chosen to fit a balance between expressiveness (to express requirements from some application domains - see Section 7.5) and runtime efficiency.

7.3.1 Syntax and semantics of a PTAV

A PTAV 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 $V = (v_1, \ldots, v_n)$, $\mathcal{D}_V$ 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 $\mathcal{D}_V \to \{\texttt{tt}, \texttt{ff}\}$, and can also be seen as the subset of $\mathcal{D}_V$ which maps to $\texttt{tt}$. 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 $\chi : X \to \mathbb{R}_{\geq 0}$. 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 7.1 (Syntax of PTAVs) *A PTAV is a tuple* $\mathcal{A}(p) = (p, V, C, \Theta, L, l_0, F, X, \Sigma_p, \Delta)$ *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 $F \subseteq L$ the set of accepting locations;*
- *Σ is a non-empty finite set of actions, and an action $a \in \Sigma$ has a signature $sig(a) = (t_0, t_1, \ldots, t_k)$ 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 (l, a, p, c, G, A, l') 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 $c = (c_1, \ldots c_k)$ 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*
    - *$A^D : \mathcal{D}_V \times \mathcal{D}_{sig(a)} \to \mathcal{D}_V$ defines the evolution of internal variables.*
    - *$A^X \subseteq X$ is the set of clocks to be reset.*

A PTAV $\mathcal{A}(p)$ 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 $\mathcal{A}(\pi)$. PTAVs allow to describe a set of identical timed automata extended with internal and external variables that only differ by the value of p. In Section 5.3 we explain how a parametric enforcement monitor generated from a PTAV 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 7.1 (Multiple parameters) *For the sake of simpler notations, PTAVs are presented with only one parameter p. PTAVs 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 $\mathcal{A}(p) = (p, V, C, \Theta, L, l_0, F, X, \Sigma, \Delta)$ be a PTAV. For a value π of parameter p, the semantics of instance $\mathcal{A}(\pi)$ is a timed transition system, where a state consists of a location, and 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]

1. Considering a tuple of parameters instead of a single parameter does not have any effect on the semantics of a PTAV.

Definition 7.2 (Semantics of PTAVs) *For a value π of p, the semantics of $\mathcal{A}(\pi)$ is a timed transition system $[\![\mathcal{A}(\pi)]\!] = (Q, q_0, Q_F, \Gamma, \rightarrow)$, defined as follows:*
- *$Q = L \times \mathcal{D}_V \times (X \rightarrow \mathbb{R}_{\geq 0})$, is the set of states of the form $q = (l, \nu, \chi)$ where $l \in L$ is a location, $\nu \in \mathcal{D}_V$ is a valuation of internal variables, χ is a valuation of clocks;*
- *$Q_0 = \{(l_0, \nu, \chi_{[X \leftarrow 0]}) \mid \Theta(\pi, \nu) = \texttt{tt}\}$ is the set of initial states;*
- *$Q_F = F \times \mathcal{D}_V \times (X \rightarrow \mathbb{R}_{\geq 0})$ 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 $(l, \nu, \chi) \xrightarrow{(\delta, a(\pi,\eta))} \langle l', \nu', \chi' \rangle$ such that $\exists (l, a, p, c, G, A, l') \in \Delta$, with $G^X(\chi + \delta) \wedge G^D(\nu, \pi, \eta)$ evaluating to $\texttt{tt}$, $\nu' = A^D(\nu, \pi, \eta)$ and $\chi' = (\chi + \delta)[A^X \leftarrow 0]$.*

The set of timed words over Σ where the actions carry parameter value and other data is denoted by $\text{tw}(\Lambda)$. A *run* ρ of $[\![\mathcal{A}(\pi)]\!]$ from a state $q \in Q$ triggered at time $t \in \mathbb{R}_{\geq 0}$ over a timed *trace* $w_t = (t_1, a_1(\pi, \eta_1)) \cdot (t_2, a_2(\pi, \eta_2)) \cdots (t_n, a_n(\pi, \eta_n))$ is a sequence of moves in $[\![\mathcal{A}(\pi)]\!]$: $\rho = q \xrightarrow{(\delta_1, a_1(\pi,\eta_1))} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n, a_n(\pi,\eta_n))} q_n$, for some $n \in \mathbb{N}$, satisfying the condition $t_1 = t + \delta_1$ and $\forall i \in [2, n], t_i = t_{i-1} + \delta_i$. The set of runs from the initial state $q_0 \in Q$, starting at $t = 0$ is denoted $\text{Run}(\mathcal{A}(\pi))$ and $\text{Run}_{Q_F}(\mathcal{A}(\pi))$ denotes the subset of those runs *accepted* by $\mathcal{A}[\![\pi]\!]$, i.e., ending in an accepting state $q_n \in Q_F$. We note $q \xrightarrow{w_t} q_n$ in this case, and generalize to $q \xrightarrow{w_t} P$ when $q_n \in P$ for a subset P of Q. We note $\mathcal{L}(\mathcal{A}(\pi))$ the set of traces of $\text{Run}(\mathcal{A}(\pi))$. We extend this notation to $\mathcal{L}_{Q_F}(\mathcal{A}(\pi))$ as the traces of runs in $\text{Run}_{Q_F}(\mathcal{A}(\pi))$. We thus say that a timed word is accepted by $\mathcal{A}(\pi)$ if it is the trace of an accepted run.

Safety PTAVs. Safety properties state that "nothing bad should ever happen". Similarly to timed safety properties (See Definition 4.3), a parametric timed safety property is defined as a set of prefix-closed languages which are parameterized by p. In this chapter, example properties considered are parametric timed safety properties that can be represented by PTAVs. [2]

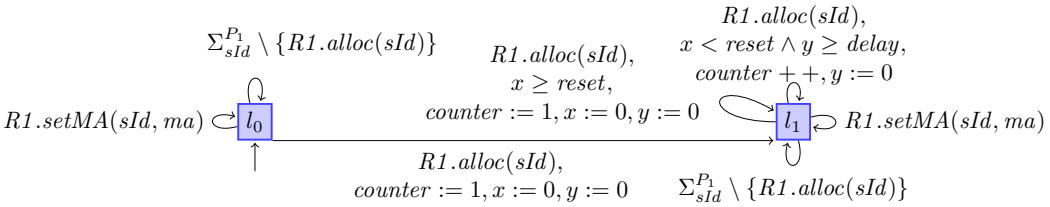
Definition 7.3 (Safety PTAV) *A PTAV $\mathcal{A}(p) = (p, V, C, \Theta, L, l_0, F, X, \Sigma, \Delta)$ is said to be a* safety *PTAV if $l_0 \in F \wedge \nexists (l, a, p, c, G, A, l') \in \Delta : l \in L \setminus F \wedge l' \in F$.*

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

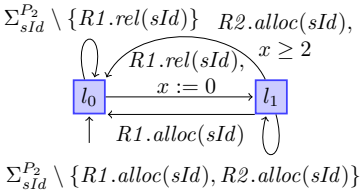
7.3.2 Defining properties using PTAVs: A motivating example

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 PTAVs. The PTAV shown in Figure 7.1a models the property *"There should be a dynamic delay between two allocation requests*

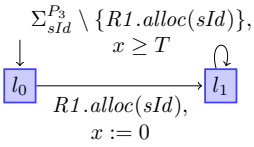
2. Note that the results also apply to any regular parametric timed property which can be specified using PTAV.



(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.

Figure 7.1: PTAVs for resource allocation.

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 PTAVs.

We now explain the different notions of the PTAV model through this example. The PTAV in Figure 7.1a 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 $(counter, reset, delay)$, and the tuple of external variables is the singleton $(maxAlloc)$. The variable *counter* is an integer incremented after each $R1.alloc(sId)$ event, and *maxAlloc* (ma) is an integer that defines the allowed number of $R1.alloc(sId)$ messages per each increment of the delay. Note that variable ma is an external variable since it is used to receive data via the action $R1.setMA(sId, ma)$. The variable *reset* defines the time period for resetting the counter. The delay to be introduced between allocation requests (kept track by variable *delay*) is computed dynamically based on the number of allocation requests. *delay* is defined as 0 if $counter < ma$ and int $\left(\frac{counter * sId}{ma}\right)$ otherwise.

7.4 Enforcement of Parametric Timed Properties

In Chapter 5, enforcement mechanisms are described at several levels for propositional timed properties. An enforcement function (see Section 5.4.2) dedicated to a desired property φ defines at an abstract level, for any input word σ, the output word $E_\varphi(\sigma)$ expected from an enforcement mechanism. An enforcement function E_φ for a property φ, specified by a TA $\mathcal{A}_\varphi$, is implemented by an enforcement monitor (EM), which provides a more operational view of the enforcement mechanism. An EM is defined as a transition system $\mathcal{E}$ (see Section 5.5.3). Algorithms in Section 6.1 describe

how an enforcement monitor can be realized using two processes running concurrently.

The techniques of the propositional case can be adapted to generate a parametric enforcement function (and monitor) for a parametric timed property φ_p with parameter p.

For a parametric timed property φ_p specified by a PTAV $\mathcal{A}(p)$, and a particular value π of p, φ_π is defined by the language $\mathcal{L}_{Q_F}(\mathcal{A}(\pi))$. It is straightforward to see that, for a given instance $\mathcal{A}(\pi)$, the constraints, functional definition and enforcement monitor remain the same as in the propositional case, except that now we consider semantics of PTAV. The input and output timed words belong to $\mathrm{tw}(\Lambda)$, and all the input and output events will carry the same parameter value π.

Taking (instantiated) variables into account. Actually, the main adaptation lies in the definition of the update function used by the enforcement monitor to compute optimal delays, taking into account the semantics of PTAVs instead of the semantics of TAs. Notice that the update function is computable as the parameter value π is a constant known at runtime (each action carries this information), the state q contains values of internal variables ν and location information, and the values of external variables η are known (data sent along with actions in addition to the value of the parameter).

It is then not hard to adapt the proofs of the propositional case in Chapter 5 to the following propositions:

Proposition 7.1 *Given a PTAV $\mathcal{A}(p)$ specifying a parametric timed 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, satisfies the physical **(Phy)**, soundness* **(Snd)**, *and transparency* **(Tr)** *constraints w.r.t. $\mathcal{A}(\pi)$, as per Definition 5.7.*

Proof 7.1 *The proof of Proposition 7.1 is a straightforward adaptation of the proof of Proposition 5.1.*

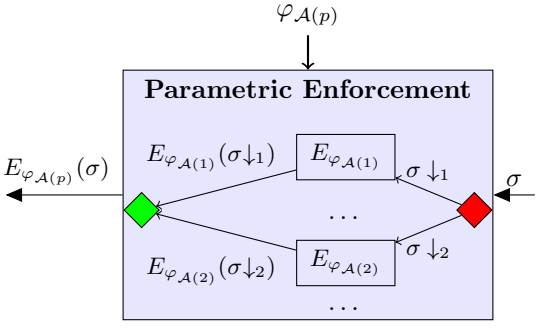


Figure 7.2: Global scenario.

Indexed enforcement monitors and slicing. The proposed strategy uses indexed enforcement monitors and slicing of [CR09] that is simplified in a way slicing is meaningful for runtime enforcement. [3]

As illustrated in Figure 7.2, 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.

Monitor instance for each parameter value. The monitor instance for each parameter value is exactly the same as the enforcement monitor in the propositional case that we saw in Chapter 5. All the constraints, functional definition, enforcement monitor definition and algorithms remain the same. What we additionally provide here is a mechanism to be able to create multiple monitor instances, and actions to carry some data. The value of the parameter in the event allows to decide to which monitor instance the event should be fed as input. This allows us to consider much richer specifications in our framework. Each monitor instance satisfies the physical, soundness, transparency and optimality constraints.

Now, let us see the global view, about how our enforcement mechanism transforms an input sequence (with events carrying different parameter values), using multiple enforcers each transforming a subsequence of the global input sequence, where the output of each enforcer is sound, transparent, optimal and satisfies physical constraint.

Definition 7.4 (Parametric Enforcement Function) *The enforcement function* $E_{\varphi_p}(\sigma) : \mathrm{tw}(\Lambda) \to \mathrm{tw}(\Lambda)$ *for PTAV* $\mathcal{A}(p)$ *is defined as:*

$$E_{\varphi_p}(\sigma) = \textit{let } n = |\mathcal{D}om(\sigma)| \textit{ in} \\ \textit{let } o_\pi = E_{\varphi_{\mathcal{A}(\pi)}}(\sigma{\downarrow_\pi}),\ \textit{for } \pi \in [1,n],\ \textit{in} \\ \big(\mathrm{merge}(\{o_1,\ldots,o_n\})\big).$$

The global output of the enforcement function is defined as the merge of the local outputs $(o_1,\ldots,o_n)$ produced by the enforcement functions synthesized for PTAV 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 $E_{\varphi_p}(\sigma)$ 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.

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 in previous events.

Let $Phy(E_\varphi, \sigma)$, $Snd(E_\varphi, \sigma)$ and $Tr(E_\varphi, \sigma)$ denote the physical, soundness and transparency constraints in the propositional case (See Definition 5.7). Consequently, the following definitions of parametric physical constraint, soundness, and transparency stem from the fact that, using an indexed strategy, enforcement monitors act and output events independently.

Definition 7.5 (Constraints on an parametric enforcement mechanism) *A parametric enforcement function $E_{\varphi_p} : \mathrm{tw}(\Lambda) \to \mathrm{tw}(\Lambda)$ for a PTAV $\mathcal{A}(p)$ with parameter p satisfies the following constraints:*

- ***physical constraint*** *means that if for any input timed word σ, for any possible value π of p, the output stream can only be modified by appending new events to its tail: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in \mathrm{tw}(\Lambda) : Phy_\pi(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$;*
- ***soundness*** *means that for any input timed word σ, for any possible value π of p, the output obtained from the projection of the input timed word on π, satisfies the property: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in \mathrm{tw}(\Lambda) : Snd_\pi(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$;*
- ***transparency*** *means that for any input timed word σ, for any possible value π of p, the output obtained from the projection of the input timed word on π is a delayed subsequence of the projection of the input timed word on π: $\forall \pi \in \mathcal{D}om(p), \forall \sigma \in \mathrm{tw}(\Lambda) : Tr_\pi(E_{\varphi_{\mathcal{A}(\pi)}}, \sigma{\downarrow_\pi})$;*

where predicates Phy Snd, and Tr are lifted to parametric traces.

Using the results from Chapter 5, and how parametric enforcers are built, we immediately get the following proposition:

Proposition 7.2 *Given a PTAV $\mathcal{A}(p)$ specifying a parametric timed safety property $\varphi_{\mathcal{A}(p)}$, the enforcement function E_{φ_p} as per Definition 7.4 satisfies physical, soundness, and transparency constraints with respect to $\mathcal{A}(\pi)$, as per Definition 7.5.*

Proposition 7.2 can be proved using Proposition 7.1, Definitions 7.4 and 7.5 and the results from Chapter 5.

Example 7.1 (Parametric soundness and transparency) *Consider the property shown in Figure 7.1a. Let the initial values of variables delay and reset be 5 and 100, respectively. Consider the input sequence $\sigma = (2, R1.alloc(1)) \cdot (3, R1.alloc(2)) \cdot (4, R1.alloc(1))$. $\mathcal{D}om(\sigma) = \{1, 2\}$, and thus we have two monitor instances and both are sound, and transparent. The projection of the input sequence on the actions carrying parameter value 1 is $\sigma{\downarrow_1} = (2, R1.alloc(1)) \cdot (4, R1.alloc(1))$, which is input to $E_{\varphi_{\mathcal{A}(1)}}$, and the output of $E_{\varphi_{\mathcal{A}(1)}}$ is $(2, R1.alloc(1)) \cdot (7, R1.alloc(1))$, satisfying soundness and transparency. The projection of the input sequence on the actions carrying parameter value 2 is $\sigma{\downarrow_2} = (3, R1.alloc(2))$, which is input to $E_{\varphi_{\mathcal{A}(2)}}$, and the output of $E_{\varphi_{\mathcal{A}(2)}}$ is $(3, R1.alloc(2))$, satisfying soundness and transparency.*

7.5 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 re-

quirements modeled as PTAVs from which we synthesize enforcement monitors that try to maintain the PTAV in accepting locations by increasing dates or suppressing events when necessary and possible.

Using PTAVs we obtain abstract and concise representations of the requirements. Enforcement monitors are a lightweight, modular, and flexible implementation of these requirements.

7.5.1 Resource allocation

Let us consider a common client-server model used in distributed applications and web servers (described in [FHTH10]). A system consists of three layers (see Figure 7.3): clients, services and shared resources. Clients send their requests to services and wait for their response.

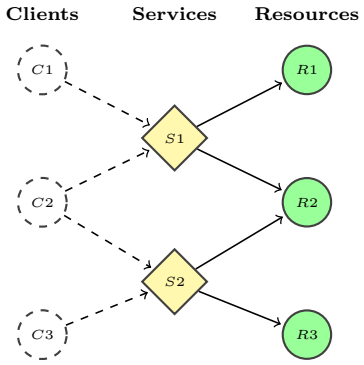


Figure 7.3: 3-layer model.

Requests to a service are stored in a FIFO queue, and a service processes them sequentially. In order to process a request, a service has to do some computation using some resources such as processors, files, and network connection managers. Concurrent accesses to shared resources by various services can 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. Runtime enforcement improves resource management and prevents such problems.

Leveraging runtime enforcement for fair resource allocation. In [FHTH10], 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 PTAVs. The event $R1.alloc(sId)$ (resp. $R2.alloc(sId)$) corresponds to the allocation of $R1$ (resp. $R2$), and $R1.rel(sId)$ (resp. $R2.rel(sId)$) corresponds to the release of $R1$ (resp. $R2$) in the session sId. One monitor instance is associated to each service instance. The requirements are listed below.

P1 *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.* The requirement is formalized with the PTAV in Figure 7.1a. The PTAV 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 events $\Sigma_{sId}^{P_1} = \{R1.alloc(sId)\}$. The tuple of internal variables is $(counter, reset, delay)$, and the tuple of external variables is the singleton $(maxAlloc)$, where *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; *incr* is a constant value that defines the delay's increment; *reset* defines the time period for resetting the counter and is reset after certain time period. The delay to be introduced between allocation requests (kept track by variable *delay*) is computed dynamically based on the number of allocation requests.

$$delay = \begin{cases} 0 & \text{if } counter < maxAlloc \\ \text{int}\left(\frac{counter}{maxAlloc}\right) & \text{otherwise} \end{cases}$$

The *delay* is 0 if the number of $R1.alloc(sId)$ events received is less (when $counter < maxAlloc$). Otherwise, it is defined as $\text{int}\left(\frac{counter}{maxAlloc}\right)$, where *counter* indicates the number of $R1.alloc(sId)$ events received. The clock x is used to reset the variable *counter*. In location l_0, upon receiving an $R1.alloc(sId)$ event, if $x \leq reset$, then a transition is made to location l_1, *counter* is incremented (and reset to 1 otherwise), and y is reset. In location l_1, upon receiving $R1.alloc(sId)$, if $x \leq reset$ and $y \geq delay$, then *counter* is incremented, and y is reset.

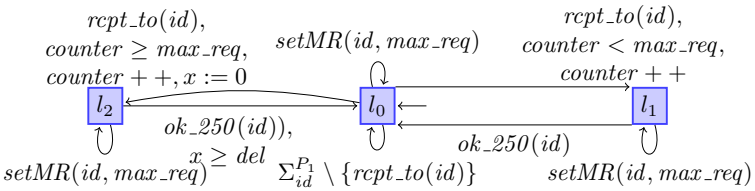
P2 *After releasing R*1*, there should be a delay of at least 2 t.u. before allocating R*2.
The requirement is formalized by the PTAV in Figure 7.1b. The set of events is $\Sigma_{sId}^{P_2} = \{R1.alloc(sId), R1.rel(sId), R2.alloc(sId), R2.rel(sId)\}$.

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
$\Sigma_{sId}^{P_3} = \{R1.alloc(sId), R1.op1(sId), R1.op2(sId), R1.op3(sId)\}$ where *op1*, *op2*, and *op3* are possible operations on a resource. The requirement is formalized by the PTAV in Figure 7.1c.

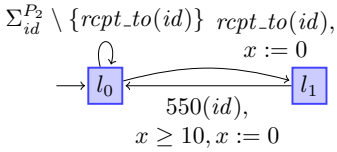
Using the indexed approach described in Section 7.4, for each requirement a parametric enforcement monitor can be derived from the requirement modeled as a PTAV formalizing the requirement. One monitor instance is associated to each service instance.

7.5.2 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 the sender and receiver email addresses with a MAIL_FROM message with sender's address as argument, and an RCPT_TO message with receiver's address as argument [HTJ03]. The server responds with an OK_250 message if the addresses are valid. The client should wait for the response from the server to transmit data [HTJ03].



(a) Compute delay dynamically



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

Figure 7.4: Robust mail servers

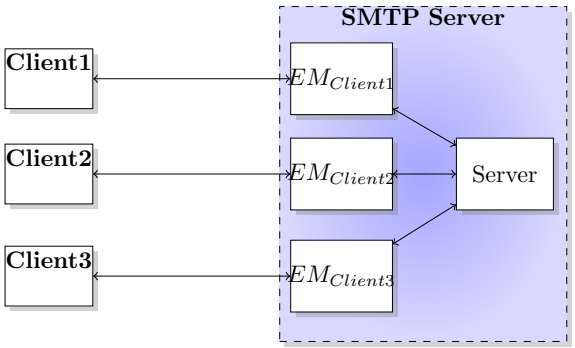


Figure 7.5: Architectural setting

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 [HTJ03]. 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. In [HTJ03], an implementation of tarpits is described, where the delay introduced depends on the number of request messages from a client. The expressivity of PTAVs 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.

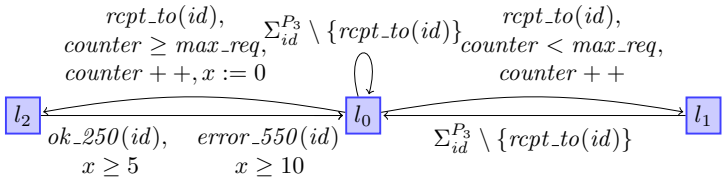


Figure 7.6: Robust mail servers: Different delays based on the response message

As shown in Figure 7.5, we consider an architectural setting where enforcement monitors run on a server. An instance of monitor is created for each new client. 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 PTAVs parameterized by a client identifier (id).

R1 *If the number of RCPT_TO messages from a client is greater than 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 PTAV in Fig. 7.4a with alphabet $\Sigma_{id}^{R_1} = \{rcpt\_to(id), \texttt{ok}, setMR(id, max\_req)\}$. The set of variables is $\{counter, max\_req, del\}$: *counter* is incremented after each $rcpt\_to(id)$ from the client, and *max_req* is a constant that defines the number of $rcpt\_to(id)$ messages per each increment of the delay. The delay is computed dynamically and depends on the number of received $rcpt\_to(id)$ messages. The delay *del* is defined as 0 if $counter < max\_req$ and $\text{int}(\frac{counter}{max\_req})$ otherwise. Upon receiving an $rcpt\_to(id)$ message, if $counter \leq max\_req$, the PTAV moves from l_0 to l_1. Otherwise, the PTAV goes to l_2, resetting the clock x. Upon receiving an ok, the PTAV moves from l_1 to l_0, or from l_2 to l_0 if $x \geq del$.

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 PTAV in Fig. 7.4b with alphabet $\Sigma_{id}^{R_2} = \{rcpt\_to(id), 550(id)\}$.

R3 *If the number of RCPT_TO messages is greater than 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 PTAV in Fig. 7.6 with alphabet $\Sigma_{id}^{R_3} = \{rcpt\_to(id), \texttt{ok}\}$. The set of used variables is $\{counter, max\_req\}$: *counter* is an integer incremented after each $rcpt\_to(id)$ message from the client, and *max_req* is an integer constant that defines the number of $rcpt\_to(id)$ messages allowed before introducing delays.

R4 This requirement is formalized by the PTAV in Fig. 7.7. Compared to the PTAV for R1, an additional feature is handled: *counter* is reset according to the computed delay, if there is sufficient time between two $rcpt\_to(id)$ messages from the client. To handle this feature, we also need an integer variable *min_D*, defining the sufficient delay between RCPT_TO messages to reset the counter.

The enforcement monitors synthesized from these requirements (modeled as PTAVs), when integrated with the server can prevent the server from processing a large number

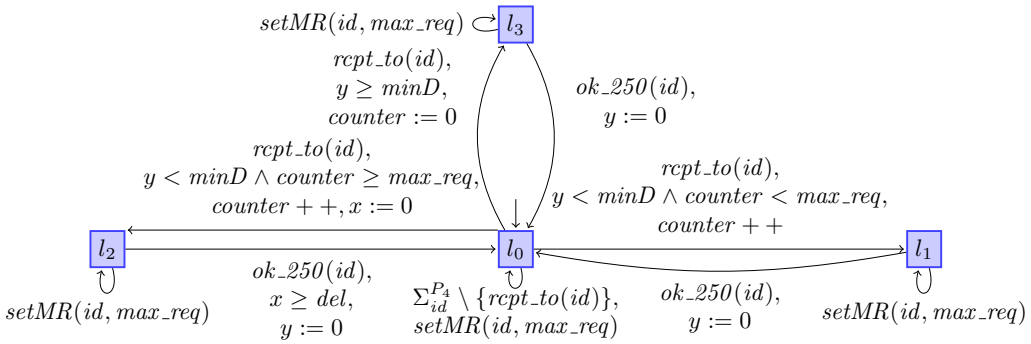


Figure 7.7: Robust mail servers: Decrease or Increase delay dynamically

of spam messages.

7.6 Implementation and Evaluation of Parametric Enforcement Monitors

A prototype tool in Python based on the algorithms proposed in Chapter 6 was implemented. 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 prototype is first to show feasibility and then to have a first assessment of performance. An enforcement monitor for a PTAV 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 ([LPY97]) and stored in XML. We use UPPAAL as a library to implement the update and post functions (see [PFJ+12]), and the PyUPPAAL library to parse the properties. [4]

7.6.1 The experimental setup

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 PTAV in Figure 7.1b. 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.

Enforcement monitors can be created and deleted on-the-fly. As explained in Section 5.3, some mechanism is needed to split the input trace based on parameter values. For slicing as in [CR09] but based on a single parameter, the simple procedure is described in Algorithm 4.

4. Given an input state and event, the post function computes the state reached in the underlying timed transition system of a PTAV.

Algorithm 4 FrontEndProcess

```
Enforcers ← {}
while tt do
  (t, a(π, ν)) ← await(event)
  e ← Enforcers.get(π)
  if (e ≠ NONE) then
    e.addEvent((t, a(π, ν)))
  else
    e = new Enforcer()
    Enforcers.add(π, e)
    e.addEvent((t, a(π, ν)))
  end if
end while
```

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 4 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.

7.6.2 Performance analysis

Results of performance analysis are presented in Table 7.1. 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 PTAVs (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.

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 PTAV instance.

Analysis. From Table 7.1, 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 7.1, we can 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.

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 7.1: Performance evaluation

7.7 Discussion and Summary

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. PTAVs are inspired from existing parametric formalisms for untimed properties [CR09]. In term of indexing, our framework uses a subset of the possibilities described in [CR09] 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 we considered 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 [CR09] is restricted to propositional formalisms to specify (indexed) plugin monitors while PTAV instances handle variables, guards, and assignments.

Summary. In this chapter, we introduced Parametrized Timed Automata with Variables (PTAV), which is an extension of timed automaton with a parameter, internal and external variables. PTAVs allow to formally define much richer requirements. In scenarios where we have multiple clients and a server, monitor instances for each client on the server can be created on-the-fly. Moreover, the formalism also allows messages

to carry data, and allows to express constraints both on time and data. We also saw that it is straightforward to extend the enforcement mechanism for timed automata described in Chapter 5 to PTAV. The features of PTAVs allow to specify requirements that we commonly encounter in several domains. In Section 7.5, we also saw how requirements for some applications domains can be defined as PTAVs.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Although several formal modeling and analysis techniques, and tools based on them have been developed over the years, scalability of the techniques is a central issue that has prevented their widespread adoption. On the one hand research efforts continue to handle issues such as state space explosion, light weight formal techniques such as runtime verification for validation of critical systems are also actively under research on the other hand.

Runtime verification is a formal verification technique, complementing the other formal verification techniques such as model checking. Runtime verification techniques allow to check whether a run of a system under scrutiny satisfies (or violates) a given correctness property. A verification monitor does not influence the system execution. Runtime verification techniques are lightweight and also do not require a formal model of the system, avoiding problems such as state explosion, since only a single execution of the system is considered.

Runtime enforcement extends runtime verification and refers to the theories, techniques, and tools aiming at ensuring the conformance of the executions of systems under scrutiny with respect to some desired property. Using an enforcement monitor, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies with a property (e.g., formalizing a safety requirement). A central concept in runtime verification and enforcement, is to generate monitors from some high-level specification of the property (which the monitor should verify (or enforce)).

Contributions. We developed formally based runtime enforcement mechanisms for requirements with real-time constraints. In this thesis, we described how to synthesize enforcement monitors from a formal description of the property. Timed automaton is the model we use to formally define a property from which an enforcement monitor is synthesized. In addition to defining and proving the enforcement mechanisms formally, prototypes based on the proposed mechanisms were implemented, to demonstrate prac-

tical feasibility of the proposed approach.

We also proposed a model called as PTAV, an extension of timed automata model with variables and parameters. The PTAV model allows to formalize richer specifications with both time and data constraints, and also allows actions to carry data. Enforcement mechanisms proposed for TA's have been extended to PTAV's. We also showed how enforcement monitors can be synthesized from PTAV's. Moreover, how enforcement monitoring is suitable in domains such as monitoring networks and preventing denial-of-service attacks is shown.

This work resulted in publications in international conferences and journals. Let us now briefly recall results of each of them.

- **Runtime Enforcement of Timed Properties [PFJ+12].**
  In [PFJ+12] we introduced the first steps to runtime enforcement of (continuous) timed properties. Initially we did not consider all regular timed properties, and focused on safety and co-safety properties described by timed automata. Also the power of enforcement monitors is limited to delaying events, and suppressing events is not allowed. Moreover, here timed words are formalized using delays instead of dates, and we only allow to increase delays.
  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 for 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.
- **Runtime Enforcement of Regular Timed Properties [PFJM14b].**
  The approach in [PFJ+12] targets explicitly safety and co-safety properties. We later investigated whether more expressive properties can be enforced. We generalized the results of [PFJ+12], showed how to synthesize enforcement mechanisms for any regular timed property (modeled with a timed automaton). Indeed, some regular properties may express interesting properties of systems belonging to a larger class that allows to specify some form of transactional behavior. The difficulty that arises is that the enforcement mechanisms should then consider the alternation between currently satisfying and not satisfying the property. In [PFJM14b] we presented a general enforcement monitoring framework for systems with timing requirements. Enforcement mechanisms are described at several levels of abstraction (enforcement function, monitor, and algorithm), thus facilitating the design and implementation of such mechanisms. The focus was on considering more properties, and the power of enforcement mechanism is still limited to delaying events, and suppressing events is not possible as in [PFJ+12].
- **Runtime Enforcement of Timed Properties Revisited [PFJ+14].**
  In a journal paper [PFJ+14] the results of the two papers [PFJ+12, PFJM14b] are combined, and elaborated. More specifically, this paper provided the following additional contributions:
  - a more complete and revised theoretical framework for runtime enforcement of

timed properties has been proposed: we have re-visited the notations, unified and simplified the main definitions;
- a completely new implementation of our enforcement monitors has been proposed that
  i) offers better performance (compared to the ones in [PFJ+12]), ii) is loosely-coupled to UPPAAL;
- enforcement monitors for more properties on longer executions have been synthesized and evaluated;
- correctness proofs of the proposed mechanisms were provided.

– **Runtime Enforcement of Regular Timed Properties by Suppressing and Delaying Events.**
In [PFJ+12, PFJM14b, PFJ+14] enforcement mechanisms receive sequences of events composed of actions and delays between them, and can only increase those delays to satisfy the desired timed property. In a journal paper which has been recently submitted to SCP journal [FJMP14], we consider events composed of actions with absolute occurrence dates, and we allow to increase the dates (while allowing to reduce delays between events). Moreover, suppressing events is also introduced. An event is suppressed if it is not possible to satisfy the property by delaying, whatever are the future continuations of the input sequence (i.e., the underlying TA can only reach non-accepting states from which no accepting state can be reached). Formalizing suppression required us to revisit the formalization of all enforcement mechanisms. The enforcement mechanisms proposed in this paper have the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus allowing the enforcement mechanisms and systems to continue executing.

– **Runtime Enforcement of Parametric Timed Properties with Practical Applications [PFJM14a].**
When we considered requirements from some real application domains such as network security, we noticed that some requirements have constraints both on time and data. Events also carry some data. In [PFJM14a], 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 enforcement monitoring for parametric timed specifications. To handle expressive specifications in our framework, we introduce the model of Parametrized Timed Automata with Variables (PTAVs). To guide us in the choice of expressiveness features we considered requirements in several application domains. 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.

8.2 Future Work

Based on the results presented, there are several other topics related to the area of runtime enforcement of timed properties for further investigation. Few topics are:

- **Enforcement monitoring for systems with limited memory.** An enforcement monitor basically acts as a filter storing the input events in its memory, until it is certain that those events will satisfy the underlying property when released as output. It is certainly interesting to investigate, and to be able to define bounds on memory usage, and to show that the mechanism will effectively work with limited resources. It is also interesting to see whether all or only a subclass of regular timed properties are enforceable with limited resources.
- **Combining enforcement monitors.** For enforcing multiple properties, one possibility is to combine the properties, and to synthesize one enforcement monitor for the resulting property. However, as the number of properties grows, the complexity of the resulting automaton grows. So the time required to traverse the automaton also increases with increase in complexity of the automaton, which is not desirable for runtime enforcement. Thus, it is desirable to investigate whether enforcement monitors can be combined in series, which classes of properties can be combined etc.
- **Predictive runtime enforcement.** We are also investigating on possibilities of some improvements in our framework, when we for example have some knowledge about all the possible input sequences. In case if the monitor has some knowledge about the event emitter, and knows the set of input sequences that it may receive, then the monitor can do even better with respect to deciding when to release events as output. For some events, the monitor can take decision to release them as output earlier without requiring to store them in the memory and waiting to see the future events, when it knows that all the possible continuations of the input it has observed will not violate the property (will only lead to accepting states). Solving the problem will also provide us with a framework to combine enforcement monitors in series.
- **Monitor synthesis Vs Controller synthesis.** Runtime verification (enforcement) monitoring techniques focus on synthesizing monitors from high level specifications. Monitors generally do not influence the execution of the system being monitored, and generally monitors have the power to observe and also control (in case of enforcement) all the events. On the other side, controller synthesis techniques focus on synthesizing a controller from a model of the system and both the system and controller work in a closed loop (the controller influences the system). Moreover, a controller can observe and control only a subset of actions of the system. It is interesting to see how techniques in one area can be useful in the other area and vice versa. For example, also in the domain of runtime monitoring, it is more realistic to consider that the monitor cannot observe and control all the events.
- **Enforcement monitor synthesis mechanism to realize some requirements automatically.** Currently, (verification and enforcement) monitors are

seen as modules outside the system, which take as input a stream of events (output of the system being monitored) and verify or correct this stream according to the property. It will be very interesting to investigate whether these techniques can be used to realize some requirements (such as some requirement related to security). One can imagine enforcement monitors (realizing some requirements) integrated as another layer on top of the core functionality.

- **Enforcement monitoring techniques to guarantee behavior of external components.** If we consider any complex embedded system, the entire system is often not designed and developed from scratch in general. Often parts of the system are bought from another partner company and part of development activities may be outsourced. Thus for some modules, the internal details may be unknown and only the interfaces are known. Enforcement monitors may be used to guarantee that this module communicates well with the other modules. It is interesting to study further about formal theory of interfaces and component based design to show how enforcement mechanisms help to guarantee the behavior of an external component in the system.
- **Applying enforcement monitoring techniques in a particular domain.** It is also very important to work on showing the practical feasibility of applying these techniques in some particular application domains. We already showed how we can formalize requirements related to preventing denial-of-service and automatically synthesize monitors which can be integrated to a system to guarantee such properties. This work can be further continued, focusing on some particular domain such as monitoring network traffic, integrating monitors to a real system to demonstrate practical applicability of these formal based techniques.

All the above proposed research tracks aims to contribute towards runtime monitoring (specifically runtime enforcement) techniques, applying these techniques to develop fault resilient systems, using such mechanisms to enforce requirements related to security and quality-of-service.

Bibliography

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

[AMJM11] Wilkerson L. Andrade, Patricia D. L. Machado, Thierry Jéron, and Herve Marchand. Abstracting time and data for conformance testing of real-time systems. In *2011 IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 9–17. IEEE Computer Society, 2011.

[BB05] Henrik Bohnenkamp and Axel Belinfante. Timed testing with torx. In John Fitzgerald, IanJ. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2005.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[BFH+12] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.

[BJKZ13] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1):3:1–3:26, June 2013.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKZ11] David Basin, Felix Klaedtke, and Eugen Zalinescu. Algorithms for monitoring real-time properties. In Sarfraz Khurshid and Koushik Sen, editors, *Proceedings of the 2nd International Conference on Runtime Verification (RV 2011)*, volume 7186 of *Lecture Notes in Computer Science*, pages 260–275. Springer-Verlag, 2011.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[Bou] Patricia Bouyer. An introduction to timed automata. http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/bouyer-etr05.pdf.

[Bou09] Patricia Bouyer. *From Qualitative to Quantitative Analysis of Timed Systems*. Mémoire d'habilitation, Université Paris 7, Paris, France, January 2009.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Proceedings of the 4th Advanced Course on Petri Nets - Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer Berlin Heidelberg, 2002.

[CMP93] Edward Chang, Zohar Manna, and Amir Pnueli. The safety-progress classification. In FriedrichL. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *NATO ASI Series*, pages 143–202. Springer Berlin Heidelberg, 1993.

[CPS09a] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Proceedings of the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009)*, pages 33–37. IEEE Computer Society, 2009.

[CPS09b] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Safe runtime verification of real-time properties. In Joël Ouaknine and Frits W. Vaandrager, editors, *Proceedings of the 7th International Conference on Formal*

Modeling and Analysis of Timed Systems (FORMATS 2009), volume 5813 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2009.

[CR09] Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In *15th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 246–261, 2009.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99. ACM, 1999.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Softw.*, 19(1):42–51, January 2002.

[Fal10] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Proceedings of the 1st international conference on Runtime verification (RV 2010)*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2010.

[FHTH10] Pascal Fradet and Stéphane Hong Tuan Ha. Aspects of availability. *Sci. Comput. Program.*, 75(7):516–542, July 2010.

[FJMP14] Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Science of Computer Programming (Submitted for review)*, 2014.

[FMFR11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[FZ12] Yliès Falcone and LenoreD. Zuck. Runtime verification: The application perspective. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 284–291. Springer Berlin Heidelberg, 2012.

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153(2):117–133, May 2006.

[HBB+11] Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. Sawja: Static analysis workshop for java. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin Heidelberg, 2011.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, New York, NY, USA, 2004.

[HTJ03] Tim Hunter, Paul Terry, and Alan Judge. Distributed tarpitting: Impeding spam across multiple servers. In *17th Large Installation Systems Administration*, USENIX, San Diego, CA, 2003.

[Jan02] Tretmans Jan. Testing techniques. *Lecture Notes*, 2002. http://www-i2.informatik.rwth-aachen.de/dl/mbt08/lec_notes_04.pdf.

[JJ05] Claude Jard and Thierry Jéron. TGV: Theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, August 2005.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[LBW09] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, January 2009.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[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.

[Met] Metamath. http://us.metamath.org/. Accessed: 2014-11-26.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to timed automata. In *4th Int. conference on Formal Modeling and Analysis of Timed Systems*, LNCS, pages 274–289, 2006.

[NM07] Dejan Nickovic and Oded Maler. AMT: a property-based monitoring tool for analog systems. In Jean-François Raskin and P. S. Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal modeling and analysis of timed systems (FORMATS 2007)*, volume 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer-Verlag, 2007.

[NP10] D. Nickovic and N. Piterman. From MTL to deterministic timed automata. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2010)*, volume 6246 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2010.

[PFJ+12] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. Runtime enforcement of timed properties. In Shaz Qadeer and Serdar Tasiran, editors, *Proceedings of the Third International Conference on Runtime Verification (RV 2012)*, volume 7687 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2012.

[PFJ+14] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014.

[PFJM14a] S. Pinisetty, Y. Falcone, T. Jéron, and H. Marchand. Runtime enforcement of parametric timed properties with practical applications. In *IEEE International Workshop on Discrete Event Systems*, May 2014.

[PFJM14b] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, and Hervé Marchand. Runtime enforcement of regular timed properties. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Proceedings of the ACM Symposium on Applied Computing (SAC-SVT)*, pages 1279–1286. ACM, 2014.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[RBJ00] Vlad Rusu, Lydie Du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In *Proc. Integrated Formal Methods*, pages 338–357. Springer Verlag, 2000.

[Rin03] Martin Rinard. Acceptability-oriented computing. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference On Object-Oriented Programming Sytems, Languages, and Applications Companion (OOPSLA 03 COMPANION)*, pages 221–239. ACM Press, 2003.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[SLS05] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, 0:147–153, 2005.

[TR05] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electronic Notes in Theoretical Computer Science*, 113:145–162, 2005.

[VK04] Mahesh Viswanathan and Moonzoo Kim. Foundations for the run-time monitoring of reactive systems - Fundamentals of the MaC language. In *ICTAC: International Colloquium on Theoretical Aspects of Computing*, Lecture Notes in Computer Science, pages 543–556, 2004.

List of Figures

Résumé

L'enforcement à l'exécution est une technique efficace de vérification et de validation dont le but est de corriger les exécutions incorrectes d'un système, par rapport à un ensemble de propriétés désirées. En utilisant un moniteur d'enforcement, une exécution (possiblement incorrecte), vue comme une séquence d'événements, est passée en entrée du moniteur, puis corrigée en sortie par rapport à la propriété. Durant les dix dernières années, l'enforcement à l'exécution a été étudiée pour des propriétés non temporisées.

Dans cette thèse, nous considérons l'enforcement à l'exécution pour des systèmes où le temps entre les actions du système influence les propriétés à valider. Les exécutions sont donc modélisées par des séquences d'événements composées d'actions avec leurs dates d'occurence (des mots temporisés). Nous considérons l'enforcement à l'exécution pour des spécifications régulières modélisées par des automates temporisés. Les moniteurs d'enforcement peuvent, soit retarder les actions, soit les supprimer lorsque retarder les actions ne permet pas de satisfaire la spécification, permettant ainsi à l'exécution de continuer. Pour faciliter leur conception et la preuve de leur correction, les mécanismes d'enforcement sont modélisés à différents niveaux d'abstraction : les fonctions d'enforcement qui spécifient le comportement attendu des mécanismes en termes d'entrées-sorties, les contraintes qui doivent être satisfaites par ces fonctions, les moniteurs d'enforcement qui décrivent les mécanismes de manière opérationnelle, et les algorithmes d'enforcement qui fournissent une implémentation des moniteurs d'enforcement. La faisabilité de l'enforcement à l'exécution pour des propriétés temporisées est validée en prototypant la synthèse des moniteurs d'enforcement à partir d'automates temporisés. Nous montrons également l'utilité de l'enforcement à l'exécution de spécifications temporisées pour plusieurs domaines d'application.

Mots clés : *vérification et enforcement à l'exécution, propriétés temporisées, automates temporisés, génie logiciel formel.*

Abstract

Runtime enforcement is a verification/validation technique aiming at correcting possibly incorrect executions of a system of interest. It is a powerful technique to ensure that a running system satisfies 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 with a property. Over the last decade, runtime enforcement has been mainly studied in the context of untimed properties.

In this thesis, we consider enforcement monitoring for systems where the physical time elapsing between actions matters. Executions are thus modeled as sequences of events composed of actions with dates (called timed words). We consider runtime enforcement for timed specifications modeled as timed automata, in the general case of regular timed properties. The proposed enforcement mechanism has the power of both delaying events to match timing constraints, and suppressing events when no delaying is appropriate, thus allowing the enforcement mechanisms and systems to continue executing. To ease their design and correctness-proof, enforcement mechanisms are described at several levels: enforcement functions that specify the input-output behavior in terms of transformations of timed words, constraints that should be satisfied by such functions, enforcement monitors that describe the operational behavior of enforcement functions, and enforcement algorithms that describe the implementation of enforcement monitors. The feasibility of enforcement monitoring for timed properties is validated by prototyping the synthesis of enforcement monitors from timed automata. We also show the usefulness of enforcement monitoring of timed specifications for several application-domains.

Keywords: *runtime verification, runtime enforcement, timed properties, timed automata, software engineering.*

Appendix A

Proofs

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

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

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

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

where operators CanD() and $\kappa_\varphi()$ are defined in Section 5.4.1 and Section 5.4.2, respectively.

A.1 Proof of Proposition 5.1 (p. 57)

We shall prove that, given a property $\varphi \subseteq \mathrm{tw}(\Sigma)$, the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$, defined as per Definition 5.8 (p. 55), satisfies the physical constraint, is sound and transparent. These constraints are recalled below:

– **Physical constraint:**

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

– **Soundness:**

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

– **Transparency:**

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

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

We now prove together **(Snd)** and **(Tr)** by an induction on the length of the input timed word σ.

We actually prove a slightly stronger property of E_φ: for any $\sigma \in \mathrm{tw}(\Sigma)$ with $|\sigma| \leq n$, (i) E_φ satisfies $(\mathbf{Snd})_\sigma \stackrel{\text{def}}{=} E_\varphi(\sigma) \models \varphi \vee E_\varphi(\sigma) = \epsilon$ and $(\mathbf{Tr})_\sigma \stackrel{\text{def}}{=} E_\varphi(\sigma) \lhd_d \sigma$, and (ii) $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$, where σ_s and σ_c are as in the definition of store(), recalled above.

Induction Basis ($\sigma = \epsilon$) The proof of the induction basis is immediate from the definitions of E_φ, store(ϵ), $\lhd$, and $\lhd_d$.

Induction Step Let us consider $\sigma' = \sigma \cdot (t, a)$ for some $\sigma \in \mathrm{tw}(\Sigma)$ of length n, $t \in \mathbb{R}_{\geq 0}$, $t \geq \mathrm{end}(\sigma)$, and $a \in \Sigma$. Suppose that $\mathrm{store}(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma'_c = \sigma_c \cdot (t, a)$, where $\mathrm{end}(\sigma_c) \leq t$. We distinguish two cases:

- Case $\kappa_\varphi(\sigma_s, \sigma'_c) \neq \emptyset$. In this case, we have $E_\varphi(\sigma \cdot (t, a)) = \Pi_1\left(\mathrm{store}\left(\sigma \cdot (t, a)\right)\right) = \sigma_s \cdot \min_{\preceq_{\mathrm{lex}}, \mathrm{end}} \kappa_\varphi(\sigma_s, \sigma'_c)$. From the definition of function κ_φ, we have $\kappa_\varphi(\sigma_s, \sigma'_c) \subseteq \sigma_s^{-1} \cdot \varphi$ and thus $E_\varphi(\sigma \cdot (t, a)) \in \varphi$. Thus E_φ satisfies $(\mathbf{Snd})_{\sigma'}$.
  From the induction hypothesis, we know that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$. We deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t, a)) \lhd \Pi_\Sigma(\sigma \cdot (t, a))$ which shows that (ii) holds again for σ'.
  Let $w \in \kappa_\varphi(\sigma_s, \sigma'_c)$. Since $w \in \sigma_s^{-1} \cdot \varphi$, we have $\mathrm{start}(w) \geq \mathrm{end}(\sigma_s)$, which implies that $\sigma_s \cdot w \in \mathrm{tw}(\Sigma)$. Since $w \in \mathrm{CanD}(\sigma'_c)$, we have $\mathrm{start}(w) \geq t$ and $w \succcurlyeq_d \sigma'_c$, which entails that $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma'_c)$. Moreover, from $\mathrm{start}(w) \geq t$, we know that all dates of the events in w have dates greater than or equal to those of the events in $\sigma \cdot (t, a)$. From $\Pi_\Sigma(w) = \Pi_\Sigma(\sigma'_c)$ and $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma'_c) \lhd \Pi_\Sigma(\sigma \cdot (t, a))$, we deduce $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(w) \lhd \Pi_\Sigma(\sigma \cdot (t, a))$. Thus, from this and using $\sigma_s \lhd_d \sigma$, we obtain $\sigma_s \cdot w \lhd_d \sigma \cdot (t, a)$, i.e., E_φ satisfies $(\mathbf{Tr})_{\sigma'}$.
- Case $\kappa_\varphi(\sigma_s, \sigma'_c) = \emptyset$. Note, this case encompasses the two last cases in function store and from the definition of E_φ, in both cases we have $E_\varphi(\sigma \cdot (t, a)) = \Pi_1\big(\mathrm{store}(\sigma \cdot (t, a))\big) = \sigma_s$. Since $E_\varphi(\sigma) = \Pi_1\big(\mathrm{store}(\sigma)\big) = \sigma_s$, and by the induction hypothesis $E_\varphi(\sigma) \models \varphi$, we deduce that E_φ satisfies $(\mathbf{Snd})_{\sigma'}$.
  Moreover $E_\varphi(\sigma \cdot (t, a)) \lhd_d \sigma$ and thus $E_\varphi(\sigma \cdot (t, a)) \lhd_d \sigma \cdot (t, a)$. We deduce $(\mathbf{Tr})_{\sigma'}$.
  Finally, from the induction hypothesis $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c) \lhd \Pi_\Sigma(\sigma)$ we can conclude that $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t, a)) \lhd \Pi_\Sigma(\sigma \cdot (t, a))$, proving (ii) for σ'.

A.2 Proof of Proposition 5.2 (p. 57)

The proof of Proposition 5.2 requires the following lemma related to store which says that, when $\mathrm{store}(\sigma) = (\sigma_s, \sigma_c)$ and σ_c is not the empty timed word, there is no sequences delaying a prefix of σ_c, starting after the ending date of σ, and allowing to correct σ.

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

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

Proof A.1 *The proof is done by induction on the length of* $\sigma \in \mathrm{tw}(\Sigma)$*.*

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

Induction step *Let us consider* $\sigma \cdot (t, a) \in \mathrm{tw}(\Sigma)$ *of length* $n{+}1$*, and let* $\mathrm{store}(\sigma \cdot (t, a)) = (\sigma'_s, \sigma'_c)$*. Following the definition of function* store*, we distinguish three cases:*

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

Let us now return to the proof of Proposition 5.2. We shall prove that, given a property φ, the associated enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ as per Definition 5.8 (p. 55) satisfies the optimality constraint (**Op**) (from Proposition 5.2, p. 57). That is, we shall prove that $\forall \sigma \in \mathrm{tw}(\Sigma) : (\mathbf{Op})_\sigma$, where:

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

We perform an induction on the length of $\sigma \in \mathrm{tw}(\Sigma)$.

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

Induction step Let us consider $\sigma' = \sigma \cdot (t, a)$ for some $\sigma \in \mathrm{tw}(\Sigma)$ of length n, $t \in \mathbb{R}_{\geq 0}$, $t \geq \mathrm{end}(\sigma)$, and $a \in \Sigma$. Let us prove that $(\mathbf{Op})_{\sigma'}$ holds. Suppose $\mathrm{store}(\sigma) = (\sigma_s, \sigma_c)$ and $\sigma'_c = \sigma_c \cdot (t, a)$. We distinguish two cases:

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

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

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

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

We shall prove that

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

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

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

This amounts to prove that:

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

($\Rightarrow$) Since $\Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)) = \Pi_\Sigma(\sigma_c \cdot (t,a))$, by definition of $\succcurlyeq_d$, we have $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a)))$. From transparency, we know that $\sigma_s \lhd_d \sigma$ and $\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_c \cdot (t,a)) \lhd \Pi_\Sigma(\sigma \cdot (t,a))$. Then, from $\text{start}(w') \geq t$, we deduce $m_{\sigma'} \cdot w' \lhd_d \sigma \cdot (t,a)$.
($\Leftarrow$) From $\Pi_\Sigma(w') = \Pi_\Sigma(m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a))$, w' and $m_{\sigma'}^{-1} \cdot E_\varphi(\sigma \cdot (t,a) = m_{\sigma'}^{-1} \cdot \sigma_c \cdot (t,a)$ have the same events. Moreover, since $\text{start}(w') \geq t$, all events in w' have greater dates than t (and hence, greater than those of all events in $\sigma_c \cdot (t,a)$). Thus $w' \succcurlyeq_d \sigma_c \cdot (t,a)$.

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

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

A.3 Preliminaries to the Proof of Proposition 5.3 (p. 69): Characterizing the Configurations of Enforcement Monitors

We first convey some remarks, define some notions and lemmas related to the configurations of enforcement monitors.

A.3.1 Some remarks

Remark A.1 *In the following proofs, without loss of generality, we assume that at any time, in addition to the **idle** rule only one of the rules among the **store** and **dump** rules of the enforcement monitor applies. This simplification does not come at the price of reducing the generality nor the validity of the proofs because i) the **store** and **dump** rules of the enforcement monitor do not rely on the same conditions, and ii) the **store** and **dump** operations of enforcement monitors are assumed to be executed in zero time. The considered simplification however reduces the number of (equivalent) cases in the following proofs.*

Remark A.2 *Between the occurrences of two (input or output) events, the configuration of the enforcement monitor evolves according to the **idle** rule (since it is the rule with lowest priority). Moreover, from any configuration, applying **idle** twice consecutively each delaying for $\delta 1$ and δ_2, or applying **idle** once from the same configuration, with delay $\delta_1+\delta_2$ will result in the same configuration. To simplify notations we will use a rule to simplify the representation of $\mathcal{E}_{\text{ioo}} \in \big((\mathbb{R}_{\geq 0}\times\Sigma)\cup\{\epsilon\}\big)\times Op\times\big((\mathbb{R}_{\geq 0}\times\Sigma)\cup\{\epsilon\}\big)$ stating that*

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

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

A.3.2 Some notations

From Remark A.1, for each time instant, in addition to the ***idle*** rule, only one rule among the ***store*** and ***dump*** rules can be applied. Thus, we have at most two configurations for each time instant. Let us define the functions $\text{config}_{\text{in}}, \text{config}_{\text{out}} : \text{tw}(\Sigma)\times\mathbb{R}_{\geq 0}\to C^{\mathcal{E}}$ that give respectively the first and last configurations of an enforcement monitor at some time instant, reading an input sequence. More formally, given some $\sigma\in\text{tw}(\Sigma), t\in\mathbb{R}_{\geq 0}$:

- $\text{config}_{\text{in}}(\sigma,t) = c^t_\sigma$ such that $c^{\mathcal{E}}_0 \overset{w(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}} c^t_\sigma$ where $w(\sigma,t) \overset{\text{def}}{=} \min_{\preceq}\{w \preceq \mathcal{E}_{\text{ioo}}(\sigma,t) \mid \text{timeop}(w) = t\}$;
- $\text{config}_{\text{out}}(\sigma,t) = c^t_\sigma$ such that $c^{\mathcal{E}}_0 \overset{\mathcal{E}_{\text{ioo}}(\sigma,t)}{\hookrightarrow}{}^*_{\mathcal{E}} c^t_\sigma$.

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

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

Remark A.3 *Value of the third component of configurations. Only the* ***idle*** *rule modifies the value of the third component of configurations: it increments the third component as time elapses.So,* $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \Pi_3(\text{config}_{\text{in}}(\sigma,t)) = \Pi_3(\text{config}_{\text{out}}(\sigma,t)) = t$.

A.3.3 Some intermediate lemmas

Before tackling the proof of Proposition 5.3, we give a list of lemmas that describe the behavior of an enforcement monitor, describing the configurations or the output at some particular time instant for some input and memory content.

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

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

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

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

Proof A.2 (of Lemma A.2) *The proof directly follows from the definitions of the function* $\mathcal{E}$ *associated to an enforcement monitor (see Section 5.5.4, p. 67) which directly depends on* $\mathcal{E}_{\text{ioo}}$*, which is itself monotonic over time (because of the definition of enforcement monitors).*

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

Lemma A.3 (Separation of the output of the EM over time)

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

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

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

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

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

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

Proof A.4 (of Lemma A.4) *The proof of the lemma directly follows from the definitions of $\mathcal{E}_{\mathrm{ioo}}$ (Definition 5.11, p. 68) and* obs *(in Section 7.2). Indeed, using* $\mathrm{obs}(\sigma, t) = \mathrm{obs}(\mathrm{obs}(\sigma, t), t)$*, we deduce that $\mathcal{E}_{\mathrm{ioo}}(\sigma, t) = \mathcal{E}_{\mathrm{ioo}}(\mathrm{obs}(\sigma, t), t)$, for any $\sigma \in \mathrm{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$. Using $\mathcal{E}(\sigma, t) = \Pi_3(\mathcal{E}_{\mathrm{ioo}}(\sigma, t))$, we can deduce the expected result.*

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

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

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

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

Proof A.5 (of Lemma A.5) *The proof is a straightforward induction on the length of σ'_{ms}. It uses the fact that the considered configurations occur at time instants greater than* $\mathrm{end}(\sigma)$*, hence implying that no input event can be read any more. Consequently, following the definition of the enforcement monitor (Definition 5.10, p. 64), on the configurations of the enforcement monitor, only the **idle** and **dump** rules apply. Between* $\mathrm{end}(\sigma'_{\mathrm{ms}})$ *and* $\mathrm{end}(\sigma'_{\mathrm{ms}} \cdot (t, a))$ *where $\sigma'_{\mathrm{ms}} \preceq \sigma'_{\mathrm{ms}} \cdot (t, a) \preceq \sigma_{\mathrm{ms}}$, the configuration of*

the enforcement monitor evolves only using the ***idle*** *rule (no other rule applies) until* $\text{config}_{\text{in}}(\sigma, \text{end}(\sigma'_{\text{ms}} \cdot (t,a))) = (\sigma'^{-1}_{\text{ms}} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q)$*. The* ***dump*** *rule is then applied to get the following derivation* $(\sigma'^{-1}_{\text{ms}} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q) \overset{\epsilon/dump(t,a)/\epsilon}{\hookrightarrow} ((\sigma'_{\text{ms}} \cdot (t,a))^{-1} \cdot \sigma_{\text{ms}}, \sigma_{\text{mc}}, \text{end}(\sigma'_{\text{ms}} \cdot (t,a)), q)$.

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

Lemma A.6 (Output of the EM according to memory content)

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

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

Proof A.6 (of Lemma A.6) *The proof is performed by induction on the length of* σ_{ms} *and uses Lemma A.5.*

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

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

  Using the induction hypothesis, we find $\mathcal{E}(\sigma, t, \text{end}(\sigma')) = \text{obs}(\sigma', \text{end}(\sigma')) = \sigma'$*. Using the semantics of the enforcement monitor (only the* ***dump*** *and* ***idle*** *rules apply, no new event is received), we obtain* $\mathcal{E}(\sigma, \text{end}(\sigma'), t') = \text{obs}((t_l, a), t')$*. Thus,* $\mathcal{E}(\sigma, t, t') = \sigma' \cdot \text{obs}((t_l, a), t') = \text{obs}(\sigma' \cdot (t_l, a), t')$.

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

Lemma A.7 (Content of the internal memory)

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

Proof A.7 (of Lemma A.7) *The proof is performed by induction on the length of* σ.

- *Case* $|\sigma| = 0$. *In this case, from the definition of the enforcement monitor (Definition 5.10, p. 64), none of the store rules can be applied. Consequently, we have* $\Pi_2(\text{config}_{\text{out}}(\sigma, t)) = \epsilon$. *Regarding the enforcement function, as per Definition 5.8, we have* $\Pi_2(\text{store}(\epsilon)) = \epsilon$.
- *Induction case. Let us suppose that the lemma holds for any timed word* σ *of some length* n, *for some* $n \in \mathbb{N}$. *Let us consider* $\sigma' = \sigma \cdot (t_l, a)$ *where* $(t_l, a) \in \mathbb{R}_{\geq 0} \times \Sigma$. *From the induction hypothesis, for any* $t \geq t_l$, *we have* $\Pi_2(\text{store}(\sigma)) = \Pi_2(\text{config}_{\text{out}}(\sigma, t))$. *Let* $\sigma_c = \Pi_2(\text{store}(\sigma))$. *Consequently, we also have* $\text{config}_{\text{in}}(\sigma, t_l) = (\_, \sigma_c, t_l, \_) = \text{config}_{\text{in}}(\sigma \cdot (t_l, a), t_l)$.
  From the definition of store, *we have* $\Pi_2(\text{store}(\sigma \cdot (t_l, a))) = \sigma'_c$, *where* σ'_c *is either* ϵ, $\sigma_c \cdot (t_l, a)$, *or* σ_c *depending on which case of the* store *function applies.*
  Regarding the enforcement monitor, from the update *function (since each case in* store *has a corresponding case in* update*), we also have* $\text{config}_{\text{out}}(\sigma \cdot (t_l, a), t_l) = (\_, \sigma'_c, t_l, \_)$ *(which is obtained by applying one of the store rules based on the value returned by function* update*). For* $t > t_l$, *since none of the store rules can be applied, we can conclude that* $\text{config}_{\text{out}}(\sigma \cdot (t_l, a), t) = (\_, \sigma'_c, t, \_)$.
  Thus, we have $\Pi_2(\text{store}(\sigma \cdot (t_l, a))) = \Pi_2(\text{config}_{\text{out}}(\sigma \cdot (t_l, a), t))$.

A.3.4 Proof of Proposition 5.3: Relation between Enforcement Function and Enforcement Monitor

We shall prove that, given a property φ, the associated enforcement monitor as per Definition 5.10 (p. 64) implements the associated enforcement function $E_\varphi : \text{tw}(\Sigma) \to \text{tw}(\Sigma)$ as per Definition 5.8 (p. 55). That is:

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

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

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

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

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

– Case $t_{n+1} > t$. In this case, $\text{obs}\left(\sigma \cdot (t_{n+1}, a), t\right) = \text{obs}(\sigma, t)$, i.e., at time t, the observations of σ and $\sigma \cdot (t_{n+1}, a)$ are identical.
On the one hand, from the definition of E_φ (since the store and delayed subsequence are defined such that the date of each event in the output is greater than or equal to the date of the corresponding event in the input), we have:

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

On the other hand, regarding the enforcement monitor, since $\text{obs}\left(\sigma \cdot (t_{n+1}, a), t\right) = \text{obs}(\sigma, t)$, using Lemma A.4 (p. 131), we obtain $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}(\sigma, t)$. Using the induction hypothesis, we can conclude that $\text{obs}\left(E_\varphi\left(\sigma \cdot (t_{n+1}, a)\right), t\right) = \mathcal{E}\left(\sigma \cdot (t_{n+1}, a), t\right)$.

– Case $t_{n+1} \leq t$. In this case, we have $\text{obs}(\sigma \cdot (t_{n+1}, a), t) = \sigma \cdot (t_{n+1}, a)$ (i.e., $\sigma \cdot (t_{n+1}, a)$ is observed entirely at time t). Using Remark A.3 (p. 130), we know that the configuration of the enforcement monitor at time $\text{time}(\sigma \cdot (t_{n+1}, a))$ is $\text{config}_{\text{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma)$ for some $\sigma_{\text{ms}}, \sigma_{\text{mc}} \in \text{tw}(\Sigma)$, $q_\sigma \in Q$. Using Lemma A.7, we also have

$$\Pi_2(\text{store}(\sigma)) = \sigma_c = \Pi_2(\text{config}_{\text{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right)) = \sigma_{\text{mc}}.$$

Observe that $\text{config}_{\text{in}}\left(\sigma, t_{n+1}\right) = \text{config}_{\text{in}}\left(\sigma \cdot (t_{n+1}, a), t_{n+1}\right)$ because of i) the definition of $\text{config}_{\text{in}}$ using the definition of $\mathcal{E}_{\text{ioo}}$ and ii) the event (t_{n+1}, a) has not been yet consumed through none of the ***store*** rules by the enforcement monitor at time t_{n+1}.
We distinguish two cases according to whether $\sigma_c \cdot (t_{n+1}, a)$ can be delayed into a word satisfying φ or not, i.e., whether $\kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a)) = \emptyset$, or not.

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

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

where, $\sigma'_{\text{mc}} = \sigma_{\text{mc}}$ if $\text{update}(q_\sigma, \sigma_{\text{mc}}, (t_{n+1}, a)) = (q_\sigma, \sigma_{\text{mc}}, \texttt{bad})$, and $\sigma'_{\text{mc}} = \sigma_{\text{mc}} \cdot (t_{n+1}, a)$ otherwise. Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma)$.
Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, $\text{config}(t)$ is such that only the ***idle*** rule applies.

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

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

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

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

Observe that $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \mathcal{E}(\sigma, t_{n+1} - t_\epsilon)$ because $\text{obs}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon) = \sigma$ according to the definition of obs. Moreover, $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) = \epsilon$ since only the ***idle*** rule applies during the considered time interval. Furthermore, according to Lemma A.6, since $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}}, \sigma'_{\text{mc}}, t_{n+1}, q_\sigma)$, we get $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$. Moreover, we know that $\text{config}_{\text{in}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma)$. Since the enforcement monitor is deterministic, and from Remark A.1 (p. 129), we also get that $\text{config}_{\text{out}}(\sigma, t_{n+1}) = (\sigma_{\text{ms}}, \sigma_{\text{mc}}, t_{n+1}, q_\sigma)$. Using Lemma A.6 (p. 132) again, we get $\mathcal{E}(\sigma, t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}}, t)$.
Consequently we can deduce that $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}(\sigma, t) = \text{obs}(E_\varphi(\sigma), t) = \text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t)$.

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

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

Thus $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q')$.
Let us consider $t_\epsilon \in \mathbb{R}_{\geq 0}$ such that between $t_{n+1} - t_\epsilon$ and t_{n+1}, the enforcement monitor does not read any input nor produce any output, i.e., for all $t \in [t_{n+1} - t_\epsilon, t_{n+1}]$, config$(t)$ is such that only the ***idle*** rule applies.
Let us examine $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t)$. We have:

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

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

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

Observe that $\mathcal{E}(\sigma\cdot(t_{n+1}, a), t_{n+1}-t_\epsilon) = \mathcal{E}(\sigma, t_{n+1}-t_\epsilon)$ because $\text{obs}(\sigma\cdot(t_{n+1}, a), t_{n+1}-t_\epsilon) = \sigma$ according to the definition of obs. Moreover, $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t_{n+1} - t_\epsilon, t_{n+1}) = \epsilon$ since only the ***idle*** rule applies during the considered time interval.
Furthermore, according to Lemma A.6, since $\text{config}_{\text{out}}(\sigma \cdot (t_{n+1}, a), t_{n+1}) = (\sigma_{\text{ms}} \cdot w, \epsilon, t_{n+1}, q')$, we get $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}} \cdot w, t)$.
Now we further distinguish two more sub-cases, based on whether $\text{end}(\sigma_{\text{ms}}\cdot w) > t$ or not (whether all the elements in the memory can be released as output by time t or not).

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

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

    And

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

    One can have $\text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \Pi_1\big(\text{store}(\sigma)\big) \cdot o$, where $o \preccurlyeq \min_{\preceq_{\text{lex}},\text{end}} \kappa_\varphi(\sigma_s, \sigma_c \cdot (t_{n+1}, a))$, which is equal to $\text{obs}(E_\varphi(\sigma), t) \cdot o$, only if the dates computed by the update function are different from the dates computed by E_φ. This would violate the induction hypothesis stating that $\mathcal{E}(\sigma, t) = \text{obs}(E_\varphi(\sigma), t)$. Hence, we have $\text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \text{obs}\left(\Pi_1\left(\text{store}\left(\sigma\right)\right), t\right) = \text{obs}(E_\varphi(\sigma), t)$. Thus, $\text{obs}(E_\varphi(\sigma \cdot (t_{n+1}, a)), t) = \mathcal{E}(\sigma \cdot (t_{n+1}, a), t)$.
  - Case $\text{end}(\sigma_{\text{ms}}) \leq t$. In this case, we can follow the same reasoning as in the previous case to obtain the expected result.
- Case $\text{end}(\sigma_{\text{ms}} \cdot w) \leq t$.
  In this case, similarly following Lemma A.6 (p. 132), we have $\mathcal{E}(\sigma\cdot(t_{n+1}, a), t_{n+1}, t) = \text{obs}(\sigma_{\text{ms}} \cdot w, t) = \sigma_{\text{ms}} \cdot w$. We can also derive that $\mathcal{E}(\sigma, t_{n+1}, t) = \sigma_{\text{ms}}$. Consequently, we have $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t) = \mathcal{E}(\sigma, t) \cdot w$. From the induction hypothesis, we know that $\text{obs}(E_\varphi(\sigma), t) = \mathcal{E}(\sigma, t)$, and we have $\mathcal{E}(\sigma \cdot (t_{n+1}, a), t) = \text{obs}(E_\varphi(\sigma), t) \cdot w$.

Moreover, we have

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

and thus

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

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

Appendix B

List of Publications

B.1 International Journals

- Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014
- Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Science of Computer Programming (Submitted for review)*, 2014

B.2 International Conferences and Workshops

- Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. Runtime enforcement of timed properties. In Shaz Qadeer and Serdar Tasiran, editors, *Proceedings of the Third International Conference on Runtime Verification (RV 2012)*, volume 7687 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2012
- Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, and Hervé Marchand. Runtime enforcement of regular timed properties. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong, editors, *Proceedings of the ACM Symposium on Applied Computing (SAC-SVT)*, pages 1279–1286. ACM, 2014
- S. Pinisetty, Y. Falcone, T. Jéron, and H. Marchand. Runtime enforcement of parametric timed properties with practical applications. In *IEEE International Workshop on Discrete Event Systems*, May 2014

VU :

Le Directeur de Thèse
JÉRON Thierry

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le 23 janvier 2015

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)