



Runtime Enforcement of (Timed) Properties with Uncontrollable Events

Matthieu Renard

► To cite this version:

Matthieu Renard. Runtime Enforcement of (Timed) Properties with Uncontrollable Events. Other [cs.OH]. Université de Bordeaux, 2017. English. NNT : 2017BORD0833 . tel-01684748

HAL Id: tel-01684748

<https://theses.hal.science/tel-01684748>

Submitted on 15 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET
D'INFORMATIQUE

par **Matthieu Renard**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Runtime Enforcement of (Timed) Properties with Uncontrollable Events

Date de soutenance : 11 décembre 2017

Devant la commission d'examen composée de :

Antoine ROLLET ..	Maitre de conférences, Bordeaux INP	Co-directeur
Mohamed MOSBAH	Professeur, Bordeaux INP	Co-directeur
Yliès FALCONE	Maitre de conférences, Université de Grenoble-Alpes	Co-encadrant
Pascale LE GALL ..	Professeur, École Centrale de Paris	Rapporteur
Fatiha ZAÏDI	Maître de conférences HDR, Université Paris Sud ..	Rapporteur
Olivier H. ROUX ..	Professeur, École Centrale de Nantes	Examineur
Jérôme LEROUX ...	Directeur de Recherche, CNRS, LaBRI (Bordeaux) .	Président

Titre Enforcement à l'exécution de propriétés temporisées régulières en présence d'évènements incontrôlables

Résumé Cette thèse étudie l'enforcement de propriétés temporisées à l'exécution en présence d'évènements incontrôlables. Les travaux se placent dans le cadre plus général de la vérification à l'exécution qui vise à surveiller l'exécution d'un système afin de s'assurer qu'elle respecte certaines propriétés. Ces propriétés peuvent être spécifiées à l'aide de formules logiques, ou au moyen d'autres modèles formels, parfois équivalents, comme des automates. Nous nous intéressons à l'enforcement à l'exécution de propriétés spécifiées par des automates temporisés. Tout comme la vérification à l'exécution, l'enforcement à l'exécution surveille l'exécution d'un système, la différence étant qu'un mécanisme d'enforcement réalise certaines modifications sur l'exécution afin de la contraindre à satisfaire la propriété souhaitée. Nous étudions plus particulièrement l'enforcement à l'exécution lorsque certains évènements de l'exécution sont incontrôlables, c'est-à-dire qu'ils ne peuvent pas être modifiés par un mécanisme d'enforcement. Nous définissons des algorithmes de synthèse de mécanismes d'enforcement décrits de manières fonctionnelle puis opérationnelle, à partir de propriétés temporisées régulières (pouvant être représentées par des automates temporisés). Ainsi, deux mécanismes d'enforcement équivalents sont définis, le premier présentant une approche correcte sans considération d'implémentation, alors que le second utilise une approche basée sur la théorie des jeux permettant de précalculer certains comportements, ce qui permet de meilleures performances. Une implémentation utilisant ce précalcul est également présentée et évaluée. Les résultats sont encourageant quant à la faisabilité de l'enforcement à l'exécution en temps réel, avec des temps supplémentaires suffisamment courts sur de petites propriétés pour permettre une utilisation de tels systèmes.

Mots-clés Méthodes formelles, Vérification à l'exécution, Enforcement à l'exécution, Automates, Automates temporisés, Jeux

Title Runtime Enforcement of Timed Properties with Uncontrollable Events

Abstract This thesis studies the runtime enforcement of timed properties when some events are uncontrollable. This work falls in the domain of runtime verification, which includes all the techniques and tools based on or related to the monitoring of system executions with respect to requirement properties. These properties can be specified using different models such as logic formulae or automata. We consider timed regular properties, that can be represented by timed automata. As for runtime verification, a runtime enforcement mechanism watches the executions of a system, but instead of just outputting a

verdict, it modifies the execution so that it satisfies the property. We are interested in runtime enforcement with uncontrollable events. An uncontrollable event is an event that an enforcement mechanism can not modify. We describe the synthesis of enforcement mechanisms, in both a functional and an operational way, that enforce some desired timed regular property. We define two equivalent enforcement mechanisms, the first one being simple, without considering complexity aspects, whereas the second one has a better time complexity thanks to the use of game theory; the latter being better suited for implementation. We also detail a tool that implements the second enforcement mechanism, as well as some performance considerations. The overhead introduced by the use of our tool seems low enough to be used in some real-time application scenarios.

Keywords Formal Methods, Runtime Verification, Runtime Enforcement, Automata, Timed Automata, Games

Laboratoire d'accueil LaBRI, 351 cours de la Libération, Talence 33400, France

Résumé de la thèse

Ce résumé est en grande partie tiré de l'introduction en anglais de cette thèse.

Cette thèse s'intéresse à l'enforcement de propriétés à l'exécution. L'enforcement se situe dans le cadre plus général de la vérification, et plus particulièrement, de la vérification à l'exécution.

Techniques de vérification

La vérification de programmes consiste à améliorer la fiabilité des systèmes informatiques à l'aide de méthodes formelles. Il s'agit principalement de vérifier qu'une propriété, modélisée à l'aide de formules logiques ou d'automates par exemple, est vérifiée par les exécutions d'un système.

On peut distinguer plusieurs types de vérifications : l'analyse statique, le test actif, et la vérification à l'exécution.

L'analyse statique

L'analyse statique regroupe les techniques permettant d'analyser un programme sans l'exécuter. L'analyse statique peut se baser, par exemple, sur la lecture du code source ou du code assembleur du programme.

Parmi les techniques d'analyse statique, se trouvent par exemple l'interprétation abstraite, la conformité de modèles (model-checking) ou encore l'utilisation d'assistants automatiques de preuve lors du développement du programme. L'analyse statique connaît différentes limites : le model-checking, par exemple, peut devenir très coûteux (voire impossible) pour de gros programmes, car il peut s'y produire une explosion combinatoire dans le nombre de configurations considérées.

Le test actif

Le test actif consiste à exécuter un programme sur différentes entrées, en vérifiant à chaque fois si celui-ci est conforme à la propriété souhaitée. Tout comme le model-checking, le test actif peut devenir difficile pour de gros programmes,

dont le nombre d'entrées possible peut devenir très grand. Cependant, même dans ce cas, le test actif reste possible en ne testant pas toutes les entrées, mais seulement un nombre choisi d'entrées, qui peuvent être choisies aléatoirement. Même non-exhaustif, le test actif permet d'améliorer la fiabilité du système : il peut mettre en évidence une faille dans le système, mais ne peut pas (à moins d'être exhaustif) assurer la conformité du système vis à vis de la propriété souhaitée. Néanmoins, augmenter le nombre d'entrées testées augmente la confiance que l'on peut avoir dans la fiabilité du système.

Vérification à l'exécution

La vérification à l'exécution, aussi appelée test passif, est la dernière forme de vérification que nous mentionnerons, et celle dont nous traiterons dans cette thèse. Tout comme le test actif, la vérification à l'exécution exécute le programme, mais cette fois dans les conditions réelles d'utilisation. Le but de la vérification à l'exécution est de vérifier qu'une exécution réelle satisfait bien la propriété visée, soit en mode non-connecté (offline), c'est-à-dire en utilisant un historique d'une exécution, ou en mode connecté (online), au moment même où l'exécution a lieu. Ainsi, la vérification à l'exécution a pour vocation d'être utilisée en temps-réel, afin de prévenir de dysfonctionnements au moment où ils surviennent.

Enforcement à l'exécution

Cette thèse étudie plus particulièrement l'enforcement à l'exécution, qui fait partie de la vérification à l'exécution. L'enforcement à l'exécution consiste, comme la vérification à l'exécution, à s'assurer que l'exécution d'un système satisfait une propriété souhaitée. La principale différence entre l'enforcement et la vérification simple réside dans le fait que l'enforcement modifie l'exécution du système afin qu'elle satisfasse la propriété, plutôt que de prévenir d'une violation de la propriété.

Mécanisme d'enforcement

Un mécanisme d'enforcement est un mécanisme transformant l'exécution d'un système afin qu'elle satisfasse une propriété donnée. Plusieurs fonctionnalités sont donc attendues d'un tel mécanisme. La première fonctionnalité, la correction (soundness), consiste justement à garantir que la sortie du mécanisme d'enforcement satisfait la propriété. Une seconde contrainte, la transparence (transparency), lie l'entrée et la sortie du mécanisme, indiquant ainsi quelles sont les modifications autorisées sur l'exécution. La transparence peut, par exemple, uniquement autoriser à arrêter (tronquer) l'exécution, ou encore à imposer que l'ordre des événements ne soit pas changé, tout en autorisant de

ne pas tous les mettre en sortie. Une troisième fonctionnalité qui peut être attendue d'un mécanisme d'enforcement est l'optimalité, qui impose au mécanisme d'enforcement de modifier le moins possible l'exécution du système. L'optimalité est parfois implicitement exprimée dans la transparence, qui peut indiquer que l'exécution ne doit pas être modifiée si elle satisfait la propriété. Une dernière fonctionnalité, qui n'est parfois qu'implicitement définie, est nécessaire à un mécanisme d'enforcement. Il s'agit des contraintes physiques, indiquant que le mécanisme d'enforcement ne peut pas retirer de sa sortie des événements qui se sont déjà produits. Si cela semble évident en considérant un mécanisme réel, une modélisation formelle serait tout à fait capable de rectifier un événement passé.

Contributions de la thèse

Cette thèse s'intéresse à l'enforcement de propriétés temporisées régulières en présence d'événements incontrôlables.

Formalisme des propriétés

Nous nous intéressons uniquement à l'enforcement de propriétés temporisées régulières, c'est-à-dire de propriétés pouvant être représentées par des automates temporisés, comme décrits dans [Alur and Dill \[1992\]](#). Ces automates temporisés sont des automates classiques auxquels vient s'ajouter un ensemble d'horloges. Les horloges sont simplement des variables dont la valeur évolue de manière linéaire avec l'écoulement du temps. Ces horloges peuvent être utilisées pour former des gardes sur les transitions, qui ne peuvent alors être activées que si ces contraintes sont satisfaites. Une garde peut par exemple demander qu'une certaine horloge ait une valeur inférieure à un certain nombre. Les horloges peuvent également être remises à zéro.

Dans plusieurs chapitres, nous détaillons d'abord des mécanismes d'enforcement pour des propriétés régulières, c'est-à-dire des propriétés représentables par des automates classiques. Il est à noter que les propriétés régulières peuvent être représentées par des automates temporisés sans horloge. Toutefois, le but d'une telle présentation est principalement d'aider à la lecture, l'enforcement de propriétés régulières étant très similaire à l'enforcement de propriétés temporisées, le formalisme en est cependant généralement plus simple.

Événements incontrôlables

Nous considérons certains événements comme étant incontrôlables, autrement dit non modifiables par un mécanisme d'enforcement. Nous choisissons, sans perte de généralité, de considérer que les événements, c'est-à-dire les lettres de

l'alphabet de l'automate, sont soit contrôlables, soit incontrôlables. Un même évènement ne peut pas être contrôlable et devenir incontrôlable ou inversement.

La présence d'évènements incontrôlables empêche l'utilisation de la transparence comme fonctionnalité des mécanismes d'enforcement. En effet, la transparence indique habituellement qu'une exécution qui satisfait la propriété n'est pas modifiée par un mécanisme d'enforcement. Toutefois, en présence d'évènements incontrôlables, l'ordre entre évènements contrôlables et incontrôlables peut changer. C'est pourquoi nous changeons la transparence en conformité (compliance), qui indique que les évènements incontrôlables ne peuvent pas être modifiés et que les évènements contrôlables peuvent être retardés. Nous avons choisi de permettre de retarder les évènements contrôlables car cela a semblé le plus naturel.

Utilisation de la théorie des jeux pour l'enforcement

Nous détaillons comment utiliser la théorie des jeux, et plus spécifiquement des jeux de Büchi, afin de construire des mécanismes d'enforcement pour des propriétés temporisées, en présence d'évènements incontrôlables. L'idée est de construire un graphe de jeux, représentant les actions possible du mécanisme d'enforcement et de son adversaire, l'environnement. Puisqu'un mécanisme d'enforcement est seulement autorisé à retarder les évènements contrôlables, sans en changer l'ordre, il doit d'abord les mémoriser avant de pouvoir les ajouter à sa sortie. Les actions du mécanisme d'enforcement sont donc très limitées : il peut soit émettre le premier évènement de sa mémoire, soit ne rien faire. L'autre joueur dans ce jeu est l'environnement, qui peut ajouter des évènements, contrôlables ou incontrôlables, à l'entrée, et qui peut également faire progresser le temps. Résoudre un jeu de Büchi sur un graphe adéquat de ce genre permet de calculer le comportement d'un mécanisme d'enforcement correct, conforme, et optimal.

Implantation

Nous avons développé un outil, appelé GREP, qui fait office de mécanisme d'enforcement. Cet outil utilise l'approche basée sur la théorie des jeux afin de calculer sa sortie. Étant donnée une propriété temporisée (en utilisant une grammaire personnalisée), GREP construit d'abord un graphe symbolique afin d'abstraire le temps de l'automate temporisée, puis il utilise ce graphe symbolique afin de construire un graphe de jeu tel que décrit précédemment. La stratégie de GREP est ensuite dictée par la résolution du jeu : après avoir calculé les nœuds du graphe qui sont gagnants pour le mécanisme d'enforcement, GREP n'a qu'à émettre des évènements ou ne pas les émettre en s'assurant que le nœud courant dans le graphe de jeu est toujours gagnant. Ainsi, GREP est une implantation d'un mécanisme d'enforcement correct, conforme et optimal.

Plan succinct

Le Chapitre 1 donne un historique des techniques de vérification à l'exécution et de l'enforcement à l'exécution, afin de situer la thèse dans son contexte. Les notations utilisées au long de cette thèse sont détaillées dans le Chapitre 2.

Dans les Chapitres 3 et 4, des mécanismes d'enforcement sont décrits de manière formelle, ainsi que les fonctionnalités attendues de ces mécanismes. Les mécanismes d'enforcement sont décrits de manière déclarative, d'un point de vue fonctionnel, puis de manière opérationnelle, à l'aide d'un système de transitions. Dans ces deux chapitres, les descriptions sont d'abord effectuées pour une propriété régulière, puis dans un second temps pour des propriétés temporisées.

Le Chapitre 5 décrit l'outil GREP, et donne une analyse de ses performances.

Contents

Techniques de vérification	v
L'analyse statique	v
Le test actif	v
Vérification à l'exécution	vi
Enforcement à l'exécution	vi
Mécanisme d'enforcement	vi
Contributions de la thèse	vii
Formalisme des propriétés	vii
Événements incontrôlables	vii
Utilisation de la théorie des jeux pour l'enforcement	viii
Implantation	viii
Plan succinct	ix
Contents	xi
Introduction	1
Static Analysis	1
Active Testing	2
Runtime Verification	2
Runtime Enforcement	3
Contributions of this Thesis	4
Models for Properties	5
Uncontrollable Events	6
Modelling Enforcement Mechanisms	7
Enforcement Primitives	7
Enforcing using Games	8
Implementation	9
Detailed Outline of this Thesis	9
Associated Articles	11
1 State of the Art	13
1.1 Runtime Verification	13
1.2 Runtime Enforcement	14

1.2.1	Enforcing Safety Properties	15
1.2.2	Enforcing more than Safety Properties	16
1.2.3	Enforcing Safety Properties with Uncontrollable Events	17
1.2.4	Enforcing Timed Properties	18
1.2.5	Instrumentation of Enforcement Monitors	18
2	Preliminaries and Notation	21
2.1	Untimed notions	21
2.2	Automata	22
2.3	Timed Languages	22
2.4	Timed Automata	24
2.5	Timed properties	26
2.6	Traces manipulation	26
2.7	Graphs and Büchi games.	27
2.8	Functions	28
3	Enforcing Properties with Uncontrollable Events: A First Approach	29
	Introduction	29
3.1	Enforcing Untimed Properties	29
3.1.1	Enforcement Functions and their Requirements	30
3.1.2	A First Simple Enforcement Function	32
3.1.3	An Optimal Enforcement Function	35
3.1.4	Enforcement Monitors	41
3.2	Enforcing Timed Properties	42
3.2.1	Enforcement Functions and their Properties	43
3.2.2	A Sound, Compliant and Optimal Enforcement Function	46
3.2.3	Enforcement Monitors	53
	Conclusion	57
4	Enforcing Properties using a Büchi Game	59
	Introduction	59
4.1	Notation Changes	59
4.1.1	Timed Words	60
4.1.2	Timed Automata	61
4.1.3	Enforcement Functions	62
4.2	Enforcing Untimed Properties	62
4.2.1	Enforcement Functions and their Requirements	63
4.2.2	Synthesising Enforcement Functions	64
4.2.3	Enforcement Monitors	71
4.3	Enforcing Timed Properties	72
4.3.1	Enforcement Functions and their Properties	73
4.3.2	Synthesising Timed Enforcement Functions	75

4.3.3	Enforcement Monitors	85
	Conclusion	89
5	GREP: Games for Runtime Enforcement of Properties	91
	Introduction	91
5.1	Description of the approach	91
5.2	General Functioning of GREP	92
5.2.1	Symbolic Computing Module (SCM)	92
5.2.2	Enforcement Monitor Module (EMM)	94
5.2.3	Running GREP	97
5.3	Performance Evaluation	99
5.3.1	Comparison with TiPEX	99
5.3.2	Performance Evaluation with Uncontrollable Events	102
	Conclusion	104
	Conclusion	105
	Summary	105
	Future Work	106
A	Proofs	109
A.1	Proofs of Chapter 3	109
A.1.1	Proofs for the Untimed Setting (Section 3.1)	109
A.1.2	Proofs for the Timed Setting (Section 3.2)	118
A.2	Proofs of Chapter 4	134
A.2.1	Proofs for the untimed setting (Section 4.2)	134
A.2.2	Proofs for the timed setting (Section 4.3)	142
	Bibliography	165

Introduction

As electronic devices get more and more powerful and miniaturised, they become more present in all kinds of systems. From coffeepots, kettles, washing machines, to more complicated smartphones or game consoles, devices have increasing computational power. As a comparison, a simple pocket calculator nowadays (as of 2017) has more computational power than all the embedded systems of the Apollo 11 spaceflight, the first one to land humans on the Moon. With this increasing computational power, electronic systems can handle more and more tasks, and can now be used for real-time applications. We can for example use a smartphone to guide us using an internal GPS chip, or play games in virtual realities. A failure in the aforementioned scenarios has a limited impact. However, some systems are *critical*, in the sense that a failure could lead to a human's death or important loss. Planes constitute a good example of critical systems, where a failure in sensors or the piloting system could crash the plane. Some cars also now have self-driving capabilities, thus the autopilot can be considered as a critical system.

These critical systems must be highly reliable, so that humans' deaths are avoided. This is why such pieces of software are well tested, and parts of them are sometimes formally verified. Verification techniques can be categorised according to the moment the verification takes place. Static analysis covers all the techniques that do not require the system to run, such as model checking, abstract interpretation and proof-assisted development. Active testing, often referred to as testing, consists in simulating runs of the system to expose some possible flows. Runtime verification consists in checking that the system acts as expected at runtime, *i.e.* when it is running in a real scenario. These methods are not exclusive, and combining them should lead to even more reliable systems.

Static Analysis

The aim of static analysis is usually to confirm that the software is indeed an implementation of a formal model that has been proved to be safe with respect to some desired behaviours. The main drawback of such analysis is that it gets very difficult to analyse large pieces of software. For example, some

static analysis automatically reads the assembler code of a program and computes a graph representing function calls, with possible values for variables. Such analysis can be expensive because there can be numerous configurations depending on the ability to infer the values of some variables from the code itself. It is also possible to prove some pieces of software with proof assistants, but it usually requires human interaction, like formal specifications of functions.

On top of static analysis, one can run the system and observe its behaviour to determine if it corresponds to the expected one. This is called (active) testing.

Active Testing

In active testing, the system under scrutiny is run with different inputs, and its behaviour is analysed to check whether it is valid with the model or not. Due to the (very) large number of possible inputs, it is usually not possible to analyse every possible run of the system, thus testing can only spot invalid behaviours, but can not ensure that the system is a valid implementation of the considered formal model. Such active testing thus improves the reliability of the system, since every valid runs increases the probability that the system is correct with respect to the model.

Nevertheless, one can not be sure that a run made in real conditions has been tested nor is valid. For this reason, one can also observe the system as it runs in real conditions to check its correctness. This is called passive testing, or *Runtime Verification* (RV).

Runtime Verification

Runtime Verification is usually achieved by the use of a verification monitor, that can be internal or external to the system. An internal monitor is a piece of code appearing in the source code of the system that models a verification monitor. An external monitor is a device that only needs to observe the execution flow of the system under scrutiny to output a verdict stating if it violates the property or not.

One of the interests of such monitors is that they do not require a full specification of the system, since they usually watch a specific simple behaviour. For example, when driving a car in fog, if the car detects another car that is too close for the current velocity, it could alert the driver. Such a property does not need the full specification of the car, but only requires knowledge about the velocity and obstacle detection.

To verify at runtime, it is important to have enough computational power to be able to determine in real-time whether the property is violated or not.

Too little computational power would lead to a big overhead introduced by the verification monitor, that may be unacceptable for real-time applications. In other words, verifying at runtime degrades the performance of the overall system, *i.e.* the system under scrutiny with the verification monitor. Moreover, with timed properties, the overhead of the verification monitor could itself introduce a violation of the property. Limiting this overhead thus is one of the main considerations when dealing with runtime verification. This can be achieved, for instance, by choosing a “simple” formalism for properties.

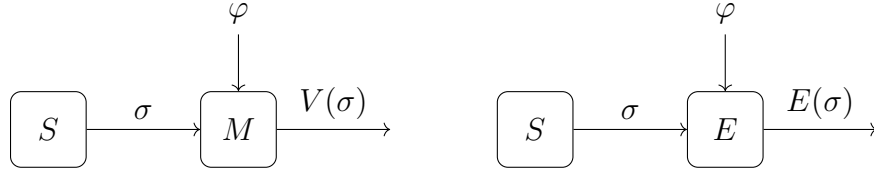
Automata (*i.e.* memoryless transition systems) can decide if a property is satisfied only based on the state reached so far. Thus, a verification monitor verifying a regular property represented by an automaton would only have to follow the path of the execution in the automaton to determine if it is violating the property. Some temporal logics such as LTL are also used to specify properties to be verified at runtime. It is possible for a verification monitor to handle LTL formulae by constructing a Büchi automaton that recognises the same language as the LTL formula and then verify that the execution satisfies the property by following the path in the automaton. Note that Büchi automata, and thus LTL formulae, recognise languages of infinite words, but a verification monitor always considers only finite executions, thus it is not always possible for a verification monitor to output a definitive verdict (see Bauer *et al.* [2006, 2007]).

Runtime Enforcement

Runtime enforcement is similar to runtime verification in its setting: runtime enforcement can be handled by an external monitor or in the source code of the system. As for verification monitors, enforcement monitors are built for a property that can be specified using different formalisms (automata, temporal logics, *etc.*). Where verification monitors output verdicts stating if the current execution of the running system under scrutiny satisfies the desired property, enforcement monitors try to *enforce* the property, *i.e.* modify the execution of the system to have it satisfy the property.

The difference between an enforcement monitor and a verification monitor is illustrated in Fig. 1. In both Figs. 1a and 1b, S is the system generating the sequence of events σ , that is fed to the verification monitor M in Fig. 1a or the enforcement monitor E in Fig. 1b, each one being constructed to verify or enforce the property φ . The difference resides in the output: M outputs a verdict V , indicating whether σ satisfies φ or not, and E outputs a sequence of events $E(\sigma)$, that should satisfy the property.

Runtime enforcement has similarities with control theory (Ramadge and Wonham [1989]; Girault *et al.* [2013]), where one tries to compute events so that the output is valid with respect to some model. There are two possible



(a) General scheme of a verification monitor, outputting a boolean verdict $V(\sigma)$ indicating if σ satisfies φ .
 (b) General scheme of an enforcement monitor, outputting a modified sequence of events $E(\sigma)$ satisfying φ .

Figure 1 – General schemes showing the difference between a verification monitor M and an enforcement monitor E .

differences between runtime enforcement and control theory. The first one is that a controller usually requires the full specification of the system under scrutiny, whereas an enforcement monitor only requires little knowledge about the system. The second one is that the primitives are usually not the same: a controller usually has the ability to add some events to the execution, whereas we consider enforcement monitors as being only able to delay events. Of course, one could consider enforcement monitors that can add events, or controllers delaying events. The two fields have different histories, and may have very few differences, thus the border between them does not seem easy to draw, but the properties verified are not the same. Nevertheless, we do not consider control theory in this thesis, and only address runtime enforcement.

Thus, enforcement monitors should satisfy several requirements. First, enforcement mechanisms should be *sound*, which means that their output (the modified execution) should satisfy the given property. Second, enforcement mechanisms should be *transparent*, meaning that they should not modify an execution that already satisfies the property. Other requirements are sometimes implicitly required, since they stem from physical constraints, stating that the output of an enforcement mechanism should be increasing. In other words, an enforcement mechanism can not remove anything from its output. Since the output of the enforcement mechanism is an execution, removing anything from its output would mean cancel an event, that could have already been handled by some other system, thus this constraint is necessary. We explicitly require enforcement monitors to satisfy this constraint in this thesis. Knowing this, transparency actually gives a notion of *optimality*: the output should be the longest prefix of the input that satisfies the property.

Contributions of this Thesis

In this thesis, we build enforcement mechanisms when some events of the executions are uncontrollable, *i.e.* can not be modified by the enforcement mechanism. We first describe formally such enforcement mechanisms and then

we describe an implementation.

Models for Properties

We build enforcement mechanisms for regular properties, *i.e.* properties that can be modelled as automata. We also build enforcement mechanisms for regular timed properties, *i.e.* properties that can be modelled as timed automata (see [Alur and Dill \[1992\]](#)). Timed automata are automata that have a set of clocks, which are variables that increase linearly with time. Transitions can have guards, that allow them to be followed only when some clock constraints are satisfied. Note that an automaton can be represented as a timed automaton with an empty set of clocks (and thus without any guard on transitions). Nevertheless, we describe enforcement mechanisms for automata before describing them for timed automata, because we think it can ease the reading since the constructions are somewhat similar, but the untimed setting yield better intuition.

We sometimes leverage some classes of automata or timed automata:

1. *Safety* automata are automata representing properties stating that “something bad should never happen”, *i.e.* we start in a good state, and must remain in a good state. As soon as a bad state is reached, the property will not be satisfied. The property stating that “the system is never turned off” is a safety property: as soon as the event “off” occurs, the property is not satisfied and never will be.
2. *Co-safety* properties are properties for which something good must happen in a finite amount of time, and once it has happen, the property is always satisfied. For example, the property “the user must authenticate” is a co-safety property: once the user is authenticated, the property is satisfied.
3. *Response* properties are properties stating that some events need to be followed by some other events. For example, the property “any question must be followed by an answer” is a response property. Over finite words, all properties are response properties ([Falcone et al. \[2012\]](#); [Pinisetty et al. \[2014b\]](#)).

We chose to consider timed automata for several reasons. Timed automata (*i.e.* regular timed properties) are more expressive than untimed automata, but remain a reasonable model to use in a real-time system. When considering real-time systems, it is natural to consider constraints with time, and it is simple to model some properties requiring that some events should happen some time before or after some other events with timed automata. Considering timed automata is also more challenging than considering only untimed ones, mainly

because some problems become undecidable (since the class of timed automata is not closed under complement).

Uncontrollable Events

We consider runtime enforcement with uncontrollable events. Uncontrollable events are events that an enforcement mechanism can not modify, *i.e.* they must be output instantaneously when received. When considering untimed regular properties, this only means that upon receiving an uncontrollable event, the first event output by the enforcement mechanism is the uncontrollable event. In the timed setting, the date of the uncontrollable event must also not be changed, and we allow the enforcement mechanism to output some events at the same date before the uncontrollable event only if the decision of outputting these events was taken prior to the reception of the uncontrollable event.

We chose to consider uncontrollable events because they naturally arise in many concrete scenarios. Uncontrollable events can indeed model some physical events that it is impossible to prevent, but that the system under scrutiny should observe to react correctly. For instance, when driving a car, an uncontrollable event could be that there is an obstacle that appeared before us (this could be another car, on which we have absolutely no control). The system should react to this event, but it can not modify it since it is only an observation of the physical world.

We consider that uncontrollable events are a parameter of the properties, *i.e.* some events of the property are uncontrollable and they always are, the other events are always controllable. Nevertheless, this model allows to change the controllability of some events by duplicating them into two events, one controllable and the other one uncontrollable. Then, changing the controllability of the event only means selecting the good event among the two.

Note that adding uncontrollable events reduces the capabilities of enforcement mechanisms (since uncontrollable events can forbid them to satisfy the property). Moreover, transparency as described previously (*i.e.* stating that an enforcement mechanism should not modify an execution that already satisfies the property) may not be satisfied by an enforcement mechanism when some events are uncontrollable. Enforcement mechanisms can, indeed, change the order between controllable and uncontrollable events in their output, since they can not modify uncontrollable events.

This is why we define the weaker notion of *compliance*, that requires that uncontrollable events are not modified by an enforcement mechanism, and that the order of controllable events is not changed.

Modelling Enforcement Mechanisms

We describe enforcement mechanisms in two different ways. On the one hand, we give a declarative description, representing enforcement mechanisms as functions taking an execution and returning the modified execution, *i.e.* the output of the enforcement mechanism. On the other hand, we give an operational point of view of enforcement mechanisms, representing them as a transition system made of few rules, that has the same output as the functional description. The requirements expressed previously: soundness, compliance, and physical constraints, are expressed as constraints on the function modelling the enforcement mechanism. When modelling enforcement mechanisms for timed properties, we define soundness as stating that the property should eventually be always satisfied, meaning that it can be not satisfied at some point provided that there is a time in the future from which the property always hold afterwards. We decided to allow the property not to hold to give more power to enforcement mechanisms, that could enforce less properties if the property should always be satisfied by the the outputs of enforcement mechanisms. Note that with our timed regular properties, the accepting condition does not depend on the value of the clocks, meaning that “eventually always φ ”, where φ is a timed regular property, is equivalent to “always eventually φ ”. Our definition of soundness is thus equivalent to the one used in [Pinisetty et al. \[2014b\]](#) for instance.

We also define some kind of *optimality* on enforcement mechanisms, that allows comparing two enforcement mechanisms. An optimal enforcement mechanism outputs the maximal number of events it can, with the lowest dates possible when in the timed setting. However, this notion of optimality is not absolute, because an enforcement mechanism can not predict the events it will receive in the future, thus it only has an incomplete knowledge of its input (a prefix up to the current date), and its decisions can only be based on this incomplete knowledge, meaning that an enforcement mechanism that would guess some future that eventually happens could output more events than an optimal one.

Enforcement Primitives

One could consider that the actions of an enforcement mechanism are restricted to two actions: suppressing an action from the input, and adding an action to the output. Nevertheless, with these two actions, an enforcement mechanism could produce any output for a given input (at least satisfying physical constraints). We chose to restrain these actions to constrain the output to satisfy some constraints when compared to the input. These constraints are expressed by compliance. As already stated, uncontrollable events must not be modified by an enforcement mechanism. We choose to only allow enforce-

ment mechanisms to *delay* controllable events, meaning that their order must remain unchanged. It is possible to delay some events indefinitely, but then all the events coming afterwards are also never output. This choice of delaying controllable events naturally leads us to consider enforcement mechanisms that have a *buffer*, *i.e.* controllable events that are stored to be possibly emitted in the future.

Note that we could have authorised enforcement mechanisms to suppress events instead of delaying them, but delaying seemed to us more realistic and more challenging. Suppressing events is simpler than delaying them, since it does not require the computation of many words that can possibly be output. Another possibility would be to allow the enforcement mechanism to output any arbitrary controllable event. However, we think that this does not fit with the philosophy of runtime enforcement, since it looks more like a reimplementation of the system rather than just enforcing it without knowledge of the system.

Changing these primitives should not be really difficult, and lead only to minor changes. The presented framework can be seen as an example of construction of enforcement mechanisms delaying controllable events, that could serve as the basis for other enforcement primitives.

Enforcing using Games

Enforcing with uncontrollable events raises several problems, the main one being that the enforcement mechanism does not have a total control on its output. Thus, an enforcement mechanism has to take into account all the possible uncontrollable events that could happen in the future before outputting anything. This can be done using some game theory. The idea is to consider the enforcement mechanism as a player who can only output things it has stored, and consider that the events received are actions of the other player (the environment). Following this scheme, it is possible to construct a graph representing a game for the enforcement mechanism to solve in order to obtain a winning strategy that builds its output. Chapter 4 presents how exactly the graph is constructed, and how it can be used to compute the output of the enforcement mechanism once the winning strategy is known. The interest of the approach using games as in Chapter 4, compared to the approach of Chapter 3 is that solving the game actually allows us to compute some decisions of the enforcement mechanism prior to the execution, *i.e.* not at runtime. This is used to reduce the time overhead introduced by the enforcement mechanism at runtime in a real implementation.

Implementation

We present a tool called GREP, that is an implementation of the enforcement mechanisms formally defined in the other chapters. More specifically, it uses the computation method formally described in Section 4.3, to enforce a regular property (with or without time). We evaluate GREP and compare it to TiPEX, which is to our knowledge the only other tool that is capable of enforcing timed properties. Note, however, that TiPEX does not consider uncontrollable events, thus we can only compare GREP with TiPEX on properties without any uncontrollable event. We are not aware of any other tool than GREP that would enforce properties (timed or untimed) with uncontrollable events. Even though the results seem satisfying, it is difficult to know how well GREP performs since there is not any similar tool to compare with.

Detailed Outline of this Thesis

We give here a brief description of the different sections of this thesis.

Chapter 1: State of the Art gives a brief history of runtime enforcement, and some related work.

Chapter 2: Preliminaries and Notation details the notation used in this thesis, defining formally all the mathematical tools that are needed to define our enforcement mechanisms.

Chapter 3: Enforcing Properties with Uncontrollable Events: A First Approach presents how we build an enforcement mechanism for a given property. We first define the requirements on enforcement mechanisms as constraints on functions representing enforcement mechanisms. Then, we define a non-optimal enforcement mechanism as a function, to give the basis of how we represent an enforcement mechanism as a function, and then we define an optimal enforcement function. We finally describe a transition system that builds the same output as the optimal enforcement function.

Chapter 4: Enforcing Properties using a Büchi Game revisits Chapter 3, this time using Büchi games to build an optimal enforcement function. We define enforcement functions with a set-theoretic representation, and all their requirements are adapted. Some notation differs from Chapter 3: delays are used instead of dates in timed words (for the timed setting). We again give a functional and operational (with a transition system) description of an optimal enforcement mechanism, that has the same output as the optimal enforcement mechanism defined in Chapter 3. The main difference resides in the

computation of this output: we use a Büchi game to improve the computation time at runtime for an implementation.

Chapter 5: GREP: Games for Runtime Enforcement of Properties gives a presentation of GREP, the tool implementing the enforcement mechanism formally described in Chapter 4. A performance evaluation of GREP is presented, with comparisons with another implementation of an enforcement monitor called TiPEX.

Section 5.3.2: Conclusion summarises the thesis and gives hints about possible improvements and perspectives.

Associated Articles

This thesis presents results that have been published in journals or conference proceedings.

Chapter 3 is taken from the proceedings of the ICTAC 2015 (Renard *et al.* [2015]), that lead to a publication in the MSCS (Mathematical Structures in Computer Science) journal (Renard *et al.* [2017a]).

Parts of Chapter 4 give results presented at the SPIN 2017 conference (Renard *et al.* [2017c]).

The tool described in Chapter 5 has been presented in a paper accepted at the ICTSS (International Conference on Testing Software and Systems) 2017 (Renard *et al.* [2017b]).

Chapter 1

State of the Art

1.1 Runtime Verification

The use of electronic systems to replace humans in many tasks increases with the regular increase of computational power. Since computers can react faster than humans, they now tend to be used even in critical systems (such as systems driving planes), where a failure can be lethal to some people. This has motivated the emergence of software analysis, to try to ensure a maximal reliability for critical systems. *Static analysis* aims at discovering erroneous behaviours by reading the source code (or even assembler code) of a software, trying to detect configurations that are reachable and should not be. Model-checking (McMillan [1993]) and source code proving (Leroy [2006]) are examples of static analysis that can be used to improve the confidence one can have in a piece of software. Every static analysis technique has limitations. For instance, in model-checking, a combinatorial explosion in the number of configurations makes it hard to analyse large pieces of software. Moreover, this kind of analysis can not prevent some electronic errors, for example in a communication between two components. *Runtime verification* (Bauer *et al.* [2007, 2011]; Falcone *et al.* [2013]), also called passive testing (Alcalde *et al.* [2004]; Cavalli *et al.* [2003]) on the other hand, aims at verifying system executions when the system runs. Since the verification happens at runtime, it is possible to detect some transmission errors in a communication, for example, which is not using static analysis. Runtime verification also has some downsides. For example, contrary to static analysis, it is not able to cover all possible executions, but only detects bad behaviours that happen during the execution. Runtime verification can also add some computational overhead when the verification monitor runs at the same time as the system under scrutiny, *i.e.* in *online* mode. Thus, runtime verification and static analysis are complementary methods that both aim at improving the reliance of systems.

Runtime verification usually consists in constructing a monitor that is attached to a system, outputting verdicts indicating whether the execution of the

monitor satisfies a given property or not. In [Falcone et al. \[2013\]](#), the authors give a general description of runtime verification techniques. They consider that runtime verification requires four steps to be achieved:

1. monitor creation from the given property;
2. instrumentation, that consists in attaching the monitor to the system under scrutiny;
3. execution, where the system and the monitor run;
4. responses, where the monitor outputs a verdict and possibly some feedback to the system, after each event.

The first phase requires a property, that can be described using some logic language for instance. From this property, a monitor is created, that is able to output a verdict stating if an execution of the system satisfies the property. Instrumentation of the monitor can be achieved in two different ways. First, the monitor can be inlined in the source code of the system, which requires to have access to this source code. Second, the monitor can be an external device communicating with the system, *i.e.* able to observe its executions.

Verification monitors output verdicts according to the validity of the execution of the system with respect to the desired property. Verdicts can be, for example, a boolean value indicating at every moment whether the execution of the system satisfies the property or not. Verdicts can also be taken from sets of more than two values. In [Bauer et al. \[2006\]](#), for instance, the authors use a three-valued domain $\{\top, \perp, ?\}$ to indicate that the running system will always satisfy the property (\top), will never satisfy it (\perp) from the current state, or that it could satisfy it or not in the future ($?$). Later, in [Bauer et al. \[2007\]](#), they use a four-valued domain $\{\top, \top^p, \perp^p, \perp\}$ that allows the monitor to distinguish the states from which the property is satisfied but may not be satisfied in the future (\top^p) from those from which the property is not satisfied but might be satisfied in the future (\perp^p). Those domains allow the monitor to be switched off whenever a \top or \perp verdict happens, since they ensure that the monitor is no longer required, because the verdict will not change anymore.

1.2 Runtime Enforcement

As runtime verification, runtime enforcement of properties consists in creating a monitor (called an *enforcement monitor* (EM)), but this time its aim is to modify the execution of a running system to ensure it satisfies a given property.

1.2.1 Enforcing Safety Properties

In 2000, Schneider et al. in [Schneider \[2000\]](#) give a model that can enforce some safety properties. The authors start by defining *security policies*. A security policy is a set of authorised executions. Then, a *property* is defined as a security policy for which there exists a predicate that decides if an execution belongs to the policy or not.

Enforcing a security policy consists in restraining the executions of a *target* system to those which belong to the policy. The authors are interested in enforcement mechanisms that watch the execution of the system step by step, and terminate the execution just before the policy is violated. They give a characterisation of the properties that can be enforced using such monitors. Since the system is halted to prevent a bad behaviour, it is pretty clear that only safety properties (stating that something bad should never happen) can be enforced. Indeed, if the desired property is not a safety property, there exists a valid execution that has an invalid prefix, meaning that the enforcement mechanism would halt the system when reading the invalid prefix, thus not enforcing correctly the property (since the good extension would not be considered as a valid execution). As noted by Viswanathan ([Viswanathan \[2000\]](#)), and Schneider himself, all safety properties are not monitorable, but only the ones for which a Turing Machine can decide if finite prefixes of an execution violate the properties.

Then, the authors show that if the property is modelled using what they call a *security automaton*, *i.e.* a safety Büchi automaton, then it is possible to construct an EM that enforces the property. Doing so is pretty straightforward: when the system starts running, a simulation of the security automaton is run in parallel, and every action of the system is fed to the automaton with the adequate event. Whenever the state reached in the automaton with a new action is accepting, then the action is indeed made, but if the state reached is not accepting, then the action is rejected and the system halted.

Security automata as per [Schneider \[2000\]](#) are later called *truncation automata* later in [Ligatti et al. \[2005, 2009\]](#).

Later, in [Bloem et al. \[2015\]](#), Bloem et al. describe *shields*, that are enforcement monitors for reactive systems. According to the authors, the output of a shield should be *correct* and provide *minimum interference*, meaning that it should satisfy the set of given properties, and that it deviates from the input the least possible, respectively. They propose a way to synthesise such monitors to enforce a set of safety properties. Moreover, the shields they propose are *k-stabilising*, meaning that when the output of the shield deviates from the input, then it will not deviate again before k steps, otherwise it will enter a fail-safe mode, where it only ensures correctness, and not minimum interference anymore. A similar approach has been studied by Wu et al. in [Wu et al. \[2016\]](#), where the authors synthesise enforcement monitors for a set of safety

properties, but this time handling burst errors (*i.e.* when errors usually occur in groups). In both of these papers, safety games are used to synthesise the enforcement monitors.

1.2.2 Enforcing more than Safety Properties

In Ligatti *et al.* [2009], the authors propose to enforce some properties that are not safety properties, by giving their monitors the possibility to insert or suppress events from the execution flow of the system. Thus, their monitors act more like filters, suppressing some events when they would lead to a violation of the desired property, and inserting them back when possible. These monitors thus can be seen as firewalls, that block or accept connections according to some specified policies. They also define two crucial properties about enforcement monitors: *soundness* and *transparency*. Soundness states that the output allowed by the monitor always satisfies the property, and transparency requires that an execution of the system that satisfies the property should not be modified by an enforcement monitor. One could note that, for safety properties, Schneider’s truncation automata are sound and transparent. Nevertheless, Ligatti *et al.* show that it is possible to produce sound and transparent enforcement monitors for some properties that are not safety properties. The idea is to store the suffix of an invalid execution that violates the property, until the execution satisfies again the property, in which case the entire stored sequence is output. Such monitors are thus transparent, since the output of the enforcement monitor is indeed the execution of the system if it satisfies the property, and the output of the enforcement monitor always satisfies the property since any invalid sequence is not output entirely. In other words, the output of such monitors is always the longest prefix of the execution that satisfies the execution. Note that if the property is not satisfied by the empty execution, then it might happen that the output of the enforcement monitor does not satisfy the property.

Ligatti *et al.* also provide the set of properties that can be enforced by such monitors: it is the set of *renewal properties*, that are the properties such that any infinite word belonging to the property have infinite number of prefixes that also satisfy the property. They show that some renewal properties are not safety properties, nor liveness, but all safety properties are renewal properties, and some liveness are renewal.

In Falcone *et al.* [2011b], the authors extend the work of Ligatti (Ligatti *et al.* [2009]), considering a different classification of properties, that extends the *Safety-Progress* classification (Manna and Pnueli [1990]; Chang *et al.* [1992]). One advantage of this classification is that each class of properties can be characterised by specific type of finite-state automata. The authors use Streett automata to model properties, adding an accepting condition to also accept some finite words, thus extending the *Safety-Progress* classification to fi-

nite properties. They show that the properties that can be enforced by sound and transparent enforcement monitors are the *response properties* from the *Safety-Progress* classification. These properties are exactly the properties that were called *infinite renewal properties* by Ligatti et al. (Ligatti et al. [2009]).

Then, Falcone et al. define an enforcement monitor as a Mealy machine whose output alphabet is a set of operations. For example, the set of operations can be $\{\text{halt}, \text{store}, \text{dump}, \text{off}\}$, such that *halt* terminates the execution of the system, *off* shuts down the enforcement monitor, *store* stores the event in the monitor's memory, and *dump* releases the events from the monitor's memory. These four operations can be seen as reactions to the four-valued semantics described in Bauer et al. [2007]: when the execution is evaluated as \top , then operation *off* is output; when it evaluates to \perp , *halt* is output; and \top^p and \perp^p evaluations output *dump* and *store*, respectively. The authors also describe how to build an enforcement monitor from a response property given as a Streett automaton.

Others, such as Hamlen et al. [2006] also classified properties to determine which are the ones that can be monitored. In Hamlen et al. [2006], the authors concluded that the set of monitorable properties depends on the capabilities of the monitor. This brought the authors in Fong [2004] to classify properties depending on the *information* that is needed to enforce them. For instance, they study special-cases monitors, that have limited power, using *shallow history automata*, that remember only the set of authorised actions (without ordering). They show that such monitors enforce less than the set of enforceable properties, but that the set of properties it enforces can be useful in some real situations.

In Dolzhenko et al. [2015], the authors model enforcement mechanisms as Mandatory Results Automata (MRA). A system executing actions on an untrusted application see its actions verified and must wait a verified result from the application before executing another action. The authors state that such enforcement mechanisms can enforce more than safety properties.

In Rinard [2003], the author uses enforcement mechanisms to lead software development, specifying several properties for different features that have to be fulfilled by the application. The author describes different kinds of enforcement mechanisms, and determines some possible uses in the context of software development.

1.2.3 Enforcing Safety Properties with Uncontrollable Events

To our knowledge, very little work has been done on the subject of enforcing properties with uncontrollable events.

In Basin et al. [2013], Basin et al. extend the work of Schneider (Schneider [2000]) by considering some events as only *observable* (which corresponds to

what we call uncontrollable events). Their enforcement monitors act exactly as the ones of Schneider, halting the system, but they can do so only if the last event was controllable, *i.e.* it is not possible for the enforcement monitor to halt the system when reading an event that is only observable. The authors determine what are the properties that can be enforced using such monitors. They first define safety properties in a way that takes into account the fact that some events are not controllable. Then, they provide several decidability results, depending on the model used to describe the properties, and describe how to build a monitor when the property is enforceable. In particular, they show that if the property is a finite-state automaton (FSA), and the universe of all possible executions can also be represented by an FSA, then it is possible to decide if the property is enforceable. The authors also note that it is possible to represent some time constraints using observable events, for example by considering events such as ticks of some clock.

1.2.4 Enforcing Timed Properties

Most of the work done so far on the subject of enforcement has been focusing on untimed properties, usually represented by automata. Recent work has extended this work to timed properties.

In [Pinisetty *et al.* \[2013, 2014b\]](#), the authors take interest in the runtime enforcement of timed properties. They propose a way to enforce timed regular properties, *i.e.* properties that can be represented as timed automata, as per [Alur and Dill \[1992\]](#). They model enforcement mechanisms as functions taking a timed word and returning another timed word, *i.e.* modifying an execution, represented as a timed word. They extend the definitions of soundness and transparency to enforcement mechanisms for timed properties, expressing them as requirements on enforcement functions. An enforcement function is *sound* if any non-empty image by this function satisfies the property or will satisfy it in the future, *i.e.* at an infinite time. An enforcement function is *transparent* if it acts as a delayer, *i.e.* the image of a timed word is a timed word whose actions form a prefix of the actions of the argument, and the delays in the image are greater than the ones of the argument. Then, they provide an enforcement function that is sound and transparent, for a given timed regular property. They also define a transition system that has the same output as the enforcement function.

1.2.5 Instrumentation of Enforcement Monitors

In [Martinell and Matteucci \[2007\]](#), the authors model Schneider's and Ligatti's EM ([Schneider \[2000\]](#); [Ligatti *et al.* \[2009\]](#)) using Process-Algebra operators, taking a step towards the implementation of such monitors. As for instrumentation of Verification Monitors (see for example [Falcone *et al.* \[2013\]](#)),

instrumenting Enforcement Monitors can be inlined in the source code, or as an external device. For inlined monitors, one could use for example JavaMOP ([Chen and Rosu \[2005\]](#)), that uses Aspect-Oriented Programming (AOP), *i.e.* AspectJ. An example of use of AspectJ to enforce properties can be found in [Cuppens *et al.* \[2006\]](#). In [Bauer *et al.* \[2009\]](#), Bauer *et al.* define a new language called Polymer that aims at simplifying the specification of the properties for runtime enforcement. They allow to specify complex properties as sets of simpler properties, in a language that can be integrated into Java applications.

The only tool to our knowledge that allows the runtime enforcement of timed properties is TiPEX. TiPEX is a tool written in Python, that acts as a sound and transparent enforcement mechanism for timed regular properties, implementing the approach presented in [Pinisetty *et al.* \[2015a\]](#). TiPEX uses some libraries from UPPAAL ([Larsen *et al.* \[1997\]](#)), in particular to read properties specifications from XML files, and to handle DBMs (Data Bounds Matrices, see [Dill \[1989\]](#)).

Chapter 2

Preliminaries and Notation

In this chapter, we describe the notation used in this document, and give formal definitions of elements we use, such as words, automata, traces, timed words, and timed automata.

2.1 Untimed notions

An *alphabet* is a finite set of symbols. A *word* over an alphabet Σ is a sequence over Σ . The set of finite words over Σ is denoted Σ^* . A *language* over Σ is any subset $L \subseteq \Sigma^*$.

The *length* of a finite word w is noted $|w|$, and the *empty word* is noted ϵ . The concatenation of two words w and w' is noted $w \cdot w'$ (or ww' when clear from the context). A word w' is a *prefix* of a word w , noted $w' \preceq w$, if there exists a word w'' such that $w = w' \cdot w''$. Word w'' is called the *residual* of w after reading the prefix w' , noted $w'' = w'^{-1} \cdot w$. Note that $w' \cdot w'' = w' \cdot w'^{-1} \cdot w = w$. These definitions are extended to languages in the natural way. A language $L \subseteq \Sigma^*$ is *extension-closed* if for any words $w \in L$ and $w' \in \Sigma^*$, $w \cdot w' \in L$. Given a word $w = a_1 \cdot a_2 \dots a_n$ and an integer i such that $1 \leq i \leq n$, we note $w(i)$ the i -th element of w , *i.e.* $w(i) = a_i$. We also note $w_{[..i]}$ the prefix of w of size i : $w_{[..i]} = a_1 \cdot a_2 \dots a_i$.

Given a tuple $e = (e_1, e_2, \dots, e_n)$ of size n , for an integer i such that $1 \leq i \leq n$, we note Π_i the projection on the i -th coordinate, *i.e.* $\Pi_i(e) = e_i$. The tuple (e_1, e_2, \dots, e_n) is sometimes noted $\langle e_1, e_2, \dots, e_n \rangle$ in order to help reading. It can be used, for example, if a tuple contains a tuple. Given a word $w \in \Sigma^*$ and $\Sigma' \subseteq \Sigma$, we define the *restriction* of w to Σ' , noted $w|_{\Sigma'}$, as the word $w' \in \Sigma'^*$ whose letters are the letters of w belonging to Σ' in the same order. Formally, $\epsilon|_{\Sigma'} = \epsilon$ and for any $\sigma \in \Sigma^*$, and any $a \in \Sigma$, $(w \cdot a)|_{\Sigma'} = w|_{\Sigma'} \cdot a$ if $a \in \Sigma'$, or $(w \cdot a)|_{\Sigma'} = w|_{\Sigma'}$ otherwise. We also note $=_{\Sigma'}$ the equality of the restrictions of two words to Σ' : for σ and σ' in Σ^* , $\sigma =_{\Sigma'} \sigma'$ if $\sigma|_{\Sigma'} = \sigma'|_{\Sigma'}$. We define in the same way $\preceq_{\Sigma'}$: $\sigma \preceq_{\Sigma'} \sigma'$ if $\sigma|_{\Sigma'} \preceq \sigma'|_{\Sigma'}$.

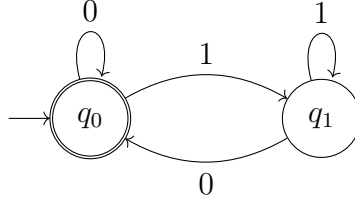


Figure 2.1 – A simple automaton

2.2 Automata

An *automaton* is a tuple $\langle Q, q_0, \Sigma, \rightarrow, F \rangle$, where Q is the finite set of *states*, $q_0 \in Q$ is the initial state, Σ is the alphabet, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of accepting states. Whenever there exists $(q, a, q') \in \rightarrow$, we note it $q \xrightarrow{a} q'$. Relation \rightarrow is extended to $Q \times \Sigma^* \times Q$, such that for $q \in Q$, $\sigma \in \Sigma^*$, $q' \in Q$, and $q'' \in Q$ and $a \in \Sigma$, $q \xrightarrow{\sigma.a} q''$ if $q \xrightarrow{\sigma} q'$ and $q' \xrightarrow{a} q''$. Moreover, for any $q \in Q$, $q \xrightarrow{\epsilon} q$ always holds.

An automaton $\mathcal{A} = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$ is *deterministic* if:

$$\forall q \in Q, \forall a \in \Sigma, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \implies q' = q''.$$

\mathcal{A} is *complete* if:

$$\forall q \in Q, \forall a \in \Sigma, \exists q' \in Q, q \xrightarrow{a} q'.$$

A word w is *accepted* by \mathcal{A} if there exists $q \in F$ such that $q_0 \xrightarrow{w} q$. The language (*i.e.* set of all words) accepted by \mathcal{A} is noted $\mathcal{L}(\mathcal{A})$. A *property* is a language over an alphabet Σ . A regular property is a language accepted by an automaton. In the sequel, we assume that a property φ is represented by a deterministic and complete automaton \mathcal{A}_φ .

Example 2.1. A simple example of automaton is given in Fig. 2.1. In this example, $Q = \{q_0, q_1\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$, and $\rightarrow = \{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 0, q_0), (q_1, 1, q_1)\}$. The accepting states (the ones belonging to F) are the double-circled states, and the initial state (q_0) is represented with an input arrow without any source and an empty label. The language accepted by this automaton is the set of all even numbers, written in binary (the ones ending with a 0 in their binary representation).

2.3 Timed Languages

Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers, and Σ a finite alphabet of actions. An *event* is a pair $(t, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, where t represents the date at which the action a occurs. We define $\text{date}((t, a)) = t$ and $\text{act}((t, a)) = a$ the projections of events on dates and actions respectively. A *timed word* over Σ is a word over $\mathbb{R}_{\geq 0} \times \Sigma$ whose real parts are ascending, *i.e.* σ is a timed word

if $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ and for any $i \in [1; |\sigma| - 1]$, $\text{date}(w(i)) \leq \text{date}(w(i + 1))$. The set of timed words over Σ is denoted $\text{tw}(\Sigma)$. For a timed word $\sigma = (t_1, a_1) \cdot (t_2, a_2) \dots (t_n, a_n)$ and an integer i such that $1 \leq i \leq n$, t_i is the time elapsed before action a_i occurs. We naturally extend the notions of *prefix* and *residual* to timed words.

We denote the total time needed to read a timed word σ by $\text{time}(\sigma)$. Formally, $\text{time}(\epsilon) = 0$, and if $\sigma \neq \epsilon$, $\text{time}(\sigma) = \text{date}(\sigma(|\sigma|))$. The *observation* of σ at time t is the timed word noted $\text{obs}(\sigma, t)$ and defined as:

$$\text{obs}(\sigma, t) = \max_{\preceq}(\{\sigma' \mid \sigma' \preceq \sigma \wedge \text{time}(\sigma') \leq t\}).$$

It corresponds to the word that would be observed at date t when reading σ , if events were received at the date they are associated with. We also define the remainder of the observation of σ at time t as

$$\text{nobs}(\sigma, t) = (\text{obs}(\sigma, t))^{-1} \cdot \sigma,$$

which corresponds to the events that are to be received after date t when reading σ .

The *untimed projection* of a timed word σ is noted $\Pi_{\Sigma}(\sigma)$, and defined as:

$$\Pi_{\Sigma}((t_1, a_1) \cdot (t_2, a_2) \dots (t_n, a_n)) = a_1 \cdot a_2 \dots a_n.$$

It is the sequence of actions of the timed word with dates ignored. For a timed word $\sigma = (t_1, a_1) \cdot (t_2, a_2) \dots (t_n, a_n)$, and a delay $\delta \in \mathbb{R}_{\geq 0}$, σ *delayed by* δ is the word noted $\sigma +_t \delta$ and such that δ is added to all dates:

$$\sigma +_t \delta = (t_1 + \delta, a_1) \cdot (t_2 + \delta, a_2) \dots (t_n + \delta, a_n).$$

Similarly, we define $\sigma -_t \delta$, when $t_1 \geq \delta$, as

$$\sigma -_t \delta = (t_1 - \delta, a_1) \cdot (t_2 - \delta, a_2) \dots (t_n - \delta, a_n).$$

We also extend the definition of the restriction of σ to $\Sigma' \subseteq \Sigma$ to timed words, such that $\epsilon_{|\Sigma'} = \epsilon$, and for $\sigma \in \text{tw}(\Sigma)$ and (t, a) such that $\sigma \cdot (t, a) \in \text{tw}(\Sigma)$, $(\sigma \cdot (t, a))_{|\Sigma'} = \sigma_{|\Sigma'} \cdot (t, a)$ if $a \in \Sigma'$, and $(\sigma \cdot (t, a))_{|\Sigma'} = \sigma_{|\Sigma'}$ otherwise. The notations $=_{\Sigma'}$ and $\preceq_{\Sigma'}$ are then naturally extended to timed words.

A *timed language* is any subset of $\text{tw}(\Sigma)$. The notion of *extension-closed* languages is naturally extended to timed languages, *i.e.* if $L \subseteq \text{tw}(\Sigma)$ is a timed language, L is extension-closed if $L = L \cdot \text{tw}(\Sigma)$. We also extend the notion of extension-closed languages to sets of elements composed of a timed word and a date: a set $S \subseteq \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ is *time-extension-closed* if for any $(\sigma, t) \in S$, for any $w \in \text{tw}(\Sigma)$ such that $\sigma \cdot w \in \text{tw}(\Sigma)$, and for any $t' \geq t$, $(\sigma \cdot w, t') \in S$. In other words, S is time-extension-closed if for every $\sigma \in \text{tw}(\Sigma)$, there exists a date t from which σ and all its extensions are in S , that is, associated with a date greater or equal to t .

Moreover, we define an order on timed words: we say that σ' is a *delayed prefix* of σ , noted $\sigma' \preceq_d \sigma$, whenever $\Pi_\Sigma(\sigma') \preceq \Pi_\Sigma(\sigma)$ and for any $i \in [1; |\sigma'| - 1]$, $\text{date}(\sigma(i)) \leq \text{date}(\sigma'(i))$. Note that the order is not the same in the different constraints: $\Pi_\Sigma(\sigma')$ is a prefix of $\Pi_\Sigma(\sigma)$, but dates in σ' exceed dates in σ . As for the equality $=$ and the prefix order \preceq , we note $\sigma' \preceq_{d\Sigma'} \sigma$ whenever $\sigma'_{|\Sigma'} \preceq_d \sigma_{|\Sigma'}$. We also define a *lexical order* \leq_{lex} on timed words with identical untimed projections, such that $\epsilon \leq_{\text{lex}} \epsilon$, and for two words σ and σ' such that $\Pi_\Sigma(\sigma) = \Pi_\Sigma(\sigma')$, and two events (t, a) and (t', a) such that $(t, a) \cdot \sigma \in \text{tw}(\Sigma)$ and $(t', a) \cdot \sigma' \in \text{tw}(\Sigma)$, $(t', a) \cdot \sigma' \leq_{\text{lex}} (t, a) \cdot \sigma$ if $t' < t \vee (t = t' \wedge \sigma' \leq_{\text{lex}} \sigma)$.

Consider for example the timed word $\sigma = (1, a) \cdot (2, b) \cdot (3, c) \cdot (4, a)$ over the alphabet $\Sigma = \{a, b, c\}$. Then, $\Pi_\Sigma(\sigma) = a.b.c.a$, $\text{obs}(\sigma, 3) = (1, a) \cdot (2, b) \cdot (3, c)$, $\text{nobs}(\sigma, 3) = (4, a)$, and if $\Sigma' = \{b, c\}$, $\sigma_{|\Sigma'} = (2, b) \cdot (3, c)$, and for instance $(1, a) \cdot (2, b) \cdot (4, c) \preceq_d \sigma$, and $\sigma \leq_{\text{lex}} (1, a) \cdot (3, b) \cdot (3, c) \cdot (3, a)$. Moreover, if $w = (1, a) \cdot (2, b)$, then $w^{-1} \cdot \sigma = (3, c) \cdot (4, a)$.

2.4 Timed Automata

Let $X = \{X_1, X_2, \dots, X_n\}$ be a finite set of *clocks*, i.e. variables that increase regularly with time. A *clock valuation* is a function ν from X to $\mathbb{R}_{\geq 0}$. The set of clock valuations for the set of clocks X is noted $\mathcal{V}(X)$, i.e. $\mathcal{V}(X) = \{\nu \mid \nu : X \rightarrow \mathbb{R}_{\geq 0}\}$. We consider the following operations on valuations:

- for any valuation $\nu \in \mathcal{V}(X)$, $\nu + \delta$ is the valuation representing the elapse of δ time units from ν , such that for any $X_i \in X$, $(\nu + \delta)(X_i) = \nu(X_i) + \delta$;
- for any subset $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the valuation representing ν with clocks in X' reset, such that:

$$(\nu[X' \leftarrow 0]) : X_i \mapsto \begin{cases} 0 & \text{if } X_i \in X' \\ \nu(X_i) & \text{otherwise.} \end{cases}$$

$\mathcal{G}(X)$ denotes the set of guards consisting of boolean combinations of constraints of the form $X_i \bowtie c$ with $X_i \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and a valuation ν , we write $\nu \models g$ when for every constraint $X_i \bowtie c$ in g , $\nu(X_i) \bowtie c$ holds.

Definition 2.1 (Timed automaton [Alur and Dill \[1992\]](#)). A *timed automaton* (TA) is a tuple $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$, such that L is a set of locations, $l_0 \in L$ is the initial location, X is a set of clocks, Σ is a finite set of events, $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ is the transition relation, and $G \subseteq L$ is a set of accepting locations. A transition $(l, g, a, X', l') \in \Delta$ is a transition from l to l' , labelled with event a , with guard g , and with the clocks in X' to be reset.

The semantics of a timed automaton \mathcal{A} is a timed transition system $\llbracket \mathcal{A} \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$ where $Q = L \times \mathcal{V}(X)$ is the (infinite) set of states, $q_0 = (l_0, \nu_0)$

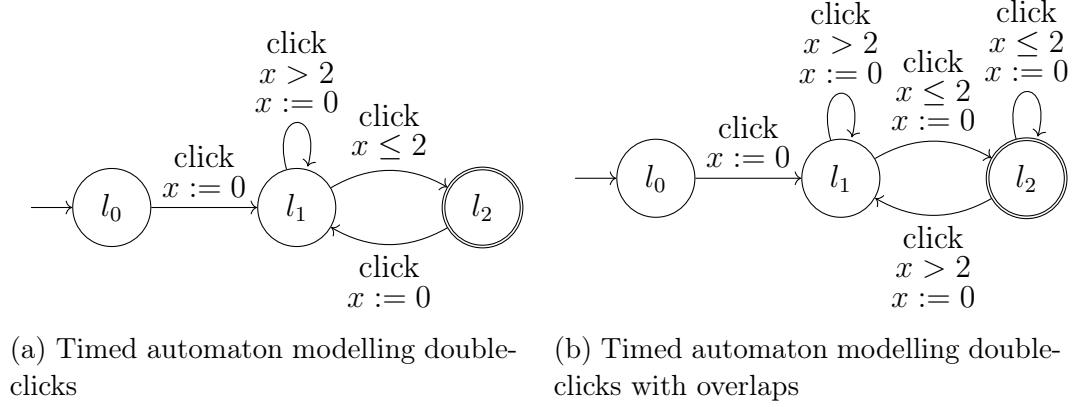


Figure 2.2 – Timed automata modelling double-clicks, without and with overlaps

is the initial state, with $\nu_0 = \nu[X \leftarrow 0]$, $F_G = G \times \mathcal{V}(X)$ is the set of accepting states, $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ is the set of transition labels, each one composed of a delay and an action. The transition relation $\rightarrow \subseteq Q \times \Gamma \times Q$ is a set of transitions of the form $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ with $\nu' = (\nu + \delta)[Y \leftarrow 0]$ whenever there is a transition $(l, g, a, Y, l') \in \Delta$ such that $\nu + \delta \models g$, for $\delta \geq 0$.

A timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ is *deterministic* if for any (l, g_1, a, Y_1, l'_1) and (l, g_2, a, Y_2, l'_2) in Δ , $g_1 \wedge g_2$ is unsatisfiable, meaning that only one transition can be fired at any time. \mathcal{A} is *complete* if for any $l \in L$ and any $a \in \Sigma$, the disjunction of the guards of all the transitions leaving l and labelled by a is valid (i.e., it holds for any clock valuation).

Example 2.2. Examples of timed automata are given in Fig. 2.2. In Fig. 2.2a, $L = \{l_0, l_1, l_2\}$, $X = \{x\}$, $\Sigma = \{\text{click}\}$, $\Delta = \{(l_0, \top, \text{click}, \{x\}, l_1), (l_1, x > 2, \text{click}, \{x\}, l_1), (l_1, x \leq 2, \text{click}, \emptyset, l_2), (l_2, \top, \text{click}, \{x\}, l_1)\}$, and $G = \{l_2\}$, where \top evaluates to true for any clock valuation. This automaton models a double click: considering that the *click* event is a mouse click, the automaton only accepts sequences of clicks that ends with a double-click. The condition for two clicks to be considered as a double-click is that the second one is made less than two time units after the first one. Note that with this modelling, double-clicks can not overlap, *i.e.* clicking three times in less than two time units will not be considered as ending with a double-click, since only the first two clicks will be considered as a double-click. Allowing overlaps would only require splitting the transition from l_2 to l_1 in two, as described in Fig. 2.2b.

A *run* ρ from $q \in Q$ is a valid sequence of transitions in $\llbracket \mathcal{A} \rrbracket$ starting from q , of the form $\rho = q \xrightarrow{(\delta_1, a_1)} q_1 \xrightarrow{(\delta_2, a_2)} q_2 \dots \xrightarrow{(\delta_n, a_n)} q_n$. The set of runs from q_0 is noted $\text{Run}(\mathcal{A})$ and $\text{Run}_{F_G}(\mathcal{A})$ denotes the subset of runs accepted by \mathcal{A} , *i.e.* ending in a state in F_G . The *trace* of the run ρ previously defined is the

timed word $(t_1, a_1).(t_2, a_2) \dots (t_n, a_n)$, with, for $1 \leq i \leq n$, $t_i = \sum_{k=1}^i \delta_k$. Thus, given the trace $\sigma = (t_1, a_1).(t_2, a_2) \dots (t_n, a_n)$ of a run ρ from a state $q \in Q$ to $q' \in Q$, we can define $w = (\delta_1, a_1).(\delta_2, a_2) \dots (\delta_n, a_n)$, with $\delta_1 = t_1$, and $\forall i \in [2; n], \delta_i = t_i - t_{i-1}$, and then $q \xrightarrow{w} q'$. To ease the notation, we will only consider traces and note $q \xrightarrow{\sigma} q'$ whenever $q \xrightarrow{w} q'$ for the previously defined w . Note that to concatenate two traces σ_1 and σ_2 , it is needed to delay σ_2 to obtain a trace: the concatenation σ of σ_1 and σ_2 is the trace defined as $\sigma = \sigma_1.(\sigma_2 +_t \text{time}(\sigma_1))$. In this case, if $q \xrightarrow{\sigma_1} q' \xrightarrow{\sigma_2} q''$, then $q \xrightarrow{\sigma} q''$.

2.5 Timed properties

A *regular timed property* is a timed language $\varphi \subseteq \text{tw}(\Sigma)$ that is accepted by a timed automaton. For a timed word σ , we say that σ *satisfies* φ , noted $\sigma \models \varphi$ whenever $\sigma \in \varphi$. We only consider regular timed properties whose associated automaton is complete and deterministic.

2.6 Traces manipulation

Given a deterministic automaton $\mathcal{A} = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$ and a word $\sigma \in \Sigma^*$, for $q \in Q$, we note q after $\sigma = q'$, where q' is such that $q \xrightarrow{\sigma} q'$, *i.e.* q' is the state reached from q after reading word σ . Since \mathcal{A} is deterministic, there exists only one such q' . We also note $\text{Reach}(\sigma) = q_0$ after σ . We extend these definitions to languages: if L is a language, q after $L = \bigcup_{\sigma \in L} \{q \text{ after } \sigma\}$ and $\text{Reach}(L) = q_0$ after L . For a state $q \in Q$ and an action $a \in \Sigma$, we note $\text{Pred}_a(q) = \{q' \in Q \mid q' \xrightarrow{a} q\}$ the set of predecessors of q by a . This notation is extended to sets of states: if $S \subseteq Q$, then $\text{Pred}_a(S) = \bigcup_{q \in S} \text{Pred}_a(q)$.

In the timed setting, if $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ is a deterministic TA, and $\llbracket \mathcal{A} \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$, we note in the same way as in the untimed setting, with $\sigma \in \text{tw}(\Sigma)$, q after $\sigma = q'$, with $q \xrightarrow{\sigma} q'$, and $\text{Reach}(\sigma) = q_0$ after σ . These operations are also extended to languages as in the untimed setting. We allow the use of the operators after and Reach with an extra parameter, representing an observation time, such that if $t \in \mathbb{R}_{\geq 0}$, then q after $(\sigma, t) = q'$ whenever $q \xrightarrow{\text{obs}(\sigma, t)} q''$, with $q'' = \langle l, \nu \rangle$, and $q' = \langle l, \nu + (t - \text{time}(\text{obs}(\sigma, t))) \rangle$, and $\text{Reach}(\sigma, t) = q_0$ after (σ, t) . The set of predecessors of a state $q \in Q$ by an action $a \in \Sigma$ is $\text{Pred}_a(q) = \{q' \in Q \mid q' \xrightarrow{(0, a)} q\}$, *i.e.* it is the set of states that are predecessors without delay. This definition is also extended to sets of states as in the untimed setting. Moreover, for $q = \langle l, \nu \rangle \in Q$, we note $\text{up}(q) = \{\langle l, \nu + t \rangle \in Q \mid t \in \mathbb{R}_{\geq 0}\}$, it is the set of states that will be reached from q as time elapses if no action occurs. This definition is extended to sets of states: for $S \subseteq Q$, $\text{up}(S) = \bigcup_{q \in S} \text{up}(q)$.

Example 2.3. Consider the property accepting sequences of clicks ending by a double-click, described in Fig. 2.2a. Let us consider that the set Q of states of the semantics of this TA is $Q = L \times \mathbb{R}_{\geq 0}$, with $L = \{l_0, l_1, l_2\}$, and where the valuations are replaced by the value of the unique clock x . Then, for instance, $\text{Reach}((1, \text{click})) = (l_0, 0)$ after $(1, \text{click}) = (l_1, 0)$, and $(l_1, 1)$ after $((1, \text{click}), 3) = (l_2, 4)$, since $x = 2$ when the action *click* occurs, enabling the transition to l_2 , and then 2 time units remain to wait, giving a final value of 4 for x .

2.7 Graphs and Büchi games.

This section presents notation and formalisms that are used in Chapters 4 and 5 only.

A *graph* is a couple $\langle V, E \rangle$ such that V is a set of elements called *vertices*, $E \subseteq V \times V$ is a relation defining *edges* between the vertices. Given a graph $G = \langle V, E \rangle$ and a partition of V into two subsets V_0 and V_1 , it is possible to play a two-player game in the *arena* $A = (V_0, V_1, E)$. A *play* over A is a path in G , i.e. a sequence of vertices such that there exists an edge in G between any two consecutive vertices in the sequence. A *strategy* for player P_0 is a mapping $\sigma : V^*V_0 \rightarrow V$ such that for all $\pi \in V^*$, for all $v_0 \in V_0$, $(v_0, \sigma(\pi.v_0)) \in E$, i.e. the strategy gives a vertex that can be reached from v_0 . Note that V_0 is thus the set of vertices from which P_0 can play, whereas the other player, P_1 , plays from the vertices in V_1 . Strategies for P_1 are defined in a similar way, replacing V_0 by V_1 . A play $\pi = v_0, v_1, \dots$ is *consistent* with the strategy σ if for any $v_i \in V_0$, $v_{i+1} = \sigma(v_0 . v_1 . \dots . v_i)$, meaning that the strategy was followed for any vertex in V_0 . The goal of a game can be, for example, to reach a state in a given subset of V (reachability game), or to ensure that a given subset of V is visited an infinite number of times (Büchi games). Thus, given a subset $F_G \subseteq V$ of vertices, the Büchi game (A, F_G) for P_0 consists in finding a *winning strategy* σ such that all plays π over A consistent with σ visit an infinite number of times the set F_G (i.e. if π is consistent with σ , $\pi \in (V^*F_G)^\omega$). We refer to the nodes in F_G as *Büchi nodes*.

It is known that it is possible to compute the set W_0 of winning vertices for P_0 (i.e. the set of vertices from where there exists a winning strategy for P_0), and the associated winning strategy from all these vertices. From all the other vertices (in $V \setminus W_0$), there exists a winning strategy for P_1 , i.e. $W_1 = V \setminus W_0$, thus P_0 can not win the game if P_1 plays perfectly from one of these vertices. Moreover, it is possible to find a strategy that is *memoryless*, meaning that the only the last vertex is needed to compute the next. Formally, a *memoryless strategy* for P_0 is a strategy $\sigma : V_0 \rightarrow V$. Such strategies are easier to compute, since they do not require to read the entire history before choosing the transition to follow.

2.8 Functions

In all this paper, we use functions to describe the input/output behaviour of enforcement mechanisms. We then use *input* and *output* to refer to “argument” and “image” of such functions, respectively.

Chapter 3

Enforcing Properties with Uncontrollable Events: A First Approach

Introduction

In this chapter, we model enforcement mechanisms for regular properties and for timed regular properties, when some events are uncontrollable. We first model enforcement mechanisms as functions, and express the expected requirements of enforcement mechanisms as constraints that should be satisfied by these functions. The expected requirements are soundness, compliance, and optimality. We describe a function that is not optimal but simple to define, and then we improve it to make it optimal. Then, we give an operational description of the enforcement mechanism, using a transition system whose output is the same as the one of the optimal function. This is first done in Section 3.1 for regular properties, represented by an automaton, and then in Section 3.2 for timed properties, represented by timed automata. The proofs of all the propositions of this chapter are given in appendix A.1.

The work described in this chapter has been published in Renard *et al.* [2015] and Renard *et al.* [2017a].

3.1 Enforcing Untimed Properties

In this section, φ is a regular property defined by an automaton $\mathcal{A}_\varphi = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$ as defined in Section 2.2. Remember the general scheme of an *enforcement mechanism* (EM), given in Fig. 1b.

We consider uncontrollable events in the set $\Sigma_u \subseteq \Sigma$. These events cannot be modified by an EM, *i.e.* they cannot be suppressed nor buffered, so they must be output by the EM whenever they are received. Let us note $\Sigma_c = \Sigma \setminus \Sigma_u$

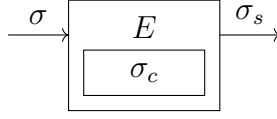


Figure 3.1 – Enforcement monitor E with input σ , output σ_s and buffer σ_c

the set of controllable events, which can be modified by the EM. An EM can decide to buffer them to delay their emission, but it cannot suppress them (nevertheless, it can buffer them endlessly, keeping their order unchanged). Thus, an EM may interleave controllable and uncontrollable events. Since controllable events can be delayed, an EM must store them before outputting them, to keep their order intact. Thus, Fig. 3.1 gives a description of an enforcement monitor E with input σ , output σ_s and buffer, *i.e.* stored controllable events, σ_c .

In this section, for $q \in Q$, we note $\text{uPred}(q) = \bigcup_{u \in \Sigma_u} \text{Pred}_u(q)$, and we extend this definition to sets of states: for $S \subseteq Q$, $\text{uPred}(S) = \bigcup_{q \in S} \text{uPred}(q)$. The operator uPred returns all the states that are predecessors of its argument by an uncontrollable event. In other words, if $q' \in \text{uPred}(q)$, then there exists an uncontrollable event that leads to q from q' . For $S \subseteq Q$, we also note $\overline{S} = Q \setminus S$.

3.1.1 Enforcement Functions and their Requirements

Enforcement Functions, Soundness and Compliance

In this section, we define enforcement functions and give the expected requirements of such functions. An enforcement function is a description of the input/output behaviour of an EM. Formally, we define *enforcement functions* as follows:

Definition 3.1 (Enforcement Function). An *enforcement function* is a function from Σ^* to Σ^* , that is increasing on Σ^* with respect to \preceq :

$$\forall \sigma \in \Sigma^*, \forall \sigma' \in (\Sigma^*), \sigma \preceq \sigma' \implies E(\sigma) \preceq E(\sigma').$$

An enforcement function is a function that modifies an execution, and that cannot remove events it has already output (supposing it is fed with a growing input).

In the sequel, we define the requirements on an EM and express them on enforcement functions. As stated previously, an EM aims at ensuring that executions of a running system satisfy φ , thus its enforcement function has to be *sound*, meaning that its output always satisfies φ .

Definition 3.2 (Soundness). An enforcement function $E : \Sigma^* \rightarrow \Sigma^*$ is *sound* with respect to φ in an extension-closed set $S \subseteq \Sigma^*$ if $\forall \sigma \in S, E(\sigma) \models \varphi$.

Since there are some uncontrollable events that are only observable by the EM, receiving uncontrollable events could lead to the property not being satisfied by the output of the enforcement mechanism. Moreover, some uncontrollable sequences could lead to a state of the property that would be a non-accepting sink state, leading to the enforcement mechanism not being able to satisfy the property any further. Consequently, in Definition 3.2, soundness is not defined for all words in Σ^* , but in a subset S , since it might happen that it is impossible to ensure it from the initial state. Thus for an EM to be effective, S needs to be extension-closed to ensure that the property is always satisfied once it has been. If S were not extension-closed, soundness would only mean that the property is sometimes satisfied. In particular, the identity function would be sound in φ . In practice, there may be an initial period where the enforcement mechanism does not ensure the property, which is unavoidable, but as soon as a safe state is reached, the property becomes enforceable forever, and the property is guaranteed to hold. This approach appears to be the closest to the usual one without uncontrollable events (Pinisetty *et al.* [2014a]).

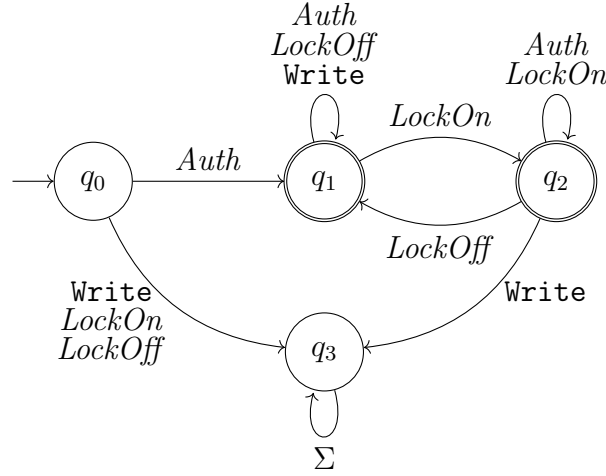
The usual notion of *transparency* (cf. Schneider [2000]; Ligatti *et al.* [2009]) states that the output of an EM is the longest prefix of the input satisfying the property. The name “transparency” stems from the fact that correct executions are left unchanged. Note that transparency also implicitly defines some kind of optimality, since the expected prefix is the longest one. However, because of uncontrollable events, events may be released in a different order from the one they are received. Therefore, transparency can not be ensured, and we define the weaker notion of *compliance*.

Definition 3.3 (Compliance). E is *compliant* with respect to Σ_u and Σ_c , noted $\text{compliant}(E, \Sigma_u, \Sigma_c)$, if

$$\forall \sigma \in \Sigma^*, E(\sigma) \preceq_{\Sigma_c} \sigma \wedge E(\sigma) =_{\Sigma_u} \sigma \wedge \forall u \in \Sigma_u, E(\sigma).u \preceq E(\sigma.u).$$

Intuitively, compliance states that the EM does not change the order of the controllable events and emits uncontrollable events simultaneously with their reception, possibly followed by stored controllable events. We chose to consider enforcement mechanisms that can delay controllable events. To our opinion, it corresponds to the most common choice in practice. However, other primitives, such as deletion or reordering of controllable events could be easily considered. Using other enforcement primitives would require only few changes, especially adapting the definitions of compliance and optimality, and the construction of G (see below). When clear from the context, the partition is not mentioned: E is said to be compliant, and we note it $\text{compliant}(E)$.

We say that a property φ is *enforceable* whenever there exists a compliant function that is sound with respect to φ .


 Figure 3.2 – Property φ_{ex} modelling a shared data storage

Example 3.1. We consider a simple untimed shared storage device. After authentication, a user can write a value only if the storage is unlocked. (Un)locking the device is decided by another entity, meaning that it is not controllable by the user. Property φ_{ex} (see Fig. 3.2) formalises the above requirement.

Property φ_{ex} is not enforceable if the uncontrollable alphabet is $\{\textit{LockOn}, \textit{LockOff}, \textit{Auth}\}$ ¹ since reading the word *LockOn* from q_0 leads to q_3 , which is not an accepting state. However, the existence of such a word does not imply that it is impossible to enforce φ_{ex} for some other input words. If word *Auth* is read, then state q_1 is reached, and from this state, it is possible to enforce φ_{ex} by emitting *Write* events only when in state q_1 . This means that it is possible to have an enforcement function that is sound with respect to φ_{ex} in $\textit{Auth} . \Sigma^*$ (actually in $\textit{Write}^* . \textit{Auth} . \Sigma^*$).

Considering the property φ , it is now possible to define a first enforcement function that is sound with respect to φ and compliant with respect to Σ_u and Σ_c .

3.1.2 A First Simple Enforcement Function

This section shows a simple enforcement function that is compliant and sound. Its main intent is to show the way we define enforcement functions, to provide the skeleton of all the enforcement functions that we define in this document. We define enforcement functions by induction on the argument, so that it can easily be constructed incrementally, which is useful when enforcing in online mode (*i.e.* at runtime).

¹Uncontrollable events are emphasised in italics.

We first define a function that, given a state of the automaton and some controllable events corresponding to the events stored so far by the enforcement mechanism, gives the set of prefixes of this controllable sequence that can be emitted by the enforcement mechanism. The goal is to obtain a sound enforcement mechanism, thus, the prefixes that will be in this set should be prefixes that ensure soundness. We define this function in the following way:

Definition 3.4 (G). For $q \in Q$, and $w \in \Sigma_c^*$, $G(q, w) = \{w' \preceq w \mid q \text{ after } w' \in Q_{\text{enf}}\}$, with $Q_{\text{enf}} = \{q' \in F \mid q \text{ after } \Sigma_u^* \subseteq F\}$.

Now, we can define our enforcement function using this set to ensure soundness. Compliance is ensured in the definition by reacting differently according to the controllability of the events received.

Definition 3.5 (Simple Enforcement Function). Let us define function $\text{store}_\varphi : \Sigma^* \rightarrow \Sigma^* \times \Sigma_c^*$ by induction as follows:

$$\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon),$$

and for $\sigma \in \Sigma^*$ and $a \in \Sigma$, if $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$,

$$\text{store}_\varphi(\sigma . a) = \begin{cases} (\sigma_s . a . \sigma'_s, \sigma'_c) & \text{if } a \in \Sigma_u \\ (\sigma_s . \sigma''_s, \sigma''_c) & \text{if } a \in \Sigma_c, \end{cases}$$

with:

$$\sigma'_s = \max_{\preceq}(G(\text{Reach}(\sigma_s . a), \sigma_c) \cup \{\epsilon\})$$

$$\sigma'_c = \sigma'^{-1}_s . \sigma_c$$

$$\sigma''_s = \max_{\preceq}(G(\text{Reach}(\sigma_s), \sigma_c . a) \cup \{\epsilon\})$$

$$\sigma''_c = \sigma''^{-1}_s . (\sigma_c . a)$$

Then, we can define an enforcement function using store_φ as follows: for $\sigma \in \Sigma^*$, $E_\varphi(\sigma) = \Pi_1(\text{store}_\varphi(\sigma))$.

Function E_φ as per Definition 3.5 is an enforcement function that is compliant with respect to Σ_u and Σ_c . In store_φ , σ_s is the word that is output by E_φ , whereas σ_c is the buffer of stored controllable events that is used to ensure that they are output in the right order. Example 3.2 details the evolution of σ_s and σ_c for a particular input.

Intuitively, E_φ considers states in Q_{enf} (see Definition 3.4) as safe states: they are the states that are in F and from which any uncontrollable word leads to a state in F . This means that as soon as a state in Q_{enf} is reached, delaying all controllable events endlessly produces an output that satisfies the property. Function E_φ does not output any controllable event before its output (thus

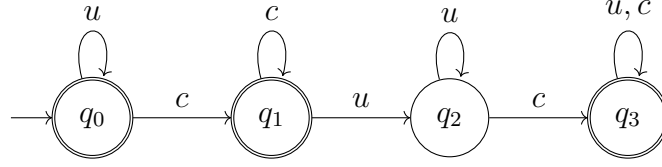

 Figure 3.3 – A property showing that E_φ is not optimal

 Table 3.1 – Evolution of $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$ with $\sigma \preceq c . c . u$

σ	σ_s	σ_c
ϵ	ϵ	ϵ
c	ϵ	c
$c . c$	ϵ	$c . c$
$c . c . u$	$u . c . c$	ϵ

composed only of the uncontrollable events of the input) has reached a state in Q_{enf} , thus it is possible to compute the set of arguments that have an image under E_φ that satisfies φ , and such that all the extensions of the argument also have their image satisfying φ . Formally, this set can be defined as follows:

Definition 3.6 ($\text{Pre}(\varphi)$). $\text{Pre}(\varphi) = \{\sigma \in \Sigma^* \mid G(\text{Reach}(\sigma|_{\Sigma_u}), \sigma|_{\Sigma_c}) \neq \emptyset\} . \Sigma^*$

Note that $\text{Pre}(\varphi)$ as per Definition 3.6 is extension-closed. This set allows us to get the last requirement on E_φ : E_φ is sound with respect to φ in $\text{Pre}(\varphi)$.

Thus, E_φ is an enforcement function that is sound with respect to φ in $\text{Pre}(\varphi)$, and compliant with respect to Σ_u and Σ_c . Proofs of these propositions are straightforward: $\text{Pre}(\varphi)$ is the set of words whose uncontrollable events lead to a state in Q_{enf} (see Definition 3.4), and by construction, from any state in Q_{enf} , any uncontrollable event leads to a state in Q_{enf} . This means that once Q_{enf} is reached, *i.e.* when the input is in $\text{Pre}(\varphi)$, the enforcement mechanism only has to output words that lead to a state in Q_{enf} , which is exactly what G (Definition 3.4) is used for. Thus, E_φ is sound in $\text{Pre}(\varphi)$. Constructing the output by induction ensures compliance, outputting uncontrollable events immediately and adding controllable events to the buffer of stored controllable events (σ_c) before deciding if it is possible to output a prefix of this buffer.

Example 3.2. Now, consider the property defined in Fig. 3.3. Considering that $\Sigma_u = \{u\}$, and $\Sigma_c = \{c\}$, for this property, $Q_{\text{enf}} = \{q_0, q_3\}$. The evolution of σ_s and σ_c as per Definition 3.5 for this property with input $c . c . u$ is given in Table 3.1. This means that $E_\varphi(c . c) = \epsilon$, but one can notice that it could be possible to have a sound and compliant enforcement function E such that $E(c . c) = c$, since $\text{Reach}(c) = q_1 \in F$, and from q_1 it is possible to wait for an uncontrollable event u to reach q_2 and then emit the second stored c event, such that $E(c . c . u) = c . u . c$, thus $\text{Reach}(E(c . c . u)) = q_3 \in Q_{\text{enf}}$.

Thus, we define another requirement on enforcement functions, that states that an enforcement function should output as many events as possible (while ensuring soundness and compliance). This requirement is called *optimality*.

Optimality

We have seen that E_φ as per Definition 3.5 is not optimal, since $E_\varphi(c \cdot c) \prec c$. One can also note that an enforcement function E defined such that for any $\sigma \in \Sigma^*$, $E(\sigma) = \epsilon$ is sound in Σ^* for the property described in Fig. 3.3, and compliant with respect to $\Sigma_u = \{u\}$ and $\Sigma_c = \{c\}$.

An enforcement mechanism should modify the sequence of actions of the system the least possible, thus we require that an enforcement mechanism should be optimal in the sense that its output sequences should be maximal (with respect to \preceq) while preserving soundness and compliance. In the same way we defined soundness in an extension-closed set, we define *optimality* as follows:

Definition 3.7 (Optimality). An enforcement function $E : \Sigma^* \rightarrow \Sigma^*$ that is compliant with respect to Σ_u and Σ_c , and sound in an extension-closed set $S \subseteq \Sigma^*$ is *optimal* in S if:

$$\begin{aligned} \forall E' : \Sigma^* \rightarrow \Sigma^*, \forall \sigma \in S, \forall a \in \Sigma, \\ (\text{compliant}(E') \wedge E'(\sigma) = E(\sigma) \wedge |E'(\sigma.a)| > |E(\sigma.a)|) \implies \\ (\exists \sigma_u \in \Sigma_u^*, E'(\sigma.a.\sigma_u) \not\models \varphi). \end{aligned}$$

Intuitively, optimality states that if there exists a compliant enforcement function that outputs a longer word than an optimal enforcement function, then there must exist a sequence of uncontrollable events that would lead the output of that enforcement function to violate φ . This would imply that this enforcement function is not sound in the same set as the optimal one. Thus, an enforcement function that outputs a longer word than an optimal enforcement function can not be sound and compliant. Since it is not always possible to satisfy the property from the beginning, this condition is restrained to an extension-closed subset of Σ^* , as is for soundness (Definition 3.2).

In the next section, we define an enforcement function that is sound with respect to the property φ , compliant with respect to Σ_u and Σ_c , and optimal.

3.1.3 An Optimal Enforcement Function

In this section, we redefine functions G , store_φ , E_φ , and the set $\text{Pre}(\varphi)$ such that E_φ becomes an enforcement function that is sound with respect to φ in $\text{Pre}(\varphi)$, compliant with respect to Σ_u and Σ_c and optimal in $\text{Pre}(\varphi)$. The idea is still the same: function G gives all the possible words that can be appended to the current output ensuring soundness. Function store_φ helps building the

enforcement function E_φ , and $\text{Pre}(\varphi)$ is the set of arguments for which E_φ ensures soundness.

To be compliant, an enforcement mechanism can buffer the controllable events it has received to emit them later (*i.e.* after having received another event). Thus, the set of states from which an enforcement mechanism can ensure soundness, *i.e.* ensure it can always compute a prefix of the buffer that leads to an accepting state, whatever uncontrollable events are received, depends on its buffer. Thus, to synthesise a sound and compliant enforcement function, one needs to compute the set of words that can be emitted from a certain state with a given buffer, ensuring that an accepting state is always reachable. Thus, to define G , the set of states from which the enforcement mechanism can wait some events knowing an accepting state will always be reachable should be known. Remark that this set has to be a subset of F since it is possible that no event is to be received. This set of states, which depends on the buffer, will be noted S , and is defined in conjunction with another set of states, I , that is used only to compute S . Thus, for a buffer $\sigma \in \Sigma_c^*$, we define the sets of states $I(\sigma)$ and $S(\sigma)$, that represent the states from which the enforcement mechanism can output the first event of σ , and the states in which the enforcement mechanism can wait for another event, respectively.

Definition 3.8 (I , S). Given a sequence of controllable events $\sigma \in \Sigma_c^*$, we define the sets of states of φ , $I(\sigma)$ and $S(\sigma)$ by induction as follows:

$$I(\epsilon) = \emptyset, \quad S(\epsilon) = \{q \in F \mid q \text{ after } \Sigma_u^* \subseteq F\},$$

and, for $\sigma \in \Sigma_c^*$ and $a \in \Sigma_c$,

$$I(a \cdot \sigma) = \text{Pred}_a(S(\sigma) \cup I(\sigma)),$$

$$S(\sigma \cdot a) = S(\sigma) \cup \max_{\subseteq}(\{Y \subseteq F_G \mid Y \cap \text{uPred}(\overline{Y \cup I(\sigma \cdot a)}) = \emptyset\}).$$

Intuitively, $S(\sigma)$ is the set of “winning” states, *i.e.* if an enforcement mechanism has reached a state in $S(\sigma)$ with buffer σ , it will always be able to reach F , whatever events are received afterwards, controllable or uncontrollable. Remember that since there is a possibility of not receiving any other event, $S(\sigma) \subseteq F$, because the EM could end in any of these states, thus this condition is needed to ensure that the output of the EM satisfies the property.

$I(\sigma)$ is the set of intermediate states, the states that can be “crossed” while emitting a prefix of the buffer. The states in $I(\sigma)$ do not need to be in F since no event can be received while the EM is in these states, because it emits all the controllable word it wishes to emit at once.

$S(\sigma \cdot a)$ is defined as the biggest subset of F such that no uncontrollable event leads outside of it or $I(\sigma \cdot a)$, meaning that whatever uncontrollable event is received from a state in $S(\sigma \cdot a)$, the state reached will be either in F (since it will be in $S(\sigma \cdot a)$) or in $I(\sigma \cdot a)$. In both cases, this means that the enforcement

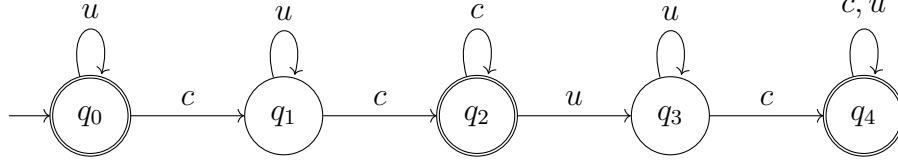


Figure 3.4 – Example property for which $I(c) = \{q_3, q_4\}$ and $S(c) = \{q_0, q_2, q_4\}$.

mechanism can reach an accepting state, whatever uncontrollable events are received.

$I(a.\sigma)$ is defined as the set of all states from which following the transition labelled by a leads either to $I(\sigma)$ or $S(\sigma)$, meaning that the EM can emit the first event of its buffer to be able to reach an accepting state, whatever uncontrollable events are received.

Example 3.3. Consider the property represented on Fig. 3.4, with $\Sigma_u = \{u\}$ and $\Sigma_c = \{c\}$. For this property, $I(\epsilon) = \emptyset$ and $S(\epsilon) = \{q_0, q_4\}$. To calculate this, notice that $F = \{q_0, q_2, q_4\}$, and:

- q_0 after $\Sigma_u^* = \{q_0\} \subseteq F$, thus $q_0 \in S(\epsilon)$.
- q_2 after $\Sigma_u^* = \{q_3\} \not\subseteq F$, thus $q_2 \notin S(\epsilon)$.
- q_4 after $\Sigma_u^* = \{q_4\} \subseteq F$, thus $q_4 \in S(\epsilon)$.

Then, $I(c) = \text{Pred}_c(\{q_0, q_4\}) = \{q_3, q_4\}$. It follows that $\overline{F \cup I(c)} = \{q_1\}$, and $\text{uPred}(\{q_1\}) = \text{Pred}_u(\{q_1\}) = \{q_1\}$. Since $F \cap \{q_1\} = \emptyset$, this means that F satisfies $F \subseteq F$ and $F \cap \text{uPred}(\overline{F \cup I(c)}) = \emptyset$. Thus, $S(c) = F = \{q_0, q_2, q_4\}$.

We can calculate in the same way that $I(c.c) = \{q_1, q_2, q_3, q_4\}$, and $I(c.c.c) = \{q_0, q_1, q_2, q_3, q_4\}$. Since for any $\sigma \in \Sigma_c^*$, $S(\sigma) \subseteq F$, if $c \preceq \sigma$, then $S(\sigma) = F$.

Thus, to output some controllable events while ensuring that the property will be satisfied, an enforcement mechanism must have stored at least three c actions. With three c actions, an enforcement mechanism can output two of them to reach q_2 , from which it must keep one c action in its buffer, to be able to output it if a u event occurs, leading to q_3 .

In other words, the enforcement mechanism is sound as soon as the state reached by its output is in $S(\sigma) \cup I(\sigma)$ with σ its buffer of stored controllable actions. The enforcement mechanism can output the first event of this buffer if the current state is in $I(\sigma)$, otherwise it must not output anything, but wait for other events.

Now, we can use S to define G , the set of words that can be emitted from a state $q \in Q$ by an enforcement mechanism with a buffer $\sigma \in \Sigma_c^*$.

Definition 3.9 (G). For $q \in Q$, $\sigma \in \Sigma_c^*$, $G(q, \sigma) = \{w \in \Sigma_c^* \mid w \preceq \sigma \wedge q \text{ after } w \in S(w^{-1}.\sigma)\}$.

Intuitively, $G(q, \sigma)$ is the set of words that can be output by a compliant enforcement mechanism to ensure soundness from state q with buffer σ . When clear from context, the parameters could be omitted: G is the value of the function for the state reached by the output of an enforcement mechanism with its buffer.

Now, we use G to define store_φ and E_φ , the enforcement function, that is sound, compliant, and optimal.

Definition 3.10 (Functions store_φ , E_φ).² Function $\text{store}_\varphi : \Sigma^* \rightarrow \Sigma^* \times \Sigma_c^*$ is defined as:

$$\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon),$$

and, for $\sigma \in \Sigma^*$ and $a \in \Sigma$, if $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, then:

$$\text{store}_\varphi(\sigma \cdot a) = \begin{cases} (\sigma_s \cdot a \cdot \sigma'_s, \sigma'_c) & \text{if } a \in \Sigma_u \\ (\sigma_s \cdot \sigma''_s, \sigma''_c) & \text{if } a \in \Sigma_c, \end{cases}$$

where, for $q \in Q$ and $w \in \Sigma_c^*$,

$$\kappa_\varphi(q, w) = \max_{\preceq} (G(q, w) \cup \{\epsilon\}),$$

and:

$$\begin{aligned} \sigma'_s &= \kappa_\varphi(\text{Reach}(\sigma_s \cdot a), \sigma_c) & \sigma'_c &= \sigma_s'^{-1} \cdot \sigma_c \\ \sigma''_s &= \kappa_\varphi(\text{Reach}(\sigma_s), \sigma_c \cdot a) & \sigma''_c &= \sigma_s''^{-1} \cdot (\sigma_c \cdot a). \end{aligned}$$

The enforcement function $E_\varphi : \Sigma^* \rightarrow \Sigma^*$ is then defined, for any $\sigma \in \Sigma^*$, as:

$$E_\varphi(\sigma) = \Pi_1(\text{store}_\varphi(\sigma)).$$

Figure 3.1 gives a scheme of the behaviour of the enforcement function. Intuitively, σ_s is the word that can be released as output, whereas σ_c is the buffer containing the events that are already read/received, but cannot be released as output yet because they lead to an unsafe state from which it would be possible to violate the property reading only uncontrollable events. Upon receiving a new event a , the enforcement mechanism distinguishes two cases:

- If a belongs to Σ_u , then it is output, as required by compliance. Then, the longest prefix of σ_c that satisfies φ and leads to a state in S for the associated buffer is also output.
- If a is in Σ_c , then it is added to σ_c , and the longest prefix of this new buffer that satisfies φ and leads to a state in S for the associated buffer is emitted, if it exists.

² E_φ and store_φ depend on Σ_u and Σ_c , but we did not write it in order to lighten the notations.

In both cases, κ_φ is used to compute the longest word that can be output, that is the longest word in G for the state reached so far with the current buffer of the enforcement mechanism, or ϵ if this set is empty. The parameters of κ_φ are those which are passed to G . They correspond to the state reached so far by the output of the enforcement mechanism, and its current buffer, respectively.

As seen in Example 3.1, some properties are not enforceable, but receiving some events may lead to a state from which it is possible to enforce the property. Therefore, it is possible to define a set of words, called $\text{Pre}(\varphi)$, such that E_φ is sound in $\text{Pre}(\varphi)$, as stated in Proposition 3.2:

Definition 3.11 (Pre). The set of input words $\text{Pre}(\varphi) \subseteq \Sigma^*$ is defined as follows:

$$\text{Pre}(\varphi) = \{\sigma \in \Sigma^* \mid G(\text{Reach}(\sigma_{|\Sigma_u}), \sigma_{|\Sigma_c}) \neq \emptyset\} \cdot \Sigma^*.$$

Intuitively, $\text{Pre}(\varphi)$ is the set of words in which E_φ is sound. This set is extension-closed, as required by Definition 3.2. In E_φ , using S ensures that once G is not empty, then it will never be afterwards, whatever events are received. Thus, $\text{Pre}(\varphi)$ is the set of input words such that the output of E_φ would belong to G . Since E_φ outputs only uncontrollable events until G becomes non-empty, the definition of $\text{Pre}(\varphi)$ considers that the state reached is the one that is reached by emitting only the uncontrollable events of σ , and the corresponding buffer would then be the controllable events of σ .

Note that this definition is similar to Definition 3.6, since all the requirements are actually handled by G , which has been redefined.

Example 3.4. Considering property φ_{ex} (Fig. 3.2), with the uncontrollable alphabet $\Sigma_u = \{\text{Auth}, \text{LockOff}, \text{LockOn}\}$, $\text{Pre}(\varphi_{\text{ex}}) = \text{Write}^* \cdot \text{Auth} \cdot \Sigma^*$. Indeed, from the initial state q_0 , if an uncontrollable event, say LockOff , is received, then q_3 is reached, which is a non-accepting sink state, and is thus not in $S(\epsilon)$. In order to reach a state in S (i.e. q_1 or q_2), it is necessary to read Auth . Once Auth is read, q_1 is reached, and from there, all uncontrollable events lead to either q_1 or q_2 . The same holds true from q_2 . Thus, it is possible to stay in the accepting states q_1 and q_2 , by delaying Write events when in q_2 until a LockOff event is received. Consequently, q_1 and q_2 are in $S(\sigma)$ for all $\sigma \in \Sigma_c^*$, and thus $\text{Pre}(\varphi_{\text{ex}}) = \text{Write}^* \cdot \text{Auth} \cdot \Sigma^*$, since Write events can be buffered while in state q_0 until event Auth is received, leading to $q_1 \in S(\text{Write}^*)$.

Properties

E_φ as per Definition 3.10, is an enforcement function that is sound with respect to φ in $\text{Pre}(\varphi)$, compliant with respect to Σ_u and Σ_c , and optimal in $\text{Pre}(\varphi)$, as stated by the following propositions. All the proofs are given in appendix A.1.1.

Proposition 3.1. E_φ as per Definition 3.10 is an enforcement function as per Definition 3.1.

Sketch of proof. We have to show that for all σ and σ' in Σ^* , $E_\varphi(\sigma) \preceq E_\varphi(\sigma.\sigma')$. Following the definition of store_φ , this holds provided that $\sigma' \in \Sigma$ (i.e. σ' is a word of size 1). Since \preceq is an order, it follows that the proposition holds for all $\sigma' \in \Sigma'$.

Proposition 3.2. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$, as per Definition 3.2.

Sketch of proof. We have to show that if $\sigma \in \text{Pre}(\varphi)$, then $E_\varphi(\sigma) \models \varphi$. The proof is made by induction on σ . In the induction step, considering $a \in \Sigma$, we distinguish three different cases:

- $\sigma.a \notin \text{Pre}(\varphi)$. Then the proposition holds.
- $\sigma.a \in \text{Pre}(\varphi)$, but $\sigma \notin \text{Pre}(\varphi)$. Then the input reaches $\text{Pre}(\varphi)$, and since it is extension-closed, all extensions of σ also are in $\text{Pre}(\varphi)$, and we prove that the proposition holds considering the definition of $\text{Pre}(\varphi)$.
- $\sigma \in \text{Pre}(\varphi)$ (and thus, $\sigma.a \in \text{Pre}(\varphi)$ since it is extension-closed). Then, we prove that the proposition holds, based on the definition of store_φ , and more precisely on the definition of S , that ensures that there always exists a compliant output that satisfies φ .

Proposition 3.3. E_φ is compliant, as per Definition 3.3.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. Considering $\sigma \in \Sigma^*$ and $a \in \Sigma$, the proof is straightforward by considering the different values of $\text{store}_\varphi(\sigma.a)$, $(\sigma.a)|_{\Sigma_u}$, and $(\sigma.a)|_{\Sigma_c}$ when $a \in \Sigma_c$ and $a \in \Sigma_u$.

Proposition 3.4. E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 3.7.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. Once $\sigma \in \text{Pre}(\varphi)$, we know that $E_\varphi(\sigma) \models \varphi$ since E_φ is sound in $\text{Pre}(\varphi)$. E_φ is optimal because in store_φ , κ_φ provides the longest possible word. If a longer word were output, then either the output would not satisfy φ , or it would lead to a state that is not in S for the corresponding buffer, meaning that there would exist an uncontrollable word leading to a non-accepting state that would not be in S for the buffer. Then, the enforcement mechanism would have to output some controllable events from the buffer to reach an accepting state, but since the state is not in S , there would exist again an uncontrollable word leading to a non-accepting state that is not in S for the updated buffer. By iterating, the buffer would become ϵ whereas the output of the enforcement mechanism would be leading to a non-accepting state. Therefore, outputting a longer word would mean that the function is not sound. This means that E_φ is optimal in $\text{Pre}(\varphi)$, since it outputs the longest word that allows to be both sound and compliant.

Table 3.2 – Example of the evolution of $(\sigma_s, \sigma_c) = \text{store}_{\varphi_{\text{ex}}}(\sigma)$, with input $\text{Auth} . \text{LockOn} . \text{Write} . \text{LockOff}$

σ	σ_s	σ_c
ϵ	ϵ	ϵ
Auth	Auth	ϵ
$\text{Auth} . \text{LockOn}$	$\text{Auth} . \text{LockOn}$	ϵ
$\text{Auth} . \text{LockOn} . \text{Write}$	$\text{Auth} . \text{LockOn}$	Write
$\text{Auth} . \text{LockOn} . \text{Write} . \text{LockOff}$	$\text{Auth} . \text{LockOn} . \text{LockOff} . \text{Write}$	ϵ

Example 3.5. Consider property φ_{ex} (Fig. 3.2). We illustrate in Table 3.2 the enforcement mechanism by showing the evolution of σ_s and σ_c with input $\sigma = \text{Auth} . \text{LockOn} . \text{Write} . \text{LockOff}$.

3.1.4 Enforcement Monitors

Enforcement monitors are operational descriptions of EMs. We give a representation of an EM for a property φ as an input/output transition system. The input/output behaviour of the enforcement monitor is the same as the one of the enforcement function E_φ as per Definition 3.10. Enforcement monitors are purposed to ease the implementation of EMs.

Definition 3.12 (Enforcement Monitor). An *enforcement monitor* \mathcal{E} for φ is a transition system $\langle C^\mathcal{E}, c_0^\mathcal{E}, \Gamma^\mathcal{E}, \hookrightarrow_\mathcal{E} \rangle$ such that:

- $C^\mathcal{E} = Q \times \Sigma_c^*$ is the set of configurations.
- $c_0^\mathcal{E} = \langle q_0, \epsilon \rangle$ is the initial configuration.
- $\Gamma^\mathcal{E} = \Sigma^* \times \{\text{dump}(\cdot), \text{pass-uncont}(\cdot), \text{store-cont}(\cdot)\} \times \Sigma^*$ is the alphabet, where the first, second, and third members are an input sequence, an enforcement operation, and an output sequence, respectively.
- $\hookrightarrow_\mathcal{E} \subseteq C^\mathcal{E} \times \Gamma^\mathcal{E} \times C^\mathcal{E}$ is the transition relation, defined as the smallest relation obtained by applying the following rules in order (where $w/\bowtie/w'$ stands for $(w, \bowtie, w') \in \Gamma^\mathcal{E}$):

- **Dump:** $\langle q, a.\sigma_c \rangle \xrightarrow{\epsilon/\text{dump}(a)/a}_\mathcal{E} \langle q', \sigma_c \rangle$, if $a \in \Sigma_c$, $G(q, a.\sigma_c) \neq \emptyset$ and $G(q, a.\sigma_c) \neq \{\epsilon\}$, with $q' = q$ after a ,
- **Pass-uncont:** $\langle q, \sigma_c \rangle \xrightarrow{a/\text{pass-uncont}(a)/a}_\mathcal{E} \langle q', \sigma_c \rangle$, with $a \in \Sigma_u$ and $q' = q$ after a ,
- **Store-cont:** $\langle q, \sigma_c \rangle \xrightarrow{a/\text{store-cont}(a)/\epsilon}_\mathcal{E} \langle q, \sigma_c.a \rangle$, with $a \in \Sigma_c$.

In \mathcal{E} , a configuration $c = \langle q, \sigma \rangle$ represents the current state of the enforcement mechanism. The state q is the one reached so far in \mathcal{A}_φ with the output of the monitor. The word of controllable events σ_c represents the buffer of the monitor, i.e. the controllable events of the input that it has not output yet. Rule **dump** outputs the first event of the buffer if it can ensure soundness afterwards (i.e. if there is a non-empty word in G , that must begin with this event). Rule **pass-uncont** releases an uncontrollable event as soon as it is received. Rule **store-cont** simply adds a controllable event at the end of the buffer. Compared to Definition 3.10, the second member of the configuration represents buffer σ_c in the definition of store_φ , whereas σ_s is here represented by state q which is the first member of the configuration, such that $q = \text{Reach}(\sigma_s)$.

Proposition 3.5. *The output of the enforcement monitor \mathcal{E} as per Definition 3.12 for input σ is $E_\varphi(\sigma)$ as per Definition 3.10.*

In Proposition 3.5, the output of the enforcement monitor is the concatenation of all the outputs of the word labelling the path followed when reading σ . A more formal definition is given in the proof of this proposition, in appendix A.1.1.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. We consider the rules applied when receiving a new event. If the event is controllable, then rule `store-cont()` can be applied, possibly followed by rule `dump()` applied several times. If the event is uncontrollable, then rule `pass-uncont()` can be applied, again possibly followed by rule `dump()` applied several times. Since rule `dump()` applies only when there is a non-empty word in G , then this word must begin with the first event of the buffer, and the rule `dump()` can be applied again if there was a word in G of size at least 2, meaning that there is another non-empty word in the new set G , and so on. Thus, the output of all the applications of the rule `dump()` corresponds to the computation of κ_φ in the definition of store_φ , and consequently the outputs of \mathcal{E} and E_φ are the same.

Remark 1. Enforcement monitors as per Definition 3.12 are somewhat similar to the configuration description of EMs in Falcone *et al.* [2011a]. The main difference with the EMs considered in Falcone *et al.* [2011a] is that the rule to be applied depends on the memory (the buffer), whereas in Falcone *et al.* [2011a] it only depends on the state and the event received.

3.2 Enforcing Timed Properties

We extend the framework presented in Section 3.1 to enforce timed properties. EMs and their properties need to be redefined to fit with timed properties. Enforcement functions need an extra parameter representing the date at which

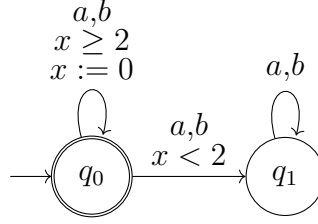


Figure 3.5 – A timed property enforceable only if $\Sigma_u = \emptyset$.

the output is observed. Soundness needs to be weakened so that, at any time instant, the property is allowed not to hold, provided that it will hold in the future.

Considering uncontrollable events with timed properties raises several difficulties. First, as in the untimed case, the order of events might be modified. Thus, previous definitions of transparency [Pinisetty *et al.* \[2012\]](#), stating that the output of an enforcement function will eventually be a delayed prefix of the input, can not be used in this situation. Moreover, when delaying some events to have the property satisfied in the future, one must consider the fact that some uncontrollable events could occur at any moment (and cannot be delayed). Finally, some properties become not enforceable because of uncontrollable events, meaning that for these properties it is impossible to obtain sound EMs, as shown in [Example 3.6](#).

In this section, φ is a timed property defined by a timed automaton $\mathcal{A}_\varphi = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ with semantics $\llbracket \mathcal{A}_\varphi \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$. As in the untimed setting, for $q \in Q$, we define $\text{uPred}(q) = \bigcup_{u \in \Sigma_u} \text{Pred}_u(q)$, and for $S \subseteq Q$, $\text{uPred}(S) = \bigcup_{q \in S} \text{uPred}(q)$ and $\bar{S} = Q \setminus S$.

Example 3.6 (Non-Enforceable Property). Consider the property defined by the automaton in [Fig. 3.5](#) with alphabet $\{a, b\}$, that requires that there is always at least two time units between two consecutive events.

If all actions are controllable ($\Sigma_u = \emptyset$), the property is enforceable because an EM just needs to delay events until clock x exceeds 2. Otherwise, the property is not enforceable. For instance, if $\Sigma_u = \{a\}$, word $(1, a)$ cannot be corrected by a compliant enforcement mechanism.

3.2.1 Enforcement Functions and their Properties

In this section, we define enforcement functions and the requirements expected to model enforcement mechanisms, as in [Section 3.1](#), but in a timed setting (*i.e.* the property now is a TA).

An enforcement function takes a timed word and the current time as input, and outputs a timed word:

Definition 3.13 (Enforcement Function). Given an alphabet of actions Σ , an *enforcement function* is a function $E : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma)$ that satisfies the following constraints:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t, E(\sigma, t) \preceq E(\sigma, t')$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall (t, a) \in \mathbb{R}_{\geq 0} \times \Sigma,$
 $\sigma \cdot (t, a) \in \text{tw}(\Sigma) \implies E(\sigma, t) \preceq E(\sigma \cdot (t, a), t).$

Definition 3.13 models physical constraints: an enforcement function can not remove something it has already output. The first condition requires that, as time elapses, the enforcement function can only add new events to its output. The second condition states that, when receiving a new event in the input, the enforcement function, again, can only add new events to its output. In both cases, the new output must be an extension of what has been output so far.

As in the untimed setting (see Definition 3.2), *soundness* requires that the property is satisfied by the output of the enforcement function. In this timed setting, *soundness* states that the output of an enforcement function should eventually always satisfy the property, meaning that the output is allowed to not satisfy the property at some point, provided that it will satisfy it in the future:

Definition 3.14 (Soundness). An enforcement function E is *sound* with respect to φ in a time-extension-closed set $S \subseteq \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ if:

$$\forall (\sigma, t) \in S, \exists t' \geq t, \forall t'' \geq t', E(\sigma, t'') \models \varphi.$$

An enforcement function is sound in a time-extension-closed set S if for any (σ, t) in S , the output of the enforcement function with input σ from date t satisfies the property in the future. As in the untimed setting, soundness is not defined for all words in $\text{tw}(\Sigma)$, but in a set of words, this time associated with dates. The reason is the same as in the untimed setting: the EM might not be able to ensure soundness from the beginning, because of bad uncontrollable sequences. Moreover, in the definition of soundness, the set S needs to be time-extension-closed to ensure that the property remains satisfied once the EM starts to operate.

Remark 2. Soundness could have been defined in the same way as in the untimed setting, *i.e.* stating that the output of an enforcement function should always satisfy the property. However, weakening soundness into enforcing “eventually always φ ” rather than φ itself allows to enforce more properties, and to let enforcement mechanisms produce longer outputs.

As in the untimed setting (see Definition 3.3), *compliance* states that uncontrollable events should be emitted instantaneously upon reception, and that controllable events can be delayed, but their order must remain unchanged:

Definition 3.15 (Compliance). Given an enforcement function E defined on an alphabet Σ , we say that E is *compliant* with respect to Σ_u and Σ_c , noted $\text{compliant}(E, \Sigma_u, \Sigma_c)$, if it satisfies the following constraints:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, E(\sigma, t) \preceq_{\text{d}\Sigma_c} \sigma$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, E(\sigma, t) =_{\Sigma_u} \text{obs}(\sigma, t)$
3. $\forall \sigma \in \text{tw}(\Sigma), \forall (t, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u, \sigma \cdot (t, u) \in \text{tw}(\Sigma) \implies E(\sigma, t) \cdot (t, u) \preceq E(\sigma \cdot (t, u), t)$.

Compliance is similar to the one in the untimed setting except that the controllable events can be delayed. However, their order must not be modified by the EM, that is, when considering the projections on controllable events, the output should be a delayed prefix of the input, as required by the first constraint of Definition 3.15. Any uncontrollable event is released immediately when received, that is, when considering the projections on uncontrollable events, the output should be equal to the input, as per the second constraint. The third constraint requires that an enforcement mechanism does not emit controllable events before a newly received uncontrollable event. In other words, it preserves causality: the reception of the uncontrollable event can cause the output of some other events by the EM, but only *after* the uncontrollable event (which can be seen as a notification that the event has already happened).

We say that a property is *enforceable* whenever there exists a sound and compliant enforcement function for this property.

For a compliant enforcement function $E : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma)$, and a timed word $\sigma \in \text{tw}(\Sigma)$, we note $E(\sigma)$ the value of E with input σ at infinite time (*i.e.* when it has stabilised). More formally, $E(\sigma) = E(\sigma, t)$, where $t \in \mathbb{R}_{\geq 0}$ is such that for all $t' \geq t$, $E(\sigma, t') = E(\sigma, t)$. Since σ is finite, and E is compliant, the output of E with input word σ is finite, thus such a t exists.

As described in the untimed setting, some enforcement mechanisms can be “better” than others, in the sense that they output more events, and thus modify less the input than others (see Section 3.1.2). Thus, we also define *optimality* for timed enforcement functions, as follows:

Definition 3.16 (Optimality). We say that an enforcement function $E : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma)$ that is compliant with respect to Σ_u and Σ_c and sound in a time-extension-closed set $S \subseteq \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ is *optimal* in S if:

$$\begin{aligned}
 & \forall E' : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma), \forall \sigma \in \text{tw}(\Sigma), \forall (t, a) \in \mathbb{R}_{\geq 0} \times \Sigma, \\
 & (\text{compliant}(E', \Sigma_u, \Sigma_c) \wedge \sigma \cdot (t, a) \in \text{tw}(\Sigma) \wedge (\sigma, t) \in S \wedge \\
 & E'(\sigma, t) = E(\sigma, t) \wedge E(\sigma \cdot (t, a)) \prec_d E'(\sigma \cdot (t, a))) \\
 & \implies \exists \sigma_u \in \text{tw}(\Sigma_u), E'(\sigma \cdot (t, a) \cdot \sigma_u) \not\preceq \varphi
 \end{aligned}$$

Optimality states that outputting a greater word (with respect to \preceq_d) than the output of an optimal enforcement function leads to either compliance or soundness not being guaranteed. This holds from the point where the input begins to belong to the set in which the function is optimal, and since it is time-extension-closed, the input will belong to this set afterwards. In Definition 3.16, E is an optimal enforcement function, and E' is another compliant enforcement function, that we consider having a greater output (with respect to \preceq_d) than E for some input word $\sigma \cdot (t, a)$. Then, since E is optimal, E' is not sound, because there exists a word of uncontrollable events such that the output of E' after receiving it eventually violates φ .

3.2.2 A Sound, Compliant and Optimal Enforcement Function

In this section, as in the untimed setting (see Section 3.1), we define S , I , G , store_φ , E_φ and $\text{Pre}(\varphi)$ such that E_φ is an enforcement function that is sound in $\text{Pre}(\varphi)$ with respect to φ , compliant with respect to Σ_u and Σ_c , and optimal in $\text{Pre}(\varphi)$.

An EM delaying events should buffer them until it can output them. Being able to enforce φ depends on the possibility of computing a timed word with the events of the buffer, even when receiving some uncontrollable events, that leads to an accepting state from the current one. Thus, we define, for every sequence σ of controllable actions, two sets of states of the semantics of \mathcal{A}_φ , $S(\sigma)$ and $I(\sigma)$. $S(\sigma)$ is the largest set such that from any of its states, it is possible to wait before emitting a word that leads to F_G , knowing that all along the path, receiving uncontrollable events will not prevent from computing such a word again. $I(\sigma)$ is the set of states from which it is possible to emit the first event of σ and reach a state from which it is possible to compute a word that leads to F_G , again such that receiving uncontrollable events does not prevent from eventually reaching F_G .

Definition 3.17 (I , S). For $\sigma \in \text{tw}(\Sigma)$, the sets of states of $\llbracket \mathcal{A}_\varphi \rrbracket$, $I(\sigma)$ and $S(\sigma)$, are inductively defined over sequences of controllable events as follows:

$$I(\epsilon) = \emptyset \quad S(\epsilon) = \{q \in F_G \mid q \text{ after } \text{tw}(\Sigma_u) \subseteq F_G\}$$

and, for $\sigma \in \Sigma_c^*$ and $a \in \Sigma_c$,

$$\begin{aligned} I(a \cdot \sigma) &= \text{Pred}_a(I(\sigma) \cup S(\sigma)), \\ S(\sigma \cdot a) &= S(\sigma) \cup \max_{\subseteq} (\{X \cup Y \subseteq Q \mid Y \subseteq F_G \wedge Y = \text{up}(Y) \wedge \\ &\quad (\forall x \in X, \exists i \in I(\sigma \cdot a), \exists \delta \in \mathbb{R}_{\geq 0}, x \text{ after } (\epsilon, \delta) = i \wedge \\ &\quad \forall t < \delta, x \text{ after } (\epsilon, t) \in X) \wedge \\ &\quad (X \cup Y) \cap \text{uPred}(\overline{X \cup Y \cup I(\sigma \cdot a)}) = \emptyset\}) \end{aligned}$$

Intuitively, in Definition 3.17, $S(\sigma)$ is the set of states of the semantics of φ that our EM will be allowed to reach with a buffer σ . It corresponds to the states from which the EM will be able to reach F_G , meaning that its output will satisfy the property, even if some uncontrollable events are received. From any state in $S(\sigma)$, the EM can compute a word of controllable events (taken from its buffer σ) leading to F_G , and if some uncontrollable events are received, it will also be able to compute a new word to reach F_G , with events taken from its (possibly modified due to previous emissions of events) buffer. The set $I(\sigma)$ is the set of states that the output of the enforcement mechanism will be authorised to “traverse”, meaning that the enforcement mechanism can emit the first event of its buffer σ immediately from these states, but not wait in them (contrary to the states in $S(\sigma)$, from which the EM could choose to wait before emitting a new event).

These sets are defined by induction on σ , which represents the buffer of the EM. If the EM has its buffer empty ($\sigma = \epsilon$), then the set of states from which it can emit a controllable event is empty, since it can only emit events from its buffer: $I(\epsilon) = \emptyset$. Nevertheless, some states in F_G can be such that all uncontrollable words lead to a state in F_G , meaning that from these states, the property will be satisfied even if some uncontrollable events are received. Consequently, $S(\epsilon) = \{q \in F_G \mid q \text{ after } \text{tw}(\Sigma_u) \subseteq F_G\}$.

If a new controllable event a is received, it is added to the buffer, and then the EM can decide to emit the first event of its buffer to reach a state that is in S or I for its new buffer, this explains the definition of $I(a \cdot \sigma)$. Adding a new event to the buffer gives more possibilities to the EM (since it could act as if it had not received this event), thus $S(\sigma) \subseteq S(\sigma \cdot a)$. Moreover, $S(\sigma \cdot a)$ is made of the union of two sets, X and Y . X is the set of states from which the EM can decide to wait before emitting the first event of its buffer, thus waiting from a state of X has to lead to a state in $I(\sigma \cdot a)$. Y is the set of states that are in F_G and from which the EM can decide to wait for a new uncontrollable event before doing anything. Since $Y \subseteq F_G$, if no uncontrollable event is to be received, the property is satisfied, and otherwise, the EM can decide what to emit to reach F_G . In order to ensure that receiving uncontrollable events do not prevent from being able to reach F_G with events from the buffer, X and Y are such that every uncontrollable event received from a state in X or Y leads to a state in X , Y , or $I(\sigma \cdot a)$. This is the purpose of the condition $(X \cup Y) \cap \text{uPred}(\overline{X \cup Y \cup I(\sigma \cdot a)}) = \emptyset$. On top of this, it is necessary to ensure that all the states reached while waiting from X or Y are in X or Y , otherwise there could be a state reached by the EM for which there is an uncontrollable event leading to a state from which it is impossible to reach F_G with events of the buffer, meaning that the enforcement would not be sound. This is ensured by the conditions $x \text{ after } (\epsilon, t) \in X$, and $Y = \text{up}(Y)$. To have the best EM possible, these sets are as large as possible.

Note that if X_1 and X_2 satisfy the conditions required for X , then $X_1 \cup X_2$

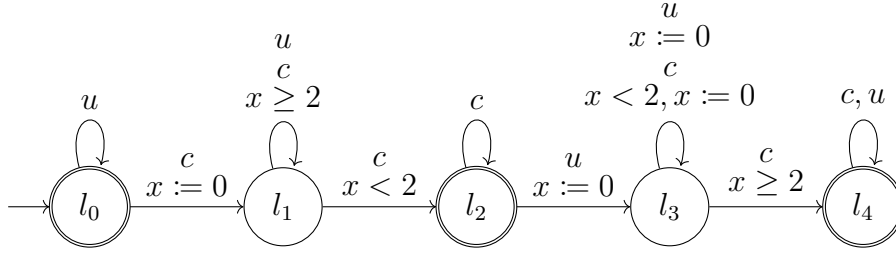


Figure 3.6 – Example property such that $I(c) = \{\langle l_3, x \rangle \mid x \geq 2\} \cup (\{l_4\} \times \mathbb{R}_{\geq 0})$ and $S(c) = \{l_0, l_2, l_4\} \times \mathbb{R}_{\geq 0}$.

also satisfies them. Thus, the bigger set satisfying these properties exists. The same holds for Y .

Example 3.7. Consider the property described in Fig. 3.6, with $\Sigma_u = \{u\}$ and $\Sigma_c = \{c\}$. This property is similar to Fig. 3.4 but with some clock constraints added. For this property, we will represent valuations as the image in $\mathbb{R}_{\geq 0}$ of the only clock. Then, $I(\epsilon) = \emptyset$ and $S(\epsilon) = \{l_0, l_4\} \times \mathbb{R}_{\geq 0}$. To calculate this, note that $F_G = \{l_0, l_2, l_4\} \times \mathbb{R}_{\geq 0}$, and:

- $\{l_0\} \times \mathbb{R}_{\geq 0}$ after $\text{tw}(\Sigma_u) = \{l_0\} \times \mathbb{R}_{\geq 0} \subseteq F_G$, thus $\{l_0\} \times \mathbb{R}_{\geq 0} \subseteq S(\epsilon)$.
- $\{l_2\} \times \mathbb{R}_{\geq 0}$ after $\text{tw}(\Sigma_u) = \langle l_3, 0 \rangle \notin F_G$, thus $\{l_2\} \times \mathbb{R}_{\geq 0}$ after $\text{tw}(\Sigma_u) \cap S(\epsilon) = \emptyset$.
- $\{l_4\} \times \mathbb{R}_{\geq 0}$ after $\text{tw}(\Sigma_u) = \{l_4\} \times \mathbb{R}_{\geq 0} \subseteq F_G$, thus $\{l_4\} \times \mathbb{R}_{\geq 0} \subseteq S(\epsilon)$.

It follows that $I(c) = \text{Pred}_c(S(\epsilon)) = \{\langle l_3, x \rangle \mid x \geq 2\} \cup (\{l_4\} \times \mathbb{R}_{\geq 0})$. Now, note that for any $q = \langle l_3, x \rangle$ such that $x < 2$, q after $(\epsilon, 2 - x) = \langle l_3, 2 \rangle \in I(c)$, and for any $t < 2 - x$, q after $(\epsilon, t) \in \{\langle l_3, x \rangle \mid x < 2\}$. Thus, if $X = I(c) \cup \{\langle l_3, x \rangle \mid x < 2\} = \{l_3, l_4\} \times \mathbb{R}_{\geq 0}$, then for any $q \in X$, there exists $i \in I(c)$ and $\delta \in \mathbb{R}_{\geq 0}$ such that q after $(\epsilon, \delta) = i$ and for any $t < \delta$, q after $(\epsilon, t) \in X$. If $q \in I(c)$, then $i = q$ and $\delta = 0$. Moreover, if $Y = \{l_0, l_2, l_4\} \times \mathbb{R}_{\geq 0}$, then $Y \subseteq F_G$ and $\text{up}(Y) = Y$. Since $X \cup Y = \{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}$, $X \cup Y \cup I(c) = \{l_1\} \times \mathbb{R}_{\geq 0}$, and $\text{uPred}(\{l_1\} \times \mathbb{R}_{\geq 0}) = \text{Pred}_u(\{l_1\} \times \mathbb{R}_{\geq 0}) = \{l_1\} \times \mathbb{R}_{\geq 0}$. Thus, $X \cup Y \cap \text{uPred}(X \cup Y \cup I(c)) = \emptyset$. This means that $\{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0} \subseteq S(c)$. Since for any $q \in \{l_1\} \times \mathbb{R}_{\geq 0}$, $q \notin F_G$ and there does not exist $i \in I(c)$ such that q after $(\epsilon, \delta) = i$ for some $\delta \in \mathbb{R}_{\geq 0}$, $S(c)$ can not be bigger than $\{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}$. Thus, $S(c) = \{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}$.

We can calculate in the same way that $I(c.c) = (\{l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}) \cup \{\langle l_1, x \rangle \mid x < 2\}$, $S(c.c) = (\{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}) \cup \{\langle l_1, x \rangle \mid x < 2\}$, $I(c.c.c) = (\{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}) \cup \{\langle l_1, x \rangle \mid x < 2\}$, and $S(c.c.c) = (\{l_0, l_2, l_3, l_4\} \times \mathbb{R}_{\geq 0}) \cup \{\langle l_1, x \rangle \mid x < 2\}$.

To be sound, an enforcement mechanism must output the events of its buffer only if the state reached by its output so far is in $I(\sigma)$, with σ its buffer.

If the state reached by its output is in $S(\sigma)$, the enforcement mechanism must wait before outputting something, or wait indefinitely. Thus, to be sound, an enforcement mechanism must have at least three c actions in its buffer to output the first one, and it must output two of them with less than 2 time units between them. Then, when in location l_2 , it must keep at least one c action in its buffer to be able to reach location l_4 in case an uncontrollable event would occur, leading to location l_3 .

Function $G : Q \times \Sigma_c^* \rightarrow 2^{\text{tw}(\Sigma)}$ gives, for a state $q \in Q$ and a sequence of controllable events $\sigma \in \Sigma_c^*$, the set of timed words made with the actions of σ that can be output from q in a safe way (*i.e.* all the states reached while emitting the word are in the set S corresponding to what remains from σ):

Definition 3.18 (G). For $q \in Q$ and $w \in \Sigma_c^*$,

$$G(q, \sigma) = \{w \in \text{tw}(\Sigma) \mid \Pi_\Sigma(w) \preceq \sigma \wedge q \text{ after } w \in F_G \wedge \forall t \in \mathbb{R}_{\geq 0}, q \text{ after } (w, t) \in S(\Pi_\Sigma(\text{obs}(w, t))^{-1} \cdot \sigma)\}.$$

It is now possible to use G to define an enforcement function for φ , denoted as E_φ :

Definition 3.19 (Functions $\text{store}_\varphi, E_\varphi$). Let $\text{store}_\varphi : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma) \times \text{tw}(\Sigma_c) \times \Sigma_c^*$ be the function inductively defined by:

$$\forall t \in \mathbb{R}_{\geq 0}, \text{store}_\varphi(\epsilon, t) = (\epsilon, \epsilon, \epsilon)$$

and, for $\sigma \in \text{tw}(\Sigma)$, $(t', a) \in \mathbb{R}_{\geq 0} \times \Sigma$ such that $\sigma.(t', a) \in \text{tw}(\Sigma)$, and $t \geq t'$, if $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$, then

$$\text{store}_\varphi(\sigma.(t', a), t) = \begin{cases} (\sigma_s.(t', a). \text{obs}(\sigma'_b, t), \sigma'_b, \sigma'_c) & \text{if } a \in \Sigma_u \\ (\sigma_s. \text{obs}(\sigma''_b, t), \sigma''_b, \sigma''_c) & \text{if } a \in \Sigma_c \end{cases}$$

with: for $q \in Q$ and $w \in \Sigma_c^*$,

$$\begin{aligned} \kappa_\varphi(q, w) &= \min_{\leq_{\text{lex}}}(\max_{\preceq}(G(q, w) \cup \{\epsilon\})), \\ \text{buf}_c &= \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c, \\ t_1 &= \min(\{t'' \in \mathbb{R}_{\geq 0} \mid t'' \geq t' \wedge G(\text{Reach}(\sigma_s.(t', a), t''), \text{buf}_c) \neq \emptyset\} \cup \{+\infty\}), \\ \sigma'_b &= \kappa_\varphi(\text{Reach}(\sigma_s.(t', a), \min(\{t, t_1\})), \text{buf}_c) +_t \min(\{t, t_1\}), \\ \sigma'_c &= \Pi_\Sigma(\sigma'_b)^{-1} \cdot \text{buf}_c, \\ t_2 &= \min(\{t'' \in \mathbb{R}_{\geq 0} \mid t'' \geq t' \wedge G(\text{Reach}(\sigma_s, t''), \text{buf}_c \cdot a) \neq \emptyset\} \cup \{+\infty\}), \\ \sigma''_b &= \kappa_\varphi(\text{Reach}(\sigma_s, \min(\{t, t_2\})), \text{buf}_c \cdot a) +_t \min(\{t, t_2\}), \\ \sigma''_c &= \Pi_\Sigma(\sigma''_b)^{-1} \cdot (\text{buf}_c \cdot a). \end{aligned}$$

For $\sigma \in \text{tw}(\Sigma)$, and $t \in \mathbb{R}_{\geq 0}$, we define $E_\varphi(\sigma, t) = (\Pi_1(\text{store}_\varphi(\text{obs}(\sigma, t), t)))$.

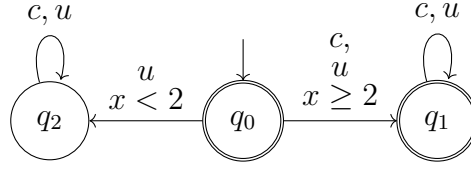


Figure 3.7 – Property that becomes enforceable as time elapses

Function store_φ takes a timed word σ and a date t as input, and outputs three words: σ_s , σ_b , and σ_c . The timed word σ_s is the output of the enforcement function at time t . The timed word σ_b is composed of controllable events. It is the word that is to be output after the date of the last event of the input, if no other event is received, such that $\sigma_s \cdot \sigma_b = E_\varphi(\sigma)$, *i.e.* the output of the enforcement function at an infinite time. The untimed word σ_c is composed of the remaining controllable actions of the buffer. It can be used to compute a new output if other events are received.

As time elapses after the last event of the input, σ_s is modified to output the events of σ_b when the dates are reached. Since letting time elapse can disable some transitions, it is possible to reach a “safe” state without emitting any event, and thus σ_b can also change as time elapses. However, σ_b changes as time elapses at most once, changing from ϵ to a word in G . This change of σ_b when letting time elapse can only happen once, since G will not be empty anymore once it has become non-empty. t_1 and t_2 are used for this purpose, they both represent the time at which G becomes non-empty, if $a \in \Sigma_u$ or $a \in \Sigma_c$ respectively. Words are thus calculated from this point whenever G has become non-empty, to ensure that what has already been output is not modified. If G is still empty, then $\min(\{t, t_1\})$ (or $\min(\{t, t_2\})$, depending on whether $a \in \Sigma_c$ or $a \in \Sigma_u$) equals to t , meaning that $\sigma_b = \epsilon$. Most of the time, t_1 , or t_2 is equal to t' , it is not the case only when G was still empty at time t' , but if G was not empty at date t' , then t_1 (or t_2) is equal to t' .

To visualise this, consider the property described in Fig. 3.7. Considering that $\Sigma_u = \{u\}$ and $\Sigma_c = \{c\}$, this property is not enforceable since word $(1, u)$ leads to q_2 , that is a non-accepting sink state, and this word can not be corrected. Nevertheless, if clock x reaches at least two time units, then the property becomes enforceable (actually, the identity function is then a sound enforcement function).

The word of controllable actions σ_c contains the actions of the input that have not been output and do not belong to σ_b . It is used to compute the new value of σ_b when possible. When receiving a new event in the input, it is appended to σ_s if it is an uncontrollable event, or the action is appended to the buffer if it is a controllable one. Then, σ_b is computed again, from the new state reached if it was an uncontrollable event, or with the new buffer if it was controllable. Note that t_1 and t_2 may not exist, since they are minima

of an interval that can be open, depending on the strictness of the considered guard. In this case, one should consider the infimum instead of the minimum, and add an infinitesimal delay, such that the required transition is taken.

As mentioned previously, an EM may not be sound from the beginning of an execution, but some uncontrollable events (or letting time elapse, see Fig. 3.7) may lead to a state from which it becomes possible to be sound. Whenever σ_b is in G , then it will always be, meaning that the output of E_φ will eventually reach a state in F_G , *i.e.* it will eventually satisfy φ . Thus, E_φ eventually satisfies φ as soon as the state reached so far is in $S(\sigma_b)$ or $I(\sigma_b)$. This leads to the definition of $\text{Pre}(\varphi, t)$, which is the set of timed words for which E_φ ensures soundness at time t . For $\sigma \in \text{tw}(\Sigma)$, if $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t)$, then σ is in $\text{Pre}(\varphi, t)$ if and only if the set $G(\text{Reach}(\text{obs}(\sigma_s, t), \Pi_\Sigma(\text{nobs}(\sigma_b, t)).\sigma_c))$ is not empty. Then, $\text{Pre}(\varphi, t)$ is used to define $\text{Pre}(\varphi)$, which is the set in which E_φ is sound:

Definition 3.20 ($\text{Pre}(\varphi)$). For $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$,

$$\begin{aligned} \text{Pre}(\varphi, t) &= \{ \sigma \in \text{tw}(\Sigma) \mid \exists \sigma' \preceq \sigma, \exists t' \leq t, \\ &\quad G(\text{Reach}(\text{obs}(\sigma', t')|_{\Sigma_u}, t'), \Pi_\Sigma(\text{obs}(\sigma', t')|_{\Sigma_c})) \neq \emptyset \} \\ \text{Pre}(\varphi) &= \{ (\sigma, t) \mid \sigma \in \text{Pre}(\varphi, t) \} \end{aligned}$$

In Definition 3.20, the definition of $\text{Pre}(\varphi, t)$ considers words that have a prefix that satisfies the required condition, for a time that is at most t , meaning that $\text{Pre}(\varphi, t)$ is extension-closed. Thus, $\text{Pre}(\varphi)$ is time-extension-closed, as required by Definition 3.14.

Since the output of our enforcement function consists only of the uncontrollable events from the input while it cannot ensure soundness, if $G(\text{Reach}(\text{obs}(\sigma, t)|_{\Sigma_u}, t), \Pi_\Sigma(\text{obs}(\sigma, t)|_{\Sigma_c}))$ is not empty, this means that there exists a word that is “safe” to emit, thus the enforcement function is sound for input σ at date t . Thus, $\text{Pre}(\varphi, t)$ is the set of inputs for which E_φ is sound after date t , and then E_φ is sound for any input in $\text{Pre}(\varphi)$ after its associated date.

Proposition 3.6. E_φ as defined in Definition 3.19 is an enforcement function, as per Definition 3.13.

Sketch of proof. We have to show that for all $\sigma \in \text{tw}(\Sigma)$, for all $t \in \mathbb{R}_{\geq 0}$ and $t' \geq t$, $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$, and for all (t, a) such that $\sigma \cdot (t, a) \in \text{tw}(\Sigma)$, $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma \cdot (t, a), t)$. To prove this, we first show by induction that $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$. Considering (t'', a) such that $\sigma \cdot (t'', a) \in \text{tw}(\Sigma)$, we distinguish different cases according to the values of t'' compared to t and t' :

- $t'' \leq t$. Then, in the definition of store_φ , t_1 (or t_2 , if a is controllable) has the same value in $\text{store}_\varphi(\sigma, t)$ and $\text{store}_\varphi(\sigma \cdot (t'', a), t')$. Then, comparing t to t_1 , either $E_\varphi(\sigma \cdot (t'', a), t) = \epsilon$ if $t < t_1$, and then $E_\varphi(\sigma \cdot (t'', a), t) \preceq$

$E_\varphi(\sigma \cdot (t'', a), t')$, or $t \geq t_1$, and then there exists σ_s and σ_b such that $E_\varphi(\sigma \cdot (t'', a), t) = \sigma_s \cdot \text{obs}(\sigma_b, t)$ and $E_\varphi(\sigma \cdot (t'', a), t') = \sigma_s \cdot \text{obs}(\sigma_b, t')$, meaning that $E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

- $t'' \geq t'$. Then the proposition holds because in the definition of E_φ , only the observation of the input word at the given time is considered, meaning that $E_\varphi(\sigma \cdot (t'', a), t) = E_\varphi(\sigma, t)$ and $E_\varphi(\sigma \cdot (t'', a), t') = E_\varphi(\sigma, t')$. By induction hypothesis, the proposition thus holds.
- $t < t'' < t'$. Then, $E_\varphi(\sigma \cdot (t'', a), t) = E_\varphi(\sigma, t)$, and $E_\varphi(\sigma \cdot (t'', a), t') = \Pi_1(\text{store}_\varphi(\sigma \cdot (t'', a), t'))$, meaning that, looking at the definition of store_φ , $E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

Thus, $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$. Then, what remains to show is that if $\sigma \cdot (t, a) \in \text{tw}(\Sigma)$, then $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma \cdot (t, a), t)$. Following the definition of store_φ , it is clear that $\Pi_1(\text{store}_\varphi(\sigma, t)) \preceq \Pi_1(\text{store}_\varphi(\sigma \cdot (t, a), t))$, and thus $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma \cdot (t, a), t)$.

Proposition 3.7. *E_φ is sound with respect to φ in $\text{Pre}(\varphi)$ as per Definition 3.14.*

Sketch of proof. As in the untimed setting, the proof is made by induction on the input $\sigma \in \text{tw}(\Sigma)$. Similarly to the untimed setting, considering $\sigma \in \text{tw}(\Sigma)$, $t \in \mathbb{R}_{\geq 0}$, and (t', a) such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$, there are three possibilities:

- $(\sigma \cdot (t', a), t) \notin \text{Pre}(\varphi)$. Then, the proposition holds.
- $(\sigma \cdot (t', a), t) \in \text{Pre}(\varphi)$, but $(\sigma, t') \notin \text{Pre}(\varphi)$. Then, this is when the input reaches $\text{Pre}(\varphi)$. Considering the definition of $\text{Pre}(\varphi)$, we then prove that it is possible to emit a word with the controllable events seen so far, leading to an accepting state in S .
- $(\sigma, t') \in \text{Pre}(\varphi)$ (and thus $(\sigma \cdot (t', a), t)$ too). Then, we prove again that there exists a controllable word made with the events which have not been output yet leading to an accepting state that is in S , but this time considering the definitions of S and I .

Proposition 3.8. *E_φ is compliant, as per Definition 3.15.*

Sketch of proof. As in the untimed setting, the proof is made by induction on the input σ , considering the different cases where the new event is controllable or uncontrollable. The only difference with the untimed setting is that one should consider dates on top of actions.

Proposition 3.9. *E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 3.16.*

Sketch of proof. This proof is made by induction on the input σ . Whenever $\sigma \in \text{Pre}(\varphi)$, since E_φ is sound in $\text{Pre}(\varphi)$, then $E_\varphi(\sigma)$ is the maximal word (with respect to \preceq_d) that satisfies φ and is safe to output. It is maximal because in the definition of store_φ , κ_φ returns the longest word with lower delays (for lexicographic order), which corresponds to the maximum with respect to \preceq_d . Thus, outputting a greater word (with respect to \preceq_d) would lead to G being empty, meaning that the EM would not be sound. Thus, E_φ is optimal in $\text{Pre}(\varphi)$, since it outputs the maximal word with respect to \preceq_d that allows to be sound and compliant.

3.2.3 Enforcement Monitors

As in the untimed setting, we define an operational description of an EM whose output is exactly the output of E_φ , as defined in Definition 3.19.

Definition 3.21. An *enforcement monitor* \mathcal{E} for φ is a transition system $\langle C^\mathcal{E}, c_0^\mathcal{E}, \Gamma^\mathcal{E}, \hookrightarrow_\mathcal{E} \rangle$ such that:

- $C^\mathcal{E} = \text{tw}(\Sigma) \times \Sigma_c^* \times Q \times \mathbb{R}_{\geq 0} \times \{\top, \perp\}$ is the set of configurations.
- $c_0^\mathcal{E} = \langle \epsilon, \epsilon, q_0, 0, \perp \rangle \in C^\mathcal{E}$ is the initial configuration.
- $\Gamma^\mathcal{E} = ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}) \times \text{Op} \times ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})$ is the alphabet, composed of an optional input, an operation and an optional output.
The set of operations is $\{\text{compute}(\cdot), \text{dump}(\cdot), \text{pass-uncont}(\cdot), \text{store-cont}(\cdot), \text{delay}(\cdot)\}$.
Whenever $(\sigma, \bowtie, \sigma') \in \Gamma^\mathcal{E}$, it will be noted $\sigma / \bowtie / \sigma'$.
- $\hookrightarrow_\mathcal{E}$ is the transition relation defined as the smallest relation obtained by applying the following rules given by their priority order:
 - **Compute:** $\langle \epsilon, \sigma_c, q, t, \perp \rangle \xrightarrow{\epsilon / \text{compute}(\cdot) / \epsilon}_\mathcal{E} \langle \sigma'_b, \sigma'_c, q, t, \top \rangle$, if $G(q, \sigma_c) \neq \emptyset$, with $\sigma'_b = \kappa_\varphi(q, \sigma_c) +_t t$, and $\sigma'_c = \Pi_\Sigma(\sigma'_b)^{-1} \cdot \sigma_c$,
 - **Dump:** $\langle (t_b, a) \cdot \sigma_b, \sigma_c, q, t_b, \top \rangle \xrightarrow{\epsilon / \text{dump}((t_b, a)) / (t_b, a)}_\mathcal{E} \langle \sigma_b, \sigma_c, q', t_b, \top \rangle$, with $q' = q$ after $(0, a)$,
 - **Pass-uncont:** $\langle \sigma_b, \sigma_c, q, t, b \rangle \xrightarrow{(t, a) / \text{pass-uncont}((t, a)) / (t, a)}_\mathcal{E} \langle \epsilon, \Pi_\Sigma(\sigma_b) \cdot \sigma_c, q', t, \perp \rangle$, with $q' = q$ after $(0, a)$,
 - **Store-cont:** $\langle \sigma_b, \sigma_c, q, t, b \rangle \xrightarrow{(t, c) / \text{store-cont}((t, c)) / \epsilon}_\mathcal{E} \langle \epsilon, \Pi_\Sigma(\sigma_b) \cdot \sigma_c \cdot c, q, t, \perp \rangle$,
 - **Delay:** $\langle \sigma_b, \sigma_c, (l, v), t, b \rangle \xrightarrow{\epsilon / \text{delay}(\delta) / \epsilon}_\mathcal{E} \langle \sigma_b, \sigma_c, (l, v + \delta), t + \delta, b \rangle$.

In a configuration $\langle \sigma_b, \sigma_c, q, t, b \rangle$, σ_b is the word to be output as time elapses; σ_c is the sequence of controllable actions from the input that are not used in σ_b and have not been output yet; q is the state of the semantics reached after reading what has already been output; t is the current time instant, *i.e.* the time elapsed since the beginning of the run; and b indicates whether σ_b and σ_c should be computed (due to the reception of a new event for example).

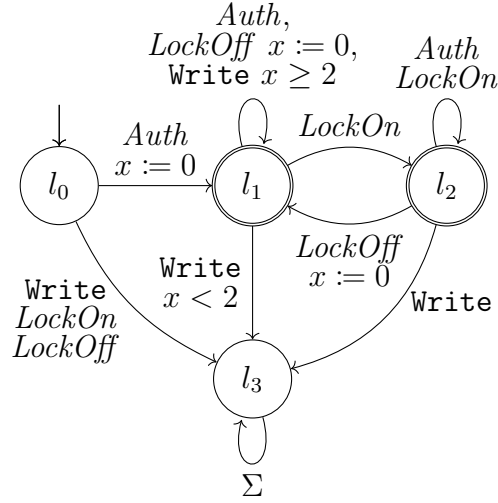
The timed word σ_b corresponds to $\text{nobs}(\sigma_b, t)$ from the definition of store_φ , whereas σ_c is the same as in the definition of store_φ . The state q represents σ_s from the definition of store_φ , such that $q = \text{Reach}(\sigma_s, t)$. Thus, the following proposition holds:

Proposition 3.10. *The output of \mathcal{E} as per Definition 3.21 for input σ is $E_\varphi(\sigma)$ as per Definition 3.19.*

As in the untimed setting, in Proposition 3.10, the output of the enforcement monitor is the concatenation of the outputs of the word labelling the path followed by the enforcement monitor when reading σ . A formal definition is given in the proof of this proposition, in appendix A.1.2.

Sketch of proof. The proof is done by induction on the input $\sigma \in \text{tw}(\Sigma)$. When receiving a new event, rule $\text{store-cont}()$ can be applied if it is controllable, or rule $\text{pass-uncont}()$ if it is uncontrollable. Doing so, the last member of the configuration is set to \perp , meaning that the word to be emitted can be computed. If the input is in $\text{Pre}(\varphi)$, then rule $\text{compute}()$ can be applied, and then the second member of the configuration will have the same value as the second member of store_φ , and the same goes for the third members. Then, rule $\text{delay}()$ can be applied, to reach the date of the first event in the second member of the current configuration, and then rule $\text{dump}()$ can be applied to output it. This process can be repeated until the desired date is reached. Thus, when date t is reached, what has been emitted since the last rule $\text{store-cont}()$ or $\text{pass-uncont}()$ is $\text{obs}(\sigma_b, t)$, where σ_b was computed by rule $\text{compute}()$ as second member. Considering the definition of store_φ , it follows that the output of \mathcal{E} with input σ at date t is $E_\varphi(\sigma, t)$.

Example 3.8. Consider Fig. 3.8, representing property φ_t , modelling the use of some shared writable device. Property φ_t is similar to property φ_{ex} (see Fig. 3.2), except that when in state l_1 , one must wait two time units before emitting a **Write** event. The status of a lock is given through the uncontrollable events *LockOn* and *LockOff* indicating that the lock has been locked by someone else, and that it has been unlocked, respectively. The uncontrollable event *Auth* is sent by the device to authorise writings. Once the *Auth* event is received, the system is able to send the controllable event **Write** after having waited some time for synchronisation. Each time the lock is taken and released, it must also wait before issuing a new **Write** order. The sets of events are: $\Sigma_c = \{\text{Write}\}$ and $\Sigma_u = \{\text{Auth}, \text{LockOff}, \text{LockOn}\}$.


 Figure 3.8 – Property φ_t

Now, let us follow the output of the store_φ function over time with the word $\sigma = (1, \text{Auth}) . (2, \text{LockOn}) . (4, \text{Write}) . (5, \text{LockOff}) . (6, \text{LockOn}) . (7, \text{Write}) . (8, \text{LockOff})$ as input: let $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\text{obs}(\sigma, t), t)$. Then the values taken by σ_s , σ_b and σ_c over time are given in Table 3.3. To calculate them, notice that for all valuation $\nu : \{x\} \rightarrow \mathbb{R}_{\geq 0}$, $(l_1, \nu) \in S(\epsilon)$, and $(l_2, \nu) \in S(\epsilon)$, since all uncontrollable words from l_1 and l_2 lead to l_1 or l_2 , which are both accepting states.

We can also follow the execution of an enforcement monitor enforcing property φ_t (see Fig. 3.8), watching the evolution of the configurations as semantic rules are applied. In a configuration, the input is on the right, the output on the left, and the middle is the current configuration of the enforcement monitor. The variable t defines the global time of the execution. Figure 3.9 shows the execution of the enforcement monitor with input $(1, \text{Auth}) . (2, \text{LockOn}) . (4, \text{Write}) . (5, \text{LockOff}) . (6, \text{LockOn}) . (7, \text{Write}) . (8, \text{LockOff})$. In Fig. 3.9, valuations are represented as integers, giving the value of the unique clock x of the property, *LockOff* is abbreviated as *off*, *LockOn* as *on*, and *Write* as *w*. First column depicts the dates of events, then red text is the current output (σ_s) of the EM, blue text shows the evolution of σ_b and green text depicts the remaining input word at this date. We can observe, as stated by Proposition 3.10, that the final output is the same as the one of the enforcement function: $(1, \text{Auth}) . (2, \text{on}) . (5, \text{off}) . (6, \text{on}) . (8, \text{off}) . (10, \text{w}) . (10, \text{w})$.

Remark 3. An EM as per Definition 3.21 outputs longer timed words than the approach in Pinisetty *et al.* [2012] and Pinisetty *et al.* [2014a] when applied only with controllable events thanks to optimality considerations. Consider the property described in Fig. 3.10 over a set of controllable actions Σ , such that *Write* $\in \Sigma$. With timed word $(1, \text{Write}) . (1.5, \text{Write})$ as input to the EM,

t = 0	$\epsilon / \langle \epsilon, \epsilon, (l_0, 0), 0, \perp \rangle / (1, Auth) . (2, on) . (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{delay}(1)$
t = 1	$\epsilon / \langle \epsilon, \epsilon, (l_0, 1), 1, \perp \rangle / (1, Auth) . (2, on) . (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{pass-uncont}((1, Auth))$
t = 1	$(1, Auth) / \langle \epsilon, \epsilon, (l_1, 0), 1, \perp \rangle / (2, on) . (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{compute}()$
t = 1	$(1, Auth) / \langle \epsilon, \epsilon, (l_1, 0), 1, \top \rangle / (2, on) . (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{delay}(1)$
t = 2	$(1, Auth) / \langle \epsilon, \epsilon, (l_1, 1), 2, \top \rangle / (2, on) . (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{pass-uncont}((2, on))$
t = 2	$(1, Auth) . (2, on) / \langle \epsilon, \epsilon, (l_2, 1), 2, \perp \rangle / (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{compute}()$
t = 2	$(1, Auth) . (2, on) / \langle \epsilon, \epsilon, (l_2, 1), 2, \top \rangle / (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{delay}(2)$
t = 4	$(1, Auth) . (2, on) / \langle \epsilon, \epsilon, (l_2, 3), 4, \top \rangle / (4, w) . (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{store-cont}((4, w))$
t = 4	$(1, Auth) . (2, on) / \langle \epsilon, (4, w), (l_2, 3), 4, \perp \rangle / (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{compute}()$
t = 4	$(1, Auth) . (2, on) / \langle \epsilon, (4, w), (l_2, 3), 4, \top \rangle / (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{delay}(1)$
t = 5	$(1, Auth) . (2, on) / \langle \epsilon, (4, w), (l_2, 4), 5, \top \rangle / (5, off) . (6, on) . (7, w) . (8, off)$ $\downarrow \text{pass-uncont}((5, off))$
t = 5	$(1, Auth) . (2, on) . (5, off) / \langle \epsilon, (7, w), (l_1, 0), 5, \perp \rangle / (6, on) . (7, w) . (8, off)$ $\downarrow \text{compute}()$
t = 5	$(1, Auth) . (2, on) . (5, off) / \langle (7, w), \epsilon, (l_1, 0), 5, \top \rangle / (6, on) . (7, w) . (8, off)$ $\downarrow \text{delay}(1)$
t = 6	$(1, Auth) . (2, on) . (5, off) / \langle (7, w), \epsilon, (l_1, 1), 6, \top \rangle / (6, on) . (7, w) . (8, off)$ $\downarrow \text{pass-uncont}((6, on))$
t = 6	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (l_2, 1), 6, \perp \rangle / (7, w) . (8, off)$ $\downarrow \text{compute}()$
t = 6	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (l_2, 1), 6, \top \rangle / (7, w) . (8, off)$ $\downarrow \text{delay}(1)$
t = 7	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (l_2, 2), 7, \top \rangle / (7, w) . (8, off)$ $\downarrow \text{store-cont}((7, w))$
t = 7	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (7, w), (l_2, 2), 7, \perp \rangle / (8, off)$ $\downarrow \text{compute}()$
t = 7	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (7, w), (l_2, 2), 7, \top \rangle / (8, off)$ $\downarrow \text{delay}(1)$
t = 8	$(1, Auth) . (2, on) . (5, off) . (6, on) / \langle \epsilon, (7, w), (7, w), (l_2, 3), 8, \top \rangle / (8, off)$ $\downarrow \text{pass-uncont}((8, off))$
t = 8	$(1, Auth) . (2, on) . (5, off) . (6, on) . (8, off) / \langle \epsilon, (10, w), (10, w), (l_1, 0), 8, \perp \rangle / \epsilon$ $\downarrow \text{compute}()$
t = 8	$(1, Auth) . (2, on) . (5, off) . (6, on) . (8, off) / \langle (10, w), (10, w), \epsilon, (l_1, 0), 8, \top \rangle / \epsilon$ $\downarrow \text{delay}(2)$
t = 10	$(1, Auth) . (2, on) . (5, off) . (6, on) . (8, off) / \langle (10, w), (10, w), \epsilon, (l_1, 2), 10, \top \rangle / \epsilon$ $\downarrow \text{dump}((10, w))$
t = 10	$(1, Auth) . (2, on) . (5, off) . (6, on) . (8, off) . (10, w) / \langle (10, w), \epsilon, (l_1, 2), 10, \top \rangle / \epsilon$ $\downarrow \text{dump}((10, w))$
t = 10	$(1, Auth) . (2, on) . (5, off) . (6, on) . (8, off) . (10, w) . (10, w) / \langle \epsilon, \epsilon, (l_1, 2), 10, \top \rangle / \epsilon$

Figure 3.9 – Execution of an enforcement monitor with input $(1, Auth) . (2, LockOn) . (4, Write) . (5, LockOff) . (6, LockOn) . (7, Write) . (8, LockOff)$

Table 3.3 – Values of $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_{\varphi_t}((1, \text{Auth}) . (2, \text{LockOn}) . (4, \text{Write}) . (5, \text{LockOff}) . (6, \text{LockOn}) . (7, \text{Write}) . (8, \text{LockOff}))$ over time.

t	σ_s	σ_b	σ_c
1	$(1, \text{Auth})$	ϵ	ϵ
2	$(1, \text{Auth}) . (2, \text{LockOn})$	ϵ	ϵ
4	$(1, \text{Auth}) . (2, \text{LockOn})$	ϵ	Write
5	$(1, \text{Auth}) . (2, \text{LockOn}) . (5, \text{LockOff})$	$(7, \text{Write})$	ϵ
6	$(1, \text{Auth}) . (2, \text{LockOn}) . (5, \text{LockOff}) . (6, \text{LockOn})$	ϵ	Write
7	$(1, \text{Auth}) . (2, \text{LockOn}) . (5, \text{LockOff}) . (6, \text{LockOn})$	ϵ	Write . Write
8	$(1, \text{Auth}) . (2, \text{LockOn}) . (5, \text{LockOff}) . (6, \text{LockOn}) . (8, \text{LockOff})$	$(10, \text{Write}) . (10, \text{Write})$	ϵ
10	$(1, \text{Auth}) . (2, \text{LockOn}) . (5, \text{LockOff}) . (6, \text{LockOn}) . (8, \text{LockOff}) . (10, \text{Write}) . (10, \text{Write})$	ϵ	ϵ

the output obtained with our approach at date $t = 4$ is $(4, \text{Write}) . (4, \text{Write})$ whereas the output obtained in [Pinisetty et al. \[2012\]](#) would be $(2, \text{Write})$.

Conclusion

In this chapter, we have defined sound, compliant and optimal enforcement mechanisms, modelled by functions and transition systems, for untimed and timed regular properties. In the next chapter, we revisit the definitions we have presented, replacing S and I by the use of a Büchi game. This aims at improving the performance of an implementation, by precomputing some decisions made by the enforcement mechanism.

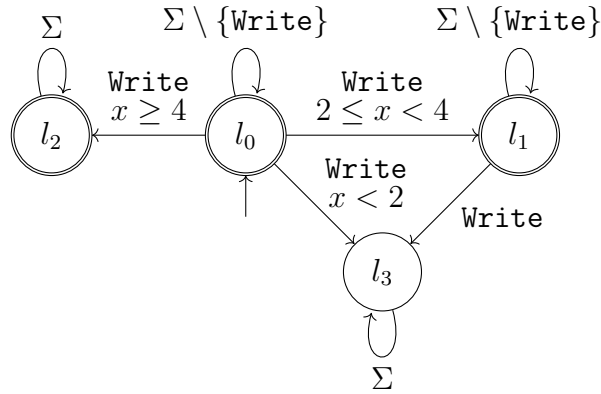


Figure 3.10 – Example of Property without uncontrollable events

Chapter 4

Enforcing Properties using a Büchi Game

Introduction

In this chapter, we revisit Chapter 3, using a Büchi game to ensure soundness, instead of S and I (see Definitions 3.8 and 3.17). Büchi games are well-suited for our purpose, since they correspond to games in which one tries to always be able to reach some nodes called Büchi nodes. An enforcement mechanism tries to always be able to reach an accepting state of the automaton representing the property, even if some uncontrollable events are received, thus this seems similar to solving a Büchi game. Using games allows us to precompute some of the decisions of the enforcement mechanism, thus improving the time overhead of an implementation.

We describe in Section 4.1 some notation changes, such as the use of delays instead of dates, then we define formally enforcement mechanisms in a similar way as in Chapter 3, but this time using a Büchi game to ensure soundness. As in Chapter 3, such enforcement mechanisms are defined for both untimed (Section 4.2) and timed regular properties (Section 4.3).

The work described in this chapter has been published in [Renard *et al.* \[2017c\]](#).

4.1 Notation Changes

In this chapter, some notation changes. The notation changes are essentially made in the timed setting, and are mostly due to the use of delays instead of dates in the definition of timed words. The use of delays seems more appropriate in this section because the use of games make delays appear naturally. This section lists all the modifications that are made to the notation.

4.1.1 Timed Words

Timed words are represented with delays instead of dates as in Chapter 3. Using delays has some advantages over dates. Considering delays, a timed word σ over an alphabet of actions Σ is a word over $\mathbb{R}_{\geq 0} \times \Sigma$, *i.e.* $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Unlike with dates, no consideration on the time has to be taken into account (remember that with dates, it is required that dates are increasing). The use of delays also seems more appropriate because we build enforcement functions by induction, taking only the current state into account, thus all the timings are calculated relatively to the current time, not to the origin. We note $\text{tw}(\Sigma) = (\mathbb{R}_{\geq 0} \times \Sigma)^*$ the set of timed words over Σ . Note that the definition of an *event* is still the same: an event is an element $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, but δ now represents a delay and not a date. Thus, for $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, we define $\text{delay}((\delta, a)) = \delta$. Since dates and delays are equivalent for a single event, the definitions of functions *date* and *delay* are also equivalent. The use of delays instead of dates impacts other definitions as well. The following notions are equivalent to the ones in Chapter 2, only their formal definitions are adapted to the use of delays. Thereby, for a timed word $\sigma = (\delta_1, a_1) \cdot (\delta_2, a_2) \dots (\delta_n, a_n)$, we define:

- $\text{time}(\sigma) = \sum_{i=1}^n \delta_i$, for $\sigma \neq \epsilon$, and $\text{time}(\epsilon) = 0$;
- for $\delta \in \mathbb{R}_{\geq 0}$, $\sigma +_t \delta = (\delta_1 + \delta, a_1) \cdot (\delta_2, a_2) \dots (\delta_n, a_n)$;
- if $\delta_1 \geq \delta$, $\sigma -_t \delta = (\delta_1 - \delta, a_1) \cdot (\delta_2, a_2) \dots (\delta_n, a_n)$.

Note that some other definitions do not need to be adapted, since they only depend on an operator that has already been redefined. For instance, for $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, $\text{obs}(\sigma, t)$ is still defined as in Chapter 2, *i.e.* $\text{obs}(\sigma, t) = \min_{\preceq}(\{\sigma' \mid \sigma' \preceq \sigma \wedge \text{time}(\sigma') \leq t\})$, since it only depends on the operator $\text{time}(\sigma)$ that has already been modified to fit with the use of delays.

The restriction of a word to an alphabet must also be redefined. If $\sigma \in \text{tw}(\Sigma)$ and $\Sigma' \subseteq \Sigma$, then $\sigma|_{\Sigma'}$ is the word composed of the events of σ whose actions belong to Σ' , but with dates kept unchanged, not delays. Thus, one must compute the new delays to keep the dates unchanged when restricting a word to an alphabet. Formally, let us consider $\Sigma' \subseteq \Sigma$. Then, we define the restriction of a timed word in $\text{tw}(\Sigma)$ to an alphabet by induction as follows:

$$\epsilon|_{\Sigma'} = \epsilon$$

and, for $\sigma \in \text{tw}(\Sigma)$ and $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$,

$$(\sigma \cdot (\delta, a))|_{\Sigma'} = \begin{cases} \sigma|_{\Sigma'} \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma|_{\Sigma'}), a) & \text{if } a \in \Sigma' \\ \sigma|_{\Sigma'} & \text{otherwise.} \end{cases}$$

Note that to concatenate two restrictions, it is also needed to adjust the delay at the beginning of the second word: for $\sigma \in \text{tw}(\Sigma)$ and $\sigma' \in \text{tw}(\Sigma)$,

$$(\sigma \cdot \sigma')|_{\Sigma'} = \sigma|_{\Sigma'} \cdot (\sigma'|_{\Sigma'} +_t (\text{time}(\sigma) - \text{time}(\sigma|_{\Sigma'}))).$$

The notion of *delayed prefix* also needs to be adapted. As the restriction to an alphabet, this notion is defined with dates and not delays, thus for two timed words σ and σ' in $\text{tw}(\Sigma)$, $\sigma \preceq_d \sigma'$ whenever $\Pi_\Sigma(\sigma) \preceq \Pi_\Sigma(\sigma')$ and for any $i \in [1; |\sigma|]$, $\text{time}(\sigma_{[1..i]}) \geq \text{time}(\sigma'_{[1..i]})$. Again, note that the orders are not the same: σ is smaller than σ' , but its dates are greater than those of σ' .

For instance, if $\sigma = (1, a) \cdot (1, b) \cdot (2, a)$, then $\sigma|_{\{a\}} = (1, a) \cdot (3, a)$, and $(1, a) \cdot (2, b) \cdot (1, a) \preceq_d \sigma$.

4.1.2 Timed Automata

In this chapter, we use an alternative definition for the semantics of a timed automaton. The definition of a timed automaton is still the same as in Chapter 2. Let us consider a timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$. We define the semantics of \mathcal{A} as the timed transition system $\llbracket \mathcal{A} \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$, where Q , q_0 , and F_G are defined in the same way as in Definition 2.1 (*i.e.* $Q = L \times \mathcal{V}(X)$, $q_0 = (l_0, \nu[X \leftarrow 0])$, and $F_G = G \times \mathcal{V}(X)$). Unlike Definition 2.1, we define here Γ as $\Gamma = \mathbb{R}_{\geq 0} \cup \Sigma$, meaning that there are two types of transitions that define $\rightarrow \subseteq Q \times \Gamma \times Q$:

- *Delay transitions:* for $\delta \in \mathbb{R}_{\geq 0}$, $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$,
- *Action transitions:* for $a \in \Sigma$, $(l, \nu) \xrightarrow{a} (l', \nu')$, with $\nu' = \nu[Y \leftarrow 0]$ whenever there is a transition $(l, g, a, Y, l') \in \Delta$ such that $\nu \models g$.

The difference with Definition 2.1 is that the transition relation is made of delay transitions that correspond to letting time elapse, and action delays that correspond to the read of an action, whereas in Definition 2.1, a transition was made of a delay followed by an action.

We need to also redefine *runs* to fit with this new definition. Considering \mathcal{A} as defined previously, and its semantics $\langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$, a *run* ρ from $q \in Q$ is a valid sequence of transitions starting from q , *i.e.* $\rho = q \xrightarrow{\delta_1} q_1 \xrightarrow{a_1} q_2 \xrightarrow{\delta_2} q_3 \dots \xrightarrow{a_n} q_{2n} \xrightarrow{\delta_{n+1}} q_{2n+1}$, where $\delta_i \in \mathbb{R}_{\geq 0}$ and $a_i \in \Sigma$, for any i . We can consider runs that alternate between delay and action transitions, since two consecutive delay transitions can be merged into one whose value is the sum of the delays of the original transitions, and two consecutive actions can be separated by a null delay transition (*i.e.* a delay transition whose delay is 0). We can also consider only runs that begin and end by a delay transition, adding some null delay transitions if necessary.

The *trace* of the run ρ previously defined is the timed word $\sigma = (\delta_1, a_1) \cdot (\delta_2, a_2) \dots (\delta_n, a_n)$. Note that δ_{n+1} does not appear in the trace, meaning that all runs with different values for δ_{n+1} share the same trace as ρ . We allow ourselves to denote by $q \xrightarrow{(\delta, a)} q'$ if $q \xrightarrow{\delta} q'' \xrightarrow{a} q'$, and thus ρ can be denoted $q \xrightarrow{\sigma} q_{2n} \xrightarrow{\delta_{n+1}} q_{2n+1}$.

Note that with this definition of \rightarrow , the definition of after with two parameters becomes more straightforward: $q \text{ after } (\sigma, t) = q'$, where $q \xrightarrow{\text{obs}(\sigma, t)} q'' \xrightarrow{t - \text{time}(\text{obs}(\sigma, t))} q'$. Moreover, it is now possible to write, for $q \in Q$, $q' \in Q$, $a \in \Sigma$ and $\delta \in \mathbb{R}_{\geq 0}$, $q' = q \text{ after } a$ if $q \xrightarrow{a} q'$, and $q' = q \text{ after } \delta$ if $q \xrightarrow{\delta} q'$.

4.1.3 Enforcement Functions

In this chapter, we use a set representation for enforcement functions. This representation is equivalent to the functional representation used in Chapter 3, such that the set representation of function $f : x \mapsto f(x)$ is the set $\{(x, f(x)) \mid x \in \text{domain}(f)\}$, where $\text{domain}(f)$ represents the domain of function f .

4.2 Enforcing Untimed Properties

This section is similar to Section 3.1: its purpose is to define enforcement mechanisms that are sound, compliant and optimal. The main difference between this section and Section 3.1 is the use of Büchi games to compute the set of “safe” states. The interest is a very practical one: using Büchi games allows us to compute “safe” states before the execution. This precomputation allows us to reduce the overhead introduced by the enforcement mechanism at runtime. Note that the interest of this approach might be limited in the untimed setting, but it can greatly improve the performance at runtime in the timed setting (see Section 4.3).

In this section, φ is a regular property defined by a complete and deterministic automaton $\mathcal{A}_\varphi = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$. As in Chapter 3, we give definitions of enforcement functions, soundness, compliance, and optimality. These definitions are equivalent to the corresponding definitions in Section 3.1, but use a set approach to functions.

Again, we consider uncontrollable events in the set $\Sigma_u \subseteq \Sigma$, and controllable events in $\Sigma_c = \Sigma \setminus \Sigma_u$. This notion of uncontrollable and controllable events must not be confused with the notion of controllable events from game theory. Even though we use some games to compute safe states, we use “uncontrollable events” only to denote events in Σ_u and “controllable events” to denote events in Σ_c . The primitives we chose to use for our enforcement mechanisms are the same as in Chapter 3, *i.e.* an EM can only delay controllable events, but not suppress them. Uncontrollable events are immediately emitted upon reception.

Thus, an EM may interleave controllable and uncontrollable events.

4.2.1 Enforcement Functions and their Requirements

We consider an alphabet of actions Σ . We consider functions as sets: a *function* from a set A to a set B is a set $f \subseteq A \times B$ such that for any element a in A , there is a unique b in B such that $(a, b) \in f$. We note $\mathcal{F}(A, B)$ the set of all functions from A to B .

An enforcement function is a description of the input/output behaviour of an EM. It is a function from Σ^* to Σ^* , increasing on Σ^* (with respect to \preceq):

Definition 4.1 (Enforcement function). A function $f \in \mathcal{F}(\Sigma^*, \Sigma^*)$ is an *enforcement function* (over Σ) if:

$$\forall i_1 \in \Sigma^*, \forall i_2 \in \Sigma^*, (i_1 \preceq i_2 \wedge (i_1, o_1) \in f \wedge (i_2, o_2) \in f) \implies o_1 \preceq o_2.$$

We note $\mathcal{F}_{\text{enf}}(\Sigma)$ the set of all enforcement functions over the alphabet Σ . When clear from the context, the parameter shall be omitted, *i.e.* \mathcal{F}_{enf} is used to designate the set of enforcement functions over Σ .

An enforcement function is a function that modifies an execution, and that cannot remove events it has already output.

As in Section 3.1.1, we provide definitions of soundness, compliance and optimality, that are the requirements expected from EMs, and express them as constraints on enforcement functions. An enforcement function should be *sound*, meaning that its output should satisfy the property:

Definition 4.2 (Soundness). An enforcement function $E \in \mathcal{F}_{\text{enf}}$ is *sound* with respect to φ in an extension-closed set $S \subseteq \Sigma^*$ if:

$$\forall i \in S, (i, o) \in f \implies o \models \varphi.$$

We note $\mathcal{F}_{\text{snd}}(\varphi, S)$ (or $\mathcal{F}_{\text{snd}}(S)$ when clear from the context) the set of all enforcement functions that are sound with respect to φ in S .

As for Definition 3.2, the property should be satisfied by the output of the enforcement function only in a subset of Σ^* , due to the potential impossibility to correct the input word into a valid one from the beginning. For example, considering property φ_{ex} (see Fig. 3.2), word *LockOff* can not be corrected into a valid one (since it is uncontrollable, receiving it leads to q_3 , which is not accepting). The set S is required to be extension-closed to ensure that the property is always satisfied once the enforcement mechanism is effective.

Remember that compliance defines how the EM can modify the input execution. We only allow to delay controllable events (they can be delayed indefinitely), uncontrollable events must be output immediately upon reception. EMs can output several stored controllable events at the same time

(keeping the order unchanged), *i.e.* without receiving any event, controllable or uncontrollable. Nevertheless, they can not output such events before an uncontrollable event after having received it.

Definition 4.3 (Compliance). $E \in \mathcal{F}_{\text{enf}}$ is *compliant* with respect to Σ_u and Σ_c , noted $\text{compliant}(E, \Sigma_u, \Sigma_c)$, if:

$$\begin{aligned} \forall i \in \Sigma^*, (i, o) \in E \implies \\ (o \preceq_{\Sigma_c} i \wedge o =_{\Sigma_u} i \wedge \forall u \in \Sigma_u, ((i \cdot u, o') \in E \implies o \cdot u \preceq o')). \end{aligned}$$

We note $\mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ the set of all enforcement functions (over $\Sigma = \Sigma_u \cup \Sigma_c$) that are compliant with respect to Σ_u and Σ_c . When clear from the context, we can denote it by \mathcal{F}_{cpl} , and $\text{compliant}(E, \Sigma_u, \Sigma_c)$ is simply noted $\text{compliant}(E)$.

Intuitively, compliance states that the EM does not change the order of the controllable events and emits uncontrollable events simultaneously with their reception, possibly followed by stored controllable events.

Moreover, an enforcement function should output the maximum number of events it possibly can. Thus, we define the optimality of sound and compliant enforcement functions as follows:

Definition 4.4 (Optimality). A sound and compliant enforcement function $E \in \mathcal{F}_{\text{snd}}(S) \cap \mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ is *optimal* in S if:

$$\begin{aligned} \forall E' \in \mathcal{F}_{\text{snd}}(S) \cap \mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c), \forall i \in S, \forall a \in \Sigma, \\ ((i, o) \in E \cap E' \wedge (i \cdot a, o') \in E \wedge (i \cdot a, p') \in E') \implies p' \preceq o'. \end{aligned}$$

Intuitively, optimality states that outputting a longer word than an optimal enforcement function breaks soundness or compliance. Since it is not always possible to satisfy the property from the beginning, this condition is restricted to an extension-closed subset of Σ^* , as in the definition of soundness (see Definition 4.2). An example has been given in Chapter 3, see Example 3.1.

Now that we have defined the input/output behaviour of enforcement mechanisms as enforcement functions, and expressed the requirements we expect of enforcement mechanisms, we can define an enforcement mechanism as an enforcement function that is sound, compliant, and optimal.

4.2.2 Synthesising Enforcement Functions

As in Section 3.1.3, in this section we redefine G , store_φ , E_φ and $\text{Pre}(\varphi)$ to fit with the set representation of functions. Functions S and I are replaced by the use of a Büchi game to compute “safe” states. G is adapted to use the Büchi game instead of S and I , but the definitions of store_φ , E_φ and $\text{Pre}(\varphi)$, that deeply depend on G , are quite similar to the ones in Section 3.1.3.

Function G , as in Section 3.1.3, gives the set of controllable sequences that can be output by a sound and compliant enforcement function for a given state and buffer. To define it, in this chapter, we solve a Büchi game over a graph representing the possible actions of an enforcement monitor. Solving a Büchi game is made by computing a set of nodes of the graph from which there exists a winning strategy for the chosen player. Then, from any of these winning nodes, this player can always come back to a Büchi node, whatever the strategy of the adversary is. Here, we construct a graph such that the enforcement mechanism is a player (the other player being the environment that feeds the events of the input to the EM), and we compute its winning nodes, with the Büchi nodes representing a valid execution. The nodes of the graph are composed of a state in Q and the stored controllable events of the enforcement mechanism. There exists two of each of these vertices: one that belongs to player P_0 , and one that belongs to player P_1 . Player P_0 represents the enforcement mechanism, and P_1 the environment.

Definition 4.5 (Game graph). The game graph \mathcal{G} is defined as $\mathcal{G} = \langle V, E \rangle$, where

- $V = Q \times \Sigma_c^* \times \{0, 1\}$,
- $E = \bigcup_{i=1}^5 E_i$, with:
 - $E_1 = \{(\langle q, w, 0 \rangle, \langle q, w, 1 \rangle) \in V \times V\}$,
 - $E_2 = \{(\langle q, c.w, 0 \rangle, \langle q \text{ after } c, w, 0 \rangle) \in V \times V \mid c \in \Sigma_c\}$,
 - $E_3 = \{(\langle q, w, 1 \rangle, \langle q \text{ after } u, w, 0 \rangle) \in V \times V \mid u \in \Sigma_u\}$,
 - $E_4 = \{(\langle q, w, 1 \rangle, \langle q, w.c, 0 \rangle) \in V \times V \mid c \in \Sigma_c\}$,
 - $E_5 = \{(\langle q, w, 1 \rangle, \langle q, w, 0 \rangle) \in V \times V\}$,

A vertex $\langle q, w, l \rangle \in V$ represents the state of the enforcement mechanism: $q \in Q$ is the state of \mathcal{A}_φ that has been reached so far by the output of the enforcement mechanism, $w \in \Sigma_c^*$ is the word made of the stored controllable events of the enforcement mechanism, and $l \in \{0, 1\}$ indicates that the vertex belongs to the player P_l . The definition of E is split into five sets, each one containing a different kind of transitions. The enforcement mechanism can only take two decisions: doing nothing, *i.e.* letting the environment play (set E_1), or emitting the first stored controllable event (set E_2), in which case it continues to play (since the destinations of the edges in E_2 belong to P_0). The sets E_3 and E_4 represent the reception of an uncontrollable and a controllable event, respectively. Receiving an event lets the enforcement mechanism (P_0) play. Since games are infinite, and we only consider finite executions, the environment can also decide to let the enforcement mechanism play without any new event (set E_5). This allows us to consider finite executions that

produce an infinite path in the game by looping on an edge in E_1 and then one in E_5 . It is also possible to consider that this corresponds to receiving an empty event (ϵ), and that player P_1 (the environment) feeds an infinite input, which is the finite one with infinitely many empty events appended.

Unfortunately, this graph has an infinite number of nodes, it is thus not possible to compute the set of winning vertices for a Büchi game over it. To overcome this, the graph is reduced to a graph with a finite number of vertices. To do this, first note that the number of vertices is infinite because the set Σ_c^* is not bounded. Thus, Σ_c^* must be abstracted to a finite set. Since the goal is to reach a state in F , the stored controllable events are used to reach some states in Q . Since Q is finite, having more controllable events than $|Q|$ means that (following the Pumping lemma) there is a loop, *i.e.* some state in Q is reached twice when emitting all the controllable events. This means that all the states that are reachable from a given state can be reached by a word of size at most $|Q|$. Thus, the number of controllable events to consider can be reduced to at most $|Q|$, since all words of size less than $|Q|$ allow to reach all the reachable states from a state. More precisely, we can reduce Σ_c^* to the set of words that allow to reach a new state (*i.e.* a state that is not reached by one of its prefixes) from at least one state in Q . Let us call this set Σ_c^n , and define it as follows:

Definition 4.6 (Σ_c^n).

$$\Sigma_c^n = \{w \in \Sigma_c^* \mid \exists q \in Q, \exists c \in \Sigma_c, \forall w' \preceq w, q \text{ after } w \cdot c \neq q \text{ after } w'\}$$



As explained previously, since Q is finite, Σ_c^n is finite as well. Now, let us redefine \mathcal{G} to an abstraction of the game graph:

Definition 4.7 (Abstracted game graph). $\mathcal{G} = \langle V', E' \rangle$, where $V' = Q \times \Sigma_c^n \times \{0, 1\}$, and E' is the same set as E , but considering vertices in V' instead of V .

\mathcal{G} is the restriction of the previous graph to a finite number of vertices. Let us now consider $W_0 \subseteq V$ the set of vertices that are winning for P_0 in the Büchi game over \mathcal{G} , with the set of Büchi nodes $F \times \Sigma_c^n \times \{0, 1\}$.

Example 4.1. The graph in Fig. 4.1 is computed from property φ_{ex} (see Fig. 3.2), with **Write** abbreviated **w** in the second member of the nodes. The Büchi nodes are double circled, and the winning nodes for player P_0 (the EM), *i.e.* nodes in W_0 , are in blue and rounded rectangles.

Each edge has a different colour and a different head depending on the set it belongs to:

- blue edges, with empty triangular head () belong to E_1 ,
- green edges, with filled triangular head () belong to E_2 ,

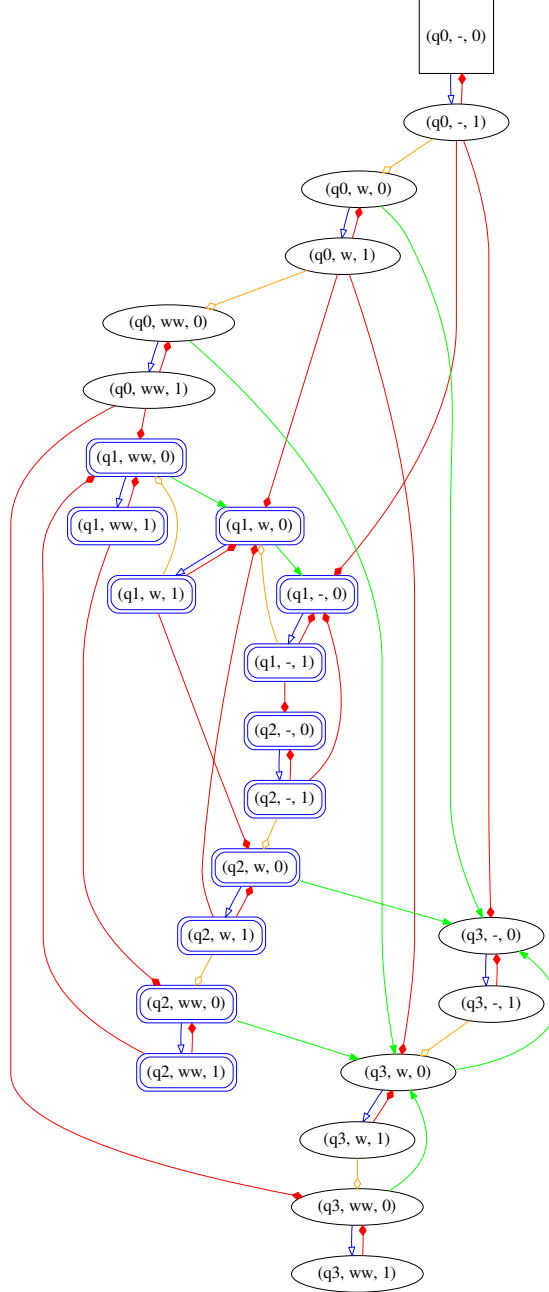


Figure 4.1 – Graph of the game associated to φ_{ex}

- orange edges, with empty diamond head ($\text{---}\diamond$) belong to E_4 ,
- red edges with filled diamond head ($\text{---}\blacklozenge$) belong to $E_3 \cup E_5$.

Each edge is represented only once, even if there are multiple edges in the set (for example, because multiple uncontrollable events lead to the same state from one state). The squared vertex is the initial vertex, and “—” stands for “ ϵ ” (empty buffer). Since the initial vertex is black (not rounded), this means that it is impossible to ensure that the property will be satisfied from the beginning. The only way to reach a winning state is to follow a red edge from a vertex in $\{q_0\} \times \{\epsilon, \mathbf{w}, \mathbf{w}.\mathbf{w}\} \times \{1\}$, that corresponds to receiving the uncontrollable event *Auth* (since it leads to a state in $\{q_2\} \times \{\epsilon, \mathbf{w}, \mathbf{w}.\mathbf{w}\} \times \{0\}$). Then, *Write* events can only be emitted when in state q_1 . This behaviour is the one expected, since in φ_{ex} , the only way to reach a state in F from q_0 is to follow a path labelled by *Auth*, and then q_1 is reached, from which it is possible to emit *Write* events, but if some uncontrollable events are received that lead to q_2 , one must wait an event *LockOff* to go back to q_1 and be able to emit another *Write* event.

Now, we can use W_0 to define G , the set of words that can be emitted from a state $q \in Q$ by an enforcement mechanism with a buffer $\sigma \in \Sigma_c^*$.

Definition 4.8 (G). For a state $q \in Q$ and a word of controllable events $\sigma \in \Sigma_c^*$, we define the set $G(q, \sigma)$ as follows:

$$G(q, \sigma) = \{w \in \Sigma_c^* \mid w \preceq \sigma \wedge q \text{ after } w \in F \wedge \langle q \text{ after } w, \max_{\preceq}(\{w' \preceq w^{-1} \cdot \sigma \mid w' \in \Sigma_c^n\}), 1 \rangle \in W_0\}.$$

Intuitively, G is the set of words that can be output by a compliant enforcement mechanism to ensure soundness.

Now, we use G to define the functional behaviour of the enforcement mechanism.

Definition 4.9 (Functions store_φ , E_φ). Function $\text{store}_\varphi \in \Sigma^* \times (\Sigma^* \times \Sigma_c^*)$ is defined by induction on its first member as follows:

$$(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi,$$

and, for $\sigma \in \Sigma^*$ and $a \in \Sigma$, if $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, then,

$$\begin{cases} (\sigma \cdot a, \langle \sigma_s \cdot a \cdot \sigma'_s, \sigma'_c \rangle) \in \text{store}_\varphi & \text{if } a \in \Sigma_u \\ (\sigma \cdot a, \langle \sigma_s \cdot \sigma''_s, \sigma''_c \rangle) \in \text{store}_\varphi & \text{if } a \in \Sigma_c, \end{cases}$$

where, for $q \in Q$ and $w \in \Sigma_c^*$,

$$\kappa_\varphi(q, w) = \max_{\preceq}(G(q, w) \cup \{\epsilon\}),$$

and

$$\begin{aligned}\sigma'_s &= \kappa_\varphi(\text{Reach}(\sigma_s \cdot a), \sigma_c) & \sigma'_c &= \sigma'^{-1}_s \cdot \sigma_c \\ \sigma''_s &= \kappa_\varphi(\text{Reach}(\sigma_s), \sigma_c \cdot a) & \sigma''_c &= \sigma''^{-1}_s \cdot (\sigma_c \cdot a).\end{aligned}$$

The enforcement function $E_\varphi \in \mathcal{F}_{\text{enf}}$ is then defined as:

$$E_\varphi = \{(\sigma, \sigma') \mid \exists w \in \Sigma_c^*, (\sigma, \langle \sigma', w \rangle) \in \text{store}_\varphi\}.$$

Intuitively, σ_s is the word that can be released as output, whereas σ_c is the buffer containing the events that are already read/received, but cannot be released as output yet because they lead to an unsafe state from which it would be possible to violate the property reading only uncontrollable events (*i.e.* they lead to a vertex in $W_1 = V \setminus W_0$). Upon receiving a new event a , the enforcement mechanism distinguishes two cases:

- If a belongs to Σ_u , then it is output, as required by compliance. Then, the longest prefix of σ_c that satisfies φ and leads to a vertex in W_0 is also output.
- If a is in Σ_c , then it is added to σ_c , and the longest prefix of this new buffer that satisfies φ and leads to a vertex in W_0 is emitted, if it exists.

In both cases, κ_φ is used to compute the longest word that can be output, that is the longest word in G for the state reached so far and the current buffer of the enforcement mechanism, or ϵ if this set is empty. The parameters of κ_φ are those which are passed to G , they correspond to the state reached so far by the output of the enforcement mechanism, and its current buffer, respectively.

Remember that some properties are not enforceable (see Example 3.1), but receiving some events may lead to a state from which it is possible to enforce. Therefore, it is possible to define the set of words $\text{Pre}(\varphi)$, such that E_φ is sound in $\text{Pre}(\varphi)$, as stated in Proposition 4.2:

Definition 4.10 (Pre). The set of input words $\text{Pre}(\varphi) \subseteq \Sigma^*$ is defined as follows:

$$\text{Pre}(\varphi) = \{\sigma \in \Sigma^* \mid G(\text{Reach}(\sigma|_{\Sigma_u}), \sigma|_{\Sigma_c}) \neq \emptyset\} \cdot \Sigma_c^*$$

Again, the definition of $\text{Pre}(\varphi)$ in Definition 4.10 is the same as in Definition 3.11, since it only depends on G , whose definition has been changed, but is equivalent to the one in Section 3.1. In E_φ , using W_0 ensures that once the set G is not empty, then it will never be afterwards, whatever events are received. Thus, $\text{Pre}(\varphi)$ is the set of input words such that the output of E_φ belongs to G . Since E_φ outputs only uncontrollable events until G becomes non-empty, the definition of $\text{Pre}(\varphi)$ considers that the state reached is the one that is reached by emitting only the uncontrollable events of σ , and the corresponding buffer would then be the controllable events of σ . Thus, $\text{Pre}(\varphi)$ is the set in which E_φ is sound.

Example 4.2. Considering property φ_{ex} as shown in Fig. 3.2, with the uncontrollable alphabet $\Sigma_u = \{\text{Auth}, \text{LockOff}, \text{LockOn}\}$, $\text{Pre}(\varphi_{\text{ex}}) = \text{Write}^* . \text{Auth} . \Sigma^*$. Indeed, from the initial state q_0 , if an uncontrollable event, say LockOff , is received, then q_3 is reached, which is a non-accepting sink state, and thus any vertex in $\{q_3\} \times \Sigma_c^n \times \{0, 1\}$ will not be in W_0 . In order to reach a vertex in W_0 (i.e. a vertex in $\{q_1, q_2\} \times \Sigma_c^n \times \{0, 1\}$), it is necessary to read Auth . Once Auth is read, q_1 is reached, and from there, all uncontrollable events lead to either q_1 or q_2 . The same holds true from q_2 . Thus, it is possible to stay in the accepting states q_1 and q_2 , by delaying Write events when in q_2 until a LockOff event is received. Consequently, $\{q_1, q_2\} \times \Sigma_c^n \times \{0, 1\} \subseteq W_0$, and thus $\text{Pre}(\varphi_{\text{ex}}) = \text{Write}^* . \text{Auth} . \Sigma^*$, since Write events can be buffered while in state q_0 until event Auth is received, leading to a vertex in $\{q_1\} \times (\text{Write}^* \cap \Sigma_c^n) \times \{0, 1\} \subseteq W_0$.

Function store_φ as per Definition 4.9 is equivalent to store_φ as per Definition 3.10, thus the evolution of σ_s and σ_c , such that $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi_{\text{ex}}}(\sigma)$ for $\sigma = \text{Auth} . \text{LockOn} . \text{Write} . \text{LockOff}$ and all its prefixes can be found in Table 3.2.

E_φ (as per Definition 4.9) is an enforcement function that is sound with respect to φ in $\text{Pre}(\varphi)$, compliant with respect to Σ_u and Σ_c , and optimal in $\text{Pre}(\varphi)$.

Proposition 4.1. E_φ as per Definition 4.9 is an enforcement function as per Definition 4.1.

Sketch of proof. We have to show that for all σ and σ' in Σ^* , $(\sigma, \sigma_o) \in E_\varphi \wedge (\sigma . \sigma', \sigma'_o) \in E_\varphi \implies \sigma_o \preceq \sigma'_o$. Following the definition of store_φ , this holds provided that $\sigma' \in \Sigma$ (i.e. σ' is a word of size 1). Since \preceq is an order, it follows that the proposition holds for all $\sigma' \in \Sigma'$.

Proposition 4.2. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$, as per Definition 4.2.

Sketch of proof. We have to show that if $\sigma \in \text{Pre}(\varphi)$, then $(\sigma, \sigma_o) \in E_\varphi \implies \sigma_o \models \varphi$. The proof is made by induction on σ . In the induction step, considering $a \in \Sigma$, we distinguish three cases:

1. $\sigma . a \notin \text{Pre}(\varphi)$. Then the proposition holds.
2. $\sigma . a \in \text{Pre}(\varphi)$, but $\sigma \notin \text{Pre}(\varphi)$. Then the input reaches $\text{Pre}(\varphi)$, and since it is extension-closed, all extensions of σ also are in $\text{Pre}(\varphi)$, and we prove that the proposition holds considering the definition of $\text{Pre}(\varphi)$.
3. $\sigma \in \text{Pre}(\varphi)$ (and thus, $\sigma . a \in \text{Pre}(\varphi)$ since it is extension-closed). Then, we prove that the proposition holds, based on the definition of store_φ ,

and more precisely on the definition of G , that uses W_0 to ensure that there always exists a compliant output that satisfies φ .

Proposition 4.3. E_φ is compliant, as per Definition 4.3.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. Considering $\sigma \in \Sigma^*$ and $a \in \Sigma$, the proof is straightforward by considering the different values of $(\sigma.a, \sigma_o) \in \text{store}_\varphi$, $(\sigma.a)_{|\Sigma_u}$, and $(\sigma.a)_{|\Sigma_c}$, when $a \in \Sigma_c$ and $a \in \Sigma_u$.

For any given input $\sigma \in \text{Pre}(\varphi)$, $E_\varphi(\sigma)$ is the longest possible word that ensures soundness and compliance, that is controllable events are blocked only when necessary. Thus, E_φ is also optimal in $\text{Pre}(\varphi)$:

Proposition 4.4. E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 4.4.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. Once $\sigma \in \text{Pre}(\varphi)$, we know that $(\sigma, \sigma_o) \in E_\varphi \implies \sigma_o \models \varphi$ since E_φ is sound in $\text{Pre}(\varphi)$. E_φ is optimal because, in store_φ , κ_φ provides the longest possible word. If a longer word were output, then either the output would not satisfy φ , or it would lead to a vertex that is not in W_0 , meaning that there would exist an uncontrollable word leading to a non-accepting state and to a vertex that would not be in W_0 . Then, the enforcement mechanism would have to output some controllable events from the buffer to reach an accepting state, but since the vertex is not in W_0 , there would exist again an uncontrollable word leading to a non-accepting state and a vertex not in W_0 . By iterating, the buffer would become ϵ whereas the output of the enforcement mechanism would be leading to a non-accepting state. Therefore, outputting a longer word would mean that the function is not sound. This means that E_φ is optimal in $\text{Pre}(\varphi)$, since it outputs the longest word that allows us to be both sound and compliant.

4.2.3 Enforcement Monitors

We can describe enforcement monitors as in Section 3.1.4, representing an enforcement monitor as a transition system. Since the enforcement monitor described in Definition 3.12 does not depend on I and S (see Definition 3.8), but only on G , the definition of a monitor using a Büchi game is the same as Definition 3.12. In other words, the use of a Büchi game is transparent to the operational monitor, because it is hidden in the use of G .

This leads us to the following proposition:

Proposition 4.5. *The output o of the enforcement monitor \mathcal{E} as per Definition 3.12 for input σ is the output of E_φ as per Definition 4.9 with input σ , i.e. $(\sigma, o) \in E_\varphi$.*

In Proposition 4.5, the output of the enforcement monitor is the concatenation of all the outputs of the word labelling the path followed when reading σ . A more formal definition is given in the proof of this proposition, in appendix A.2.1.

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. We just consider the rules that can be applied when receiving a new event. If the event is controllable, then rule `store-cont()` can be applied, possibly followed by rule `dump()` applied once or more times. If the event is uncontrollable, then rule `pass-uncont()` can be applied, again possibly followed by rule `dump()` applied once or more times. Since rule `dump()` applies only when there is a non-empty word in G , then this word must begin with the first event of the buffer, and rule `dump()` can be applied again if there was a word in G of size at least 2, meaning that there is another non-empty word in the new set G . This can be applied n times, where n is the length of the longest word in G at the beginning. Thus, the output of all the applications of rule `dump()` corresponds to the computation of κ_φ in the definition of store_φ , and consequently the outputs of \mathcal{E} and E_φ are the same.

Remark 4. For a configuration $c = \langle q, w \rangle$ of the enforcement monitor, we can consider the node of the game graph $(q, w', 0)$, with w' the longest prefix of w such that (q, w') is a node of the game graph. Then, if $\sigma \in \text{Pre}(\varphi)$, and c is the configuration reached by the enforcement monitor with input σ , then $(q, w', 0)$ is a winning node.

4.3 Enforcing Timed Properties

In this section, we extend the framework presented in Section 4.2 to enforce timed properties. As in Section 3.2, enforcement mechanisms and their properties need to be redefined to fit with timed properties. Enforcement functions take an observation time, and soundness is defined to enforce “eventually always φ ” instead of the property φ itself. This gives more flexibility to enforcement mechanisms, allowing them to less modify the input while ensuring that the property will hold in the future.

Remember that, unlike in Section 3.2, delays are used instead of dates in timed words, all across this section. All notation changes of this section are listed in Section 4.1.

In this section, φ is a timed property defined by a timed automaton $\mathcal{A}_\varphi = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ with semantics $\llbracket \mathcal{A}_\varphi \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$.

4.3.1 Enforcement Functions and their Properties

We adapt the definitions of *enforcement functions*, *soundness*, *compliance*, and *optimality* to fit with timed properties. The definitions in this section are equivalent to the ones in Section 3.2.1, but use a set representation of functions, and delays instead of dates.

An enforcement function takes a timed word and the current time as input, and outputs a timed word:

Definition 4.11 (Enforcement Function). Given an alphabet of actions Σ , an *enforcement function* (over Σ) is a function $E \in \mathcal{F}(\text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}, \text{tw}(\Sigma))$ that satisfies the following constraints:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t,$
 $\langle (\sigma, t), o_1 \rangle \in E \wedge \langle (\sigma, t'), o_2 \rangle \in E \implies o_1 \preceq o_2$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall \delta \in \mathbb{R}_{\geq 0}, \forall a \in \Sigma,$
 $((\langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o_3 \rangle \in E \wedge \langle (\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))), o_4 \rangle \in E)$
 $\implies o_3 \preceq o_4$

As in Section 4.2, we note $\mathcal{F}_{\text{enf}}(\Sigma)$ (or \mathcal{F}_{enf} when clear from the context) the set of all enforcement functions over Σ . Be aware that \mathcal{F}_{enf} in Section 4.2 is different from \mathcal{F}_{enf} as per Definition 4.11, since the current section is about timed enforcement functions.

The requirements in Definition 4.11 model physical constraints: an enforcement function can only add events to its output as the input grows. The first constraint ($o_1 \preceq o_2$) corresponds to letting time elapse, whereas the second one ($o_3 \preceq o_4$) corresponds to reading a new event. They both require the new output to be an extension of the previous one.

Soundness states that the output of an enforcement function should eventually always satisfy the desired property:

Definition 4.12 (Soundness). An enforcement function $E \in \mathcal{F}_{\text{enf}}$ is *sound* with respect to φ in a time-extension-closed set $S \subseteq \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ if:

$$\forall (\sigma, t) \in S, \exists t' \geq t, \forall t'' \geq t', ((\sigma, t''), o) \in E \implies o \models \varphi.$$

We note $\mathcal{F}_{\text{snd}}(\varphi, S)$ the set of all enforcement functions that are sound with respect to φ in S .

This definition is equivalent to Definition 3.14, thus an enforcement function is sound with respect to φ in S if for any $(\sigma, t) \in S$, the output of the enforcement function with input σ from date t eventually always satisfies S . Again, S restrains $\text{tw}(\Sigma)$ because for some properties, some words can not be corrected to satisfy the property.

As usual, *compliance* states that an enforcement mechanism can only delay controllable events:

Definition 4.13 (Compliance). An enforcement function $E \in \mathcal{F}_{\text{enf}}$ is *compliant* with respect to Σ_u and Σ_c , noted $\text{compliant}(E, \Sigma_u, \Sigma_c)$ (or $\text{compliant}(E)$ when clear from the context), if it satisfies the following constraints:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \langle (\sigma, t), o_1 \rangle \in E \implies o_1 \preceq_{\text{d}\Sigma_c} \text{obs}(\sigma, t)$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \langle (\sigma, t), o_2 \rangle \in E \implies o_2 =_{\Sigma_u} \text{obs}(\sigma, t)$
3. $\forall \sigma \in \text{tw}(\Sigma), \forall (\delta, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u,$
 $\langle (\sigma, \text{time}(\sigma \cdot (\delta, u))), o_3 \rangle \in E \wedge \langle (\sigma \cdot (\delta, u), \text{time}(\sigma \cdot (\delta, u))), o_4 \rangle \in E$
 $\implies o_3 \cdot (\text{time}(\sigma \cdot (\delta, u)) - \text{time}(o_3), u) \preceq o_4.$

We note $\mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ (or \mathcal{F}_{cpl} when clear from the context) the set of all enforcement functions that are compliant with respect to Σ_u and Σ_c .

The definition of compliance as per Definition 4.13 is equivalent to the one as per Definition 3.15. The three constraints are equivalent, they are only adapted in Definition 4.13 to fit with the use of the set representation of enforcement functions and delays instead of dates. The first constraint requires that an EM can only delay controllable events, without changing their order; the second constraint requires that uncontrollable events are not modified; and the third constraint requires that the enforcement mechanism does not react to the reception of an uncontrollable event before outputting it.

For a compliant enforcement function $E \in \mathcal{F}_{\text{enf}}$ and a timed word $\sigma \in \text{tw}(\Sigma)$, we say that $((\sigma, \infty), o) \in E$ if o is the output of E with input σ at infinite time (*i.e.* when it has stabilised). More formally, $((\sigma, \infty), o) \in E \iff \exists t \in \mathbb{R}_{\geq 0}, \forall t' \geq t, ((\sigma, t'), o) \in E$. Since σ is finite, and E is compliant, the output of E with input word σ is finite, thus such an output o exists.

As in Section 3.2.1, we define a notion of *optimality* in a set:

Definition 4.14 (Optimality). A sound and compliant enforcement function $E \in \mathcal{F}_{\text{snd}}(\varphi, S) \cap \mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ is *optimal* in S if:

$$\begin{aligned} & \forall E' \in \mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c), \forall \sigma \in \text{tw}(\Sigma), \forall (\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma, \\ & (\sigma, \text{time}(\sigma \cdot (\delta, a))) \in S \wedge \langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o \rangle \in E \cap E' \wedge \\ & \langle (\sigma \cdot (\delta, a), \infty), o_1 \rangle \in E \wedge \langle (\sigma \cdot (\delta, a), \infty), o'_1 \rangle \in E' \wedge o_1 \prec_d o'_1 \\ & \implies (\exists \sigma_u \in \text{tw}(\Sigma_u), \langle (\sigma \cdot (\delta, a) \cdot \sigma_u, \infty), o_u \rangle \in E' \wedge o_u \not\models \varphi). \end{aligned}$$

Optimality, as per Definition 4.14, is equivalent to optimality as per Definition 3.16. Intuitively, a sound and compliant enforcement function is optimal if at any moment, it outputs the longest possible word, with lower delays, ensuring soundness and compliance. In Definition 4.14, we suppose that a compliant enforcement function outputs a greater word (with respect to \preceq_d) than an optimal one, and then conclude that it is not sound (since the other function is optimal).

Now that all the requirements on enforcement functions have been redefined, we can redefine the enforcement function E_φ so that its output is the same as the one as per Definition 3.19. The difference with Section 3.2.2 is that we use a Büchi game to compute the safe states.

4.3.2 Synthesising Timed Enforcement Functions

In this section, we redefine G , store_φ , E_φ and Pre such that E_φ is an enforcement function that is sound with respect to φ in $\text{Pre}(\varphi)$, compliant with respect to Σ_u and Σ_c , and optimal in $\text{Pre}(\varphi)$. Thus, the output of E_φ is expected to be the same as in Section 3.2.2. The difference with Section 3.2.2 is that we define G using a Büchi game. Remember that function G gives, for a state and a sequence of stored controllable events, the set of timed words that can be output by a sound and compliant enforcement mechanism. To compute such words, we construct a graph on which we play a Büchi game.

The graph is defined in a very similar way to the one used in the untimed setting (Section 4.2.2). The nodes of the graph should be taken in the set $Q \times \Sigma_c^* \times \{0, 1\}$. Considering all such nodes, the graph would have an infinite number of nodes, first because the number of stored controllable events is not bounded, but also because the semantics of a timed automaton has itself an infinite number of states (*i.e.* Q is also infinite). We can use the same set Σ_c^n as in the untimed setting (see Definition 4.6), adapted to the timed setting, to restrict the number of stored controllable events to be considered. Intuitively, since the validity of a sequence only depends on the location that is reached after reading it, Σ_c^n is composed of all the controllable actions that can allow the enforcement mechanism to reach a new location. Then, we define Σ_c^n as follows:

Definition 4.15 (Σ_c^n).

$$\begin{aligned} \Sigma_c^n = \{w \in \Sigma_c^* \mid \exists q \in Q, \exists c \in \Sigma_c, \forall \sigma \in \text{tw}(\Sigma), \forall \sigma' \in \text{tw}(\Sigma), \\ \Pi_\Sigma(\sigma) = w . c \wedge \Pi_\Sigma(\sigma') \preceq w \wedge (l', \nu') = q \text{ after } \sigma \wedge (l'', \nu'') = q \text{ after } \sigma' \\ \implies l' \neq l''\} \end{aligned}$$

Since in Definition 4.15, Σ_c^n is defined as the set of controllable sequences of actions that can be used to form a word that allows to reach a new location, the length of a word in Σ_c^n can not be greater than the number of locations in the timed automaton. Thus, Σ_c^n is finite since L is finite.

As mentioned previously, the graph that we would naturally want to use is infinite because of two infinite components: Q , and Σ_c^* . We can reduce the number of nodes by restricting Σ_c^* to the finite set Σ_c^n . Now, we also need to reduce Q to a finite set, *i.e.* we need to consider an abstraction of time. In other words, we need to use a symbolic abstraction of the semantics of \mathcal{A}_φ .

A Symbolic Graph

Several abstractions for timed automata exist to reduce their semantics to a finite representation. The simplest, that satisfies all the requirements we need, is the region graph (see [Alur and Dill \[1992\]](#)) of the timed automaton. Unfortunately, this region graph is often very large, thus some more efficient abstractions have been studied. A very common abstraction is the zone graph used to compute reachability in a timed automaton ([Bengtsson and Yi \[2004\]](#)). A zone is a convex set of clock valuations, usually represented by clock constraints of the form $x \bowtie n$, where x is a clock, $\bowtie \in \{<, \leq, =, \geq, >\}$, and n is a (rational) number, or more generally by $x - y \bowtie n$, where y is another clock. This graph is usually small compared to the region graph. Nevertheless, this graph only preserves information about the existence of a state (*i.e.* a transition in the graph represents the existence of a location and a valuation in the source node to a location and a valuation in the destination node). This is not sufficient for our needs, *i.e.* to play a Büchi game.

The graph described in [Alur et al. \[1992\]](#) fits our needs and seems to be a good choice. However, we give a list of constraints that are sufficient for a graph to fit our needs. Any symbolic graph satisfying these constraints could be used to generate the game graph, on which we can play a Büchi game. We say that such graphs are *compatible* with Büchi games.

Definition 4.16 (Compatible graph). A symbolic graph $\mathcal{G}_s = \langle V_s, E_s \rangle$, with $E_s \subseteq V_s \times (\Sigma \cup \{t\}) \times V_s$ is *compatible* (with Büchi games) if it satisfies the following constraints:

1. V_s is a finite set such that $\forall v \in V_s, \exists l \in L, v \subseteq l \times 2^{\mathcal{V}(X)}$,
2. $\forall q \in Q, \exists! v \in V_s, q \in v$,
3. $E_s = E_s^a \cup E_s^\delta$,
4. $\forall v \in V_s, \forall a \in \Sigma, \exists! v' \in V_s, (\forall q \in v, \forall q' \in Q, q \xrightarrow{a} q' \implies q' \in v')$,
and $E_s^a = \{(v, a, v') \in V_s \times \Sigma \times V_s \mid \exists (q, q') \in v \times v', q \xrightarrow{a} q'\}$,
5. $\forall (v, v') \in V_s^2, \forall (q, q') \in v \times v', \forall \delta \in \mathbb{R}_{\geq 0}, q \xrightarrow{\delta} q' \implies (\forall q \in v, \exists \delta' \in \mathbb{R}_{\geq 0}, \exists q' \in v', q \xrightarrow{\delta'} q')$,
6. $\forall v \in V_s, \text{up}(v) \neq v \implies \exists! v' \in V_s, v \neq v' \wedge \forall (q, q') \in v \times v', \exists \delta \in \mathbb{R}_{\geq 0}, q' = q \text{ after } \delta \wedge \forall \delta' \leq \delta, q \text{ after } \delta' \in v \cup v'$,
and $E_s^\delta = \{(v, t, v') \in V_s \times \{t\} \times V_s \mid v \neq v' \wedge \forall (q, q') \in v \times v', \exists \delta \in \mathbb{R}_{\geq 0}, q' = q \text{ after } \delta \wedge \forall \delta' \leq \delta, q \text{ after } \delta' \in v \cup v'\}$.

Constraint (1) imposes that all the states of the semantics that are in a node of the symbolic graph share the same location. This allows us to easily define accepting nodes (as nodes whose locations are accepting).

Constraint (2) allows us to match each state of the semantics with a unique node in the symbolic graph.

Constraint (3) splits the set of edges between edges corresponding to actions and edges corresponding to delays. Each of these sets has its own constraints, described in (4) and (6).

Constraint (4) propagates the reachability and determinism of the timed automaton to the symbolic graph for actions, and defines the set E_s^a of edges corresponding to actions. The edges in E_s^a are labelled with the corresponding action from Σ .

Constraint (5) states that if a state of the semantics leads to another with a delay, and they are not in the same node, then all states in the first node can reach a state in the second node with a delay.

Constraint (6) requires that each node of the graph has at most one direct time successor, with which it is linked by an edge in the set E_s^δ of edges corresponding to delays. The edges in E_s^δ are labelled with the special action t , which is supposed not to belong to Σ .

The graph defined in Alur *et al.* [1992] is a graph that is compatible with Büchi games, as per Definition 4.16. This graph is the one that has been used as symbolic graph in the implementation (see Chapter 5).

A compatible graph is like the so-called zone graph used to compute reachability, but with more constraining properties. Constraints (5) and (6) can be seen as a kind of time determinism. In the usual zone graph used to compute reachability (see, for example, Bengtsson and Yi [2004]), an edge (v, v') between two nodes v and v' indicates that for every state of the semantics q' in v' , there exists a state q in v such that $q \rightarrow q'$. In \mathcal{G}_s , constraints (5) and (6) are more constraining, since an edge $(v, v') \in E_s^\delta$ between two nodes v and v' indicates that for all q in v and for all q' in v' , $q \xrightarrow{\delta} q'$ for some $\delta \in \mathbb{R}_{\geq 0}$. Note that for edges in E_s^a , the constraints are the same as in the usual zone graph (the existence of a state).

Example 4.3. Consider property φ_t (Fig. 3.8). Its corresponding symbolic compatible graph as per Alur *et al.* [1992] is given in Fig. 4.2. In the graph of Fig. 4.2, the nodes are labelled with a location and a zone, represented as a set of clock constraints. Edges can represent an event transition or the elapse of time. Red edges with filled diamond heads ($\text{---}\blacklozenge$) represent transitions with uncontrollable events, whereas orange edges with empty diamond heads ($\text{---}\lozenge$) represent transitions with controllable events. Purple edges with “vee” heads ($\text{---}\vee$) represent the elapse of time.

Thus, in Fig. 4.2, orange edges correspond to transitions labelled by `Write`, since it is the only controllable event. Red edges can represent transitions of any other event, `LockOn`, `LockOff`, or `Auth`. Edges are not duplicated, meaning that two events with the same controllability that label the same transition

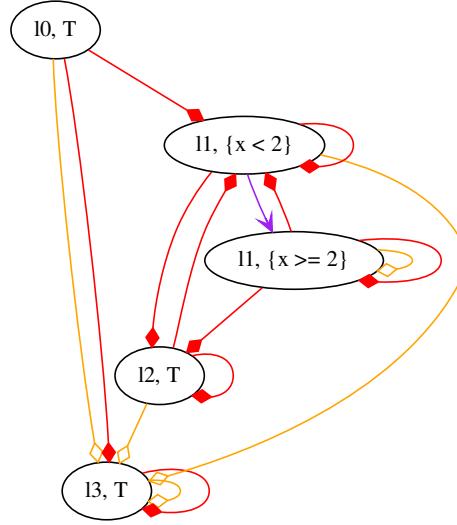


Figure 4.2 – A symbolic compatible graph of φ_t as per Alur *et al.* [1992].

will appear as a unique edge in the graph. For example, from node $(l0, \top)$, events *LockOff* and *LockOn* lead to $(l3, \top)$, but only one red edge is drawn.

From this symbolic graph, we define another graph, as in Section 4.2.2, on which we play a Büchi game. This graph is called *game graph*.

The Game Graph

Now that we have seen how to reduce Σ_c^* to the finite set Σ_c^n , and how to abstract the semantics of the timed automaton with a finite compatible graph, we can construct the graph on which we play a Büchi game. Let us consider $\mathcal{G}_s = (V_s, E_s)$, a symbolic graph compatible with Büchi games. We can now use Σ_c^n and \mathcal{G}_s to define \mathcal{G} , the finite graph on which to play the Büchi game:

Definition 4.17 (\mathcal{G}). $\mathcal{G} = \langle V, E \rangle$, where:

- $V = V_s \times \Sigma_c^n \times \{0, 1\}$,
- $E = \bigcup_{i=1}^6 E_i$, with
 - $E_1 = \{(\langle v, w, 0 \rangle, \langle v, w, 1 \rangle) \in V^2\}$,
 - $E_2 = \{(\langle v, c.w, 0 \rangle, \langle v \text{ after } c, w, 0 \rangle) \in V^2 \mid c \in \Sigma_c\}$,
 - $E_3 = \{(\langle v, w, 1 \rangle, \langle v \text{ after } u, w, 0 \rangle) \in V^2 \mid u \in \Sigma_u\}$,
 - $E_4 = \{(\langle v, w, 1 \rangle, \langle v, w.c, 0 \rangle) \in V^2 \mid c \in \Sigma_c \wedge w.c \in \Sigma_c^n\}$,

- $E_5 = \{(\langle v, w, 1 \rangle, \langle v', w, 0 \rangle) \in V^2 \mid (v, t, v') \in E_s^\delta\},$
- $E_6 = \{(\langle v, w, 1 \rangle, \langle v, w, 0 \rangle) \in V^2 \mid \text{up}(v) = v\},$




As per Definition 4.17, a node in the game graph \mathcal{G} is composed of a node of the symbolic graph \mathcal{G}_s , a buffer, and a player it belongs to. It is quite similar to the game graph as per Definition 4.5, except that the state of the untimed automaton is replaced by a node of the symbolic graph. The two players are, again, the enforcement monitor P_0 , whose associated number is 0, and the environment P_1 , whose associated number is 1. The set of edges is partitioned into six sets, each representing a different type of action. The four first sets, E_1 , E_2 , E_3 , and E_4 are similar to the ones in the untimed setting: E_1 contains the edges corresponding to P_0 letting P_1 play; edges in E_2 represent P_0 emitting the first event of its buffer; E_3 and E_4 contain edges corresponding to receiving an uncontrollable or controllable event, respectively, which are actions of P_1 . Edges in E_5 represent time elapse: it changes the node of the symbolic graph to its time successor if it has one. E_6 contains edges that allow us to consider finite inputs. Since plays are infinite, such edges are needed to allow the environment to receive nothing (it can be seen as adding an empty event to the input). Since time elapses when no event is received, these edges exist only from nodes that have no time successor, *i.e.* nodes that are stable by elapse of time.

On this graph, we play a Büchi game with the set of Büchi nodes being defined as:

$$B = \{(\langle l, Z \rangle, w, 0) \in V \mid l \in G\}$$

We can now consider W_0 the set of winning nodes of this game for player P_0 .

Example 4.4. Consider again property φ_t (Fig. 3.8) whose symbolic graph was represented in Fig. 4.2. The game graph associated with φ_t is given in Fig. 4.3. In this graph, the initial node is the square node, the Büchi nodes are the double-circled nodes, and the winning nodes (the nodes in W_0) are the rounded rectangular ones. The two first members represent a node of \mathcal{G}_s (see Fig. 4.2), and the third member is a prefix of the buffer of the enforcement mechanism, where w stands for the **Write** event, which is the only controllable event. As in the untimed setting, edges are represented differently according to the set they belong to:

- blue edges, with empty triangular heads () belong to E_1 (the enforcement mechanism does not emit),
- green edges, with filled triangular heads () belong to E_2 (the enforcement mechanism emits the first event of its buffer),
- orange edges, with empty diamond heads () belong to E_4 (a controllable event is received),

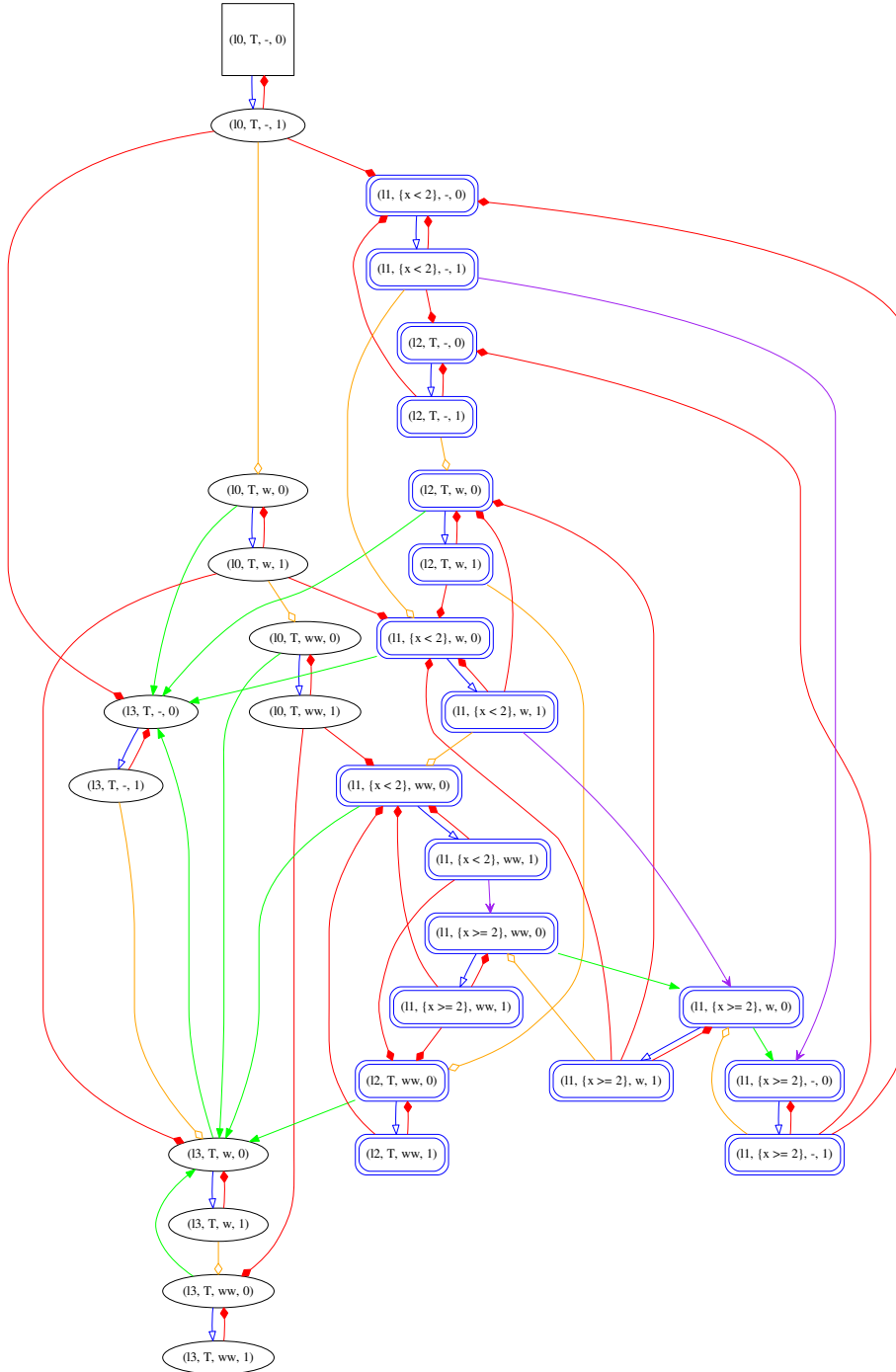


Figure 4.3 – Game graph associated with property φ_t

- red edges, with filled diamond heads (\blacklozenge) belong to E_3 (an uncontrollable event is received) or E_6 (no more event is to be received).
- purple edges, with “vee” heads (\vee) belong to E_5 (elapse of time).

For example, consider the node labelled $(l_1, \{x < 2\}, -, 1)$ in Fig. 4.3 (this node has a purple output edge). This node belongs to P_1 , meaning that this is the environment playing. Four edges leave this node. The purple edge (in E_5) leads to the node $(l_1, \{x \geq 2\}, -, 0)$, since if $x < 2$, letting enough time elapse (*i.e.* $2 - x$ time units) leads to the guard $x \geq 2$ being satisfied. Letting time elapse does not change the buffer nor the location, hence the destination node. Two red edges leave node $(l_1, \{x < 2\}, -, 1)$: one leads to node $(l_1, \{x < 2\}, -, 0)$, that corresponds to receiving the uncontrollable events *Auth* or *LockOff*; the other one leads to $(l_2, \top, -, 0)$, that corresponds to receiving the uncontrollable event *LockOn* (since (l_1, x) after *LockOn* = (l_2, x)). The last edge leaving node $(l_1, \{x < 2\}, -, 0)$ is the orange one, that corresponds to the reception of a controllable event, here *Write* is the only one, thus leading to node $(l_1, \{x < 2\}, w, 0)$, *i.e.* only the buffer changed and it is the turn of P_0 to play.

From this game graph, knowing the winning set W_0 for P_0 allows us to compute the “safe” states of an enforcement mechanism.

The Enforcement Function

Now, we redefine G , store_φ , E_φ , and Pre .

We can use W_0 to define, for $q \in Q$ and $w \in \Sigma_c^*$, $G(q, w)$, the set of words that can be output by an enforcement mechanism from state q with buffer w , ensuring compliance and soundness:

Definition 4.18 (G). For $q \in Q$, and $w \in \Sigma_c^*$,

$$\begin{aligned} G(q, w) = \{ & \sigma \in \text{tw}(\Sigma) \mid \Pi_\Sigma(\sigma) \preceq w \wedge q \text{ after } \sigma \in F_G \wedge \\ & \forall t \in \mathbb{R}_{\geq 0}, \forall v \in V_s, (q \text{ after } (\sigma, t) \in v) \\ & \implies \langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w), 1 \rangle \in W_0 \}, \end{aligned}$$

with:

$$\text{maxbuffer}(w) = \max_{\preceq}(\{w' \preceq w \mid w' \in \Sigma_c^n\}).$$

It is now possible to redefine E_φ , using this new definition of G :

Definition 4.19 (store_φ , E_φ). Let $\text{store}_\varphi \in \mathcal{F}(\text{tw}(\Sigma), \text{tw}(\Sigma) \times \Sigma_c^*)$ be the function defined inductively by:

$$(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi,$$

and for $\sigma \in \text{tw}(\Sigma)$, $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, if $t = \text{time}(\sigma.(\delta, a))$, $(\sigma_{s0}, \sigma_c) = \text{store}_\varphi(\sigma)$, and $\sigma_s = \text{obs}(\sigma_{s0}, t)$, then

$$\begin{cases} (\sigma.(\delta, a), \langle \sigma_s. (t - \text{time}(\sigma_s), a) . \sigma'_s, \sigma'_c \rangle) \in \text{store}_\varphi & \text{if } a \in \Sigma_u \\ (\sigma.(\delta, a), \langle \sigma_s. \sigma''_s, \sigma''_c \rangle) \in \text{store}_\varphi & \text{if } a \in \Sigma_c \end{cases}$$

where, for $q \in Q$, and $w \in \Sigma_c^*$,

$$\begin{aligned} T(q, w) &= \{t' \in \mathbb{R}_{\geq 0} \mid \forall t'' < t', G(q \text{ after } (\epsilon, t''), w) = \emptyset\}, \\ \kappa_\varphi(q, w) &= \min_{\text{lex}}(\max_{\preceq}(\{\epsilon\} \cup \bigcup_{t' \in T(q, w)} \{w' +_t t' \mid w' \in G(q \text{ after } (\epsilon, t'), w)\})) \\ \text{buf}_c &= \Pi_\Sigma(\text{nobs}(\sigma_{s0}, t)) . \sigma_c, \end{aligned}$$

and

$$\begin{aligned} \sigma'_s &= \kappa_\varphi(\text{Reach}(\sigma_s. (t - \text{time}(\sigma_s), a)), \text{buf}_c) & \sigma'_c &= \Pi_\Sigma(\sigma'_s)^{-1} . \text{buf}_c, \\ \sigma''_s &= \kappa_\varphi(\text{Reach}(\sigma_s, t), \text{buf}_c . a) +_t (t - \text{time}(\sigma_s)) & \sigma''_c &= \Pi_\Sigma(\sigma''_s)^{-1} . (\text{buf}_c . a). \end{aligned}$$

We then define the enforcement function E_φ as follows:

$$E_\varphi = \{(\langle \sigma, t \rangle, \text{obs}(\sigma_{s0}, t)) \in (\text{tw}(\Sigma) \times \mathbb{R}_{\geq 0}) \times \text{tw}(\Sigma) \mid \exists \sigma_c \in \Sigma_c^*, (\text{obs}(\sigma, t), \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi\}$$

Function store_φ takes a timed word σ as input, and outputs two words: σ_{s0} and σ_c . Timed word σ_{s0} is the output of the enforcement mechanism at infinite time. The controllable word of actions σ_c is the word composed of the remaining stored controllable actions of the enforcement mechanism (its buffer) at infinite time. In the definition of $G(q, w)$, the last condition requires that all nodes of \mathcal{G} that are reached by a word in $G(q, w)$ from q belong to W_0 . This is a strong condition, that is required to ensure that it is always possible to compute a word leading to an accepting state. Nevertheless, if the source node is not in W_0 , it is possible that letting time elapse leads to a node in W_0 , because it disabled some transition in the timed automaton. This explains why we defined the set $T(q, w)$, that allows us to consider words as potential outputs of the enforcement mechanism if it becomes sound (*i.e.* can ensure that the property will be satisfied) before the emission of the first event of this word, even if it is not at the time when the last event was received. Intuitively, $T(q, w)$ contains all the delays t such that an enforcement mechanism must wait at least t time units to be able to be sound. In other words, the enforcement mechanism can not ensure that the property will eventually always be satisfied from state q with buffer w , and it can not ensure it either by waiting less than t time units, for every t in $T(q, w)$. Then, $\kappa_\varphi(q, w)$ is the word that is to be output by the enforcement mechanism from state q with buffer w provided that the input does not change. It is the maximal word (with respect to \preceq_d)

4. Enforcing Properties using a Büchi Game

Table 4.1 – Table showing the values of $(\text{obs}(\sigma, t), \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi_t}$, with $\sigma = ((1, \text{Auth}) . (1, \text{LockOn}) . (2, \text{Write}) . (1, \text{LockOff}) . (1, \text{LockOn}) . (1, \text{Write}) . (1, \text{LockOff}))$ for different values of time t .

t	σ_s	σ_c
1	$(1, \text{Auth})$	ϵ
2	$(1, \text{Auth}) . (1, \text{LockOn})$	ϵ
4	$(1, \text{Auth}) . (1, \text{LockOn})$	Write
5	$(1, \text{Auth}) . (1, \text{LockOn}) . (3, \text{LockOff}) . (2, \text{Write})$	ϵ
6	$(1, \text{Auth}) . (1, \text{LockOn}) . (3, \text{LockOff}) . (1, \text{LockOn})$	Write
7	$(1, \text{Auth}) . (1, \text{LockOn}) . (3, \text{LockOff}) . (1, \text{LockOn})$	Write . Write
8	$(1, \text{Auth}) . (1, \text{LockOn}) . (3, \text{LockOff}) . (1, \text{LockOn}) . (2, \text{LockOff}) . (2, \text{Write}) . (0, \text{Write})$	ϵ

that belongs to $G(q, w)$. If $G(q, w)$ is empty, then $\kappa_{\varphi}(q, w)$ is the maximal word that belongs to $G(q \text{ after } (\epsilon, t), w)$, where t is the minimal time for which $G(q \text{ after } (\epsilon, t), w)$ is not empty. If $G(q \text{ after } (\epsilon, t), w)$ is empty for every $t \in \mathbb{R}_{\geq 0}$, then $\kappa_{\varphi}(q, w) = \epsilon$, meaning that the enforcement mechanism does not output anything. Thus, when the enforcement function is not sound, it outputs nothing but uncontrollable events.

Example 4.5. As in Example 3.8, we can follow the output of function store_{φ} over time with word $\sigma = (1, \text{Auth}) . (1, \text{LockOn}) . (2, \text{Write}) . (1, \text{LockOff}) . (1, \text{LockOn}) . (1, \text{Write}) . (1, \text{LockOff})$ as input: let $t \in \mathbb{R}_{\geq 0}$ be the observation time, and $(\text{obs}(\sigma, t), \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi}$. Then the values taken by σ_s and σ_c for different times t are given in Table 4.1. To understand the behaviour of store_{φ} , note that in the associated game graph, shown in Fig. 4.3, $\langle l_1, Z, w, p \rangle \in W_0$ and $\langle l_2, Z, w, p \rangle \in W_0$, for any $\langle l_1, Z, w, p \rangle \in V$ and $\langle l_2, Z, w, p \rangle \in V$.

As mentioned previously, an enforcement mechanism may not be sound from the beginning of an execution, but some uncontrollable events may lead to a state from which it becomes possible to be sound. In the definition of store_{φ} and E_{φ} , E_{φ} is sound whenever $T(q, w)$ is empty, with q the state reached by the output of E_{φ} at date t and w its buffer at this date. If $T(q, w)$ is empty, then the last value of σ'_s (or σ''_s depending on the controllability of the last input action) is in $G(q, w)$, meaning that the node in \mathcal{G} reached by the enforcement mechanism is in W_0 , therefore it is always possible to compute a word that leads to a state in $F_{\mathcal{G}}$. Since E_{φ} as per Definition 4.19 is equivalent to E_{φ} as per Definition 3.19, the definition of $\text{Pre}(\varphi)$ does not change, since it only depends on G :

Definition 4.20. $\text{Pre}(\varphi)$

$$\text{Pre}(\varphi) = \{(\sigma, t) \mid \sigma \in \text{Pre}(\varphi, t)\},$$

where, for $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$:

$$\text{Pre}(\varphi, t) = \{\sigma \in \text{tw}(\Sigma) \mid \exists t' \leq t, \\ G(\text{Reach}(\sigma|_{\Sigma_u}, t'), \Pi_{\Sigma}(\text{obs}(\sigma, t')|_{\Sigma_c})) \neq \emptyset\} \cdot \text{tw}(\Sigma)$$

Note that $\text{Pre}(\varphi)$ is time-extension-closed, meaning that once E_{φ} is sound, its output will always eventually satisfy φ in the future.

Considering that the output of our enforcement function was only the uncontrollable events so far, if $G(\text{Reach}(\sigma|_{\Sigma_u}, t), \Pi_{\Sigma}(\text{obs}(\sigma, t)|_{\Sigma_c}))$ is not empty, this means that the enforcement function becomes sound with input σ from time t , since there is a word that is safe to emit. Thus, $\text{Pre}(\varphi, t)$ is the set of inputs for which E_{φ} is sound after date t , and then E_{φ} is sound for any input in $\text{Pre}(\varphi)$ after its associated date.

Proposition 4.6. *E_{φ} as per Definition 4.19 is an enforcement function, as per Definition 4.11.*

Sketch of proof. We have to show that for all $\sigma \in \text{tw}(\Sigma)$, for all $t \in \mathbb{R}_{\geq 0}$ and $t' \geq t$, if $((\sigma, t), o_1) \in E_{\varphi}$ and $((\sigma, t'), o_2) \in E_{\varphi}$, then $o_1 \preceq o_2$, and that for all $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, if $((\sigma, \text{time}(\sigma \cdot (\delta, a))), o_3) \in E_{\varphi}$ and $((\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))), o_4) \in E_{\varphi}$, then $o_3 \preceq o_4$. To prove this, we first show by induction that $o_1 \preceq o_2$. Considering $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, we distinguish different cases according to the values of $\text{time}(\sigma \cdot (\delta, a))$ compared to t and t' :

- Case 1: $\text{time}(\sigma \cdot (\delta, a)) \leq t$. Then, $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma \cdot (\delta, a), t') = \sigma \cdot (\delta, a)$. Thus, if $(\sigma \cdot (\delta, a), \langle \sigma_{s0}, \sigma_{c0} \rangle) \in \text{store}_{\varphi}$, then $((\sigma \cdot (\delta, a), t), \text{obs}(\sigma_{s0}, t)) \in E_{\varphi}$ and $((\sigma \cdot (\delta, a), t'), \text{obs}(\sigma_{s0}, t')) \in E_{\varphi}$. Since $t \leq t'$, $\text{obs}(\sigma_{s0}, t) \preceq \text{obs}(\sigma_{s0}, t')$, which is what we want to prove.
- Case 2: $\text{time}(\sigma \cdot (\delta, a)) \geq t'$. Then, $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma, t)$ and $\text{obs}(\sigma \cdot (\delta, a), t') = \text{obs}(\sigma, t')$. Since in the definition of E_{φ} , only the observation of the input at the given date is used, it follows, by induction hypothesis, that the proposition holds.
- Case 3: $t < \text{time}(\sigma \cdot (\delta, a)) < t'$. Then, $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma, t)$, and $\text{obs}(\sigma \cdot (\delta, a), t') = \sigma \cdot (\delta, a)$. If $(\text{obs}(\sigma, t), \langle \sigma_{s0}, \sigma_{c0} \rangle) \in \text{store}_{\varphi}$, and $(\sigma \cdot (\delta, a), \langle \sigma_{s1}, \sigma_{c1} \rangle) \in \text{store}_{\varphi}$, then following the definition of store_{φ} , $\text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \preceq \sigma_{s1}$, meaning that $\text{obs}(\sigma_{s0}, t) \preceq \sigma_{s1}$, which is what we have to prove.

Thus, $o_1 \preceq o_2$. Then, what remains to show is that $o_3 \preceq o_4$. Following the definition of store_{φ} , it is clear that if $(\sigma, \langle \sigma_{s0}, \sigma_{c0} \rangle) \in \text{store}_{\varphi}$ and $(\sigma \cdot (\delta, a), \langle \sigma_{s1}, \sigma_{c1} \rangle) \in \text{store}_{\varphi}$, then $\sigma_{s0} \preceq \sigma_{s1}$, and thus $o_3 \preceq o_4$.

Proposition 4.7. *E_{φ} is sound with respect to φ in $\text{Pre}(\varphi)$ as per Definition 4.12.*

Sketch of proof. As in the untimed setting, the proof is made by induction on the input $\sigma \in \text{tw}(\Sigma)$. Similarly to the untimed setting, considering $\sigma \in \text{tw}(\Sigma)$, $t \in \mathbb{R}_{\geq 0}$, and $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, there are three possibilities:

- Case 1: $(\sigma \cdot (\delta, a), t) \notin \text{Pre}(\varphi)$. Then, the proposition holds.
- Case 2: $(\sigma \cdot (\delta, a), t) \in \text{Pre}(\varphi)$, but $(\sigma, \text{time}(\sigma \cdot (\delta, a))) \notin \text{Pre}(\varphi)$. Then, this is when the input reaches $\text{Pre}(\varphi)$. Considering the definition of $\text{Pre}(\varphi)$, we then prove that it is possible to emit a word with the controllable events seen so far, leading to a node of \mathcal{G} that is in W_0 .
- Case 3: $(\sigma, t') \in \text{Pre}(\varphi)$ (and thus $(\sigma \cdot (\delta, a), t)$ too). Then, we prove again that there exists a controllable word made with the stored actions that leads to a node in W_0 , but this time using the fact that the previous node was in W_0 (and since W_0 is the set of winning nodes in the Büchi game for P_0 , there always is a reachable successor node that is in W_0).

Proposition 4.8. E_φ is compliant, as per Definition 4.13.

Sketch of proof. As in the untimed setting, the proof is made by induction on the input σ , considering the different cases where the new event is controllable or uncontrollable. The only difference with the untimed setting is that one should consider dates (from delays) on top of actions.

Proposition 4.9. E_φ is optimal in $\text{Pre}(\varphi)$ as per Definition 4.14.

Sketch of proof. This proof is made by induction on the input σ . Whenever $\sigma \in \text{Pre}(\varphi)$, since E_φ is sound in $\text{Pre}(\varphi)$, then $E_\varphi(\sigma)$ is the maximal word (with respect to \preceq_d) that satisfies φ and is safe to output. It is maximal because in the definition of store_φ , κ_φ returns the longest word with lower delays (for lexicographic order), which corresponds to the maximum with respect to \preceq_d . Thus, outputting a grater word (with respect to \preceq_d) would lead to G being empty, meaning that the enforcement mechanism would not be sound. Thus, E_φ is optimal in $\text{Pre}(\varphi)$, since it outputs the maximal word with respect to \preceq_d that allows to be sound and compliant.

4.3.3 Enforcement Monitors

As in the untimed setting, we define an operational description of an enforcement mechanism whose output is exactly the output of E_φ , as per Definition 4.19.

Definition 4.21. An *enforcement monitor* \mathcal{E} for φ is a transition system $\langle C^\mathcal{E}, c_0^\mathcal{E}, \Gamma^\mathcal{E}, \hookrightarrow_\mathcal{E} \rangle$ such that:

- $C^\mathcal{E} = \text{tw}(\Sigma) \times \Sigma_c^* \times Q \times \mathbb{R}_{\geq 0}$ is the set of configurations.
- $c_0^\mathcal{E} = \langle \epsilon, \epsilon, q_0, 0 \rangle \in C^\mathcal{E}$ is the initial configuration.
- $\Gamma^\mathcal{E} = ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}) \times \text{Op} \times ((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})$ is the alphabet, composed of an optional input, an operation and an optional output.
The set of operations is $\{\text{dump}(\cdot), \text{pass-uncont}(\cdot), \text{store-cont}(\cdot), \text{delay}(\cdot)\}$.
Whenever $(\sigma, \bowtie, \sigma') \in \Gamma^\mathcal{E}$, it will be noted $\sigma / \bowtie / \sigma'$.
- $\hookrightarrow_\mathcal{E}$ is the transition relation defined as the smallest relation obtained by applying the following rules given by their priority order:
 - **Dump:** $\langle (\delta, a) \cdot \sigma_b, \sigma_c, q, \delta \rangle \xrightarrow{\epsilon / \text{dump}((\delta, a)) / (\delta, a)}_\mathcal{E} \langle \sigma_b, \sigma_c, q', 0 \rangle$, with $q' = q$ after a ,
 - **Pass-uncont:** $\langle \sigma_b, \sigma_c, q, \delta \rangle \xrightarrow{u / \text{pass-uncont}(u) / (\delta, u)}_\mathcal{E} \langle \sigma'_b, \sigma'_c, q', 0 \rangle$, with $q' = q$ after u , $\sigma'_b = \kappa_\varphi(q', \Pi_\Sigma(\sigma_b) \cdot \sigma_c)$, and $\sigma'_c = \Pi_\Sigma(\sigma'_b)^{-1} \cdot (\Pi_\Sigma(\sigma_b) \cdot \sigma_c)$,
 - **Store-cont:** $\langle \sigma_b, \sigma_c, q, \delta \rangle \xrightarrow{c / \text{store-cont}(c) / \epsilon}_\mathcal{E} \langle \sigma'_b, \sigma'_c, q, \delta \rangle$, with $\sigma'_b = \kappa_\varphi(q, \Pi_\Sigma(\sigma_b) \cdot \sigma_c \cdot c) +_t \delta$ and $\sigma'_c = \Pi_\Sigma(\sigma'_b)^{-1} \cdot (\Pi_\Sigma(\sigma_b) \cdot \sigma_c \cdot c)$,
 - **Delay:** $\langle \sigma_b, \sigma_c, (l, v), \delta \rangle \xrightarrow{\epsilon / \text{delay}(\delta') / \epsilon}_\mathcal{E} \langle \sigma_b, \sigma_c, (l, v + \delta'), \delta + \delta' \rangle$.

In a configuration $\langle \sigma_b, \sigma_c, q, \delta \rangle$, σ_b is the word to be output as time elapses; σ_c is the sequence of controllable actions from the input that are not used in σ_b ; q is the state of the semantics reached after reading what has already been output; δ is the time elapsed since the emission of the last event, it is used to output events with correct delays with respect to the previous output.

Compared to the enforcement monitor defined in Section 3.2.3, the time from the beginning is replaced by a delay since the last event output. Rule compute has disappeared, because it is not needed here. The reason is that the set G is not exactly the same in Section 3.2 and Section 4.3: in Section 3.2, it contains only words that can be emitted from the given state if this state is accepting, whereas in Section 4.3, it also contains words that can be emitted from the state reached when the first event is to be emitted. The result is the same because G was virtually computed for each possible valuation as time elapses in Section 3.2, which was exactly the purpose of rule compute.

At any time instant t , if $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, and the configuration reached by the enforcement monitor with input σ at date t is $\langle \sigma_b, \sigma_d, q, \delta \rangle$, then $\sigma_b = \text{nobs}(\sigma_s, t)$, $\sigma_d = \sigma_c$, $q = \text{Reach}(\sigma_s, t)$, and $\delta = t - \text{time}(\text{obs}(\sigma_s, t))$.

Example 4.6. An example of execution of an enforcement monitor as per Definition 4.21 enforcing property φ_t (see Fig. 3.8) is given in Fig. 4.4. Remember that in a configuration, the input is on the right, the output on the left, and the middle is the current configuration of the enforcement monitor. Variable t defines the global time of the execution. The input used for

the monitor in Fig. 4.4 is the same as in Table 4.1: $(1, \text{Auth}) . (1, \text{LockOn}) . (2, \text{Write}) . (1, \text{LockOff}) . (1, \text{LockOn}) . (1, \text{Write}) . (1, \text{LockOff})$. As in Example 3.8, in Fig. 4.4, valuations are represented as integers, giving the value of the only clock x of the property, *LockOff* is abbreviated as *off*, *LockOn* as *on*, and *Write* as *w*. First column depicts the dates of events, red text is the current output (σ_s) of the enforcement monitor, blue text shows the evolution of the first member of the configuration (σ_b) of the monitor and green text depicts the remaining input word at this date. The final output is the same as the one of the enforcement function E_φ as per Definition 4.19: $(1, \text{Auth}) . (1, \text{on}) . (2, \text{off}) . (1, \text{on}) . (2, \text{off}) . (2, \text{w}) . (0, \text{w})$. Note that this output is also the same as in Section 3.2, replacing dates with delays (since the input is also the same, replacing dates with delays).

Proposition 4.10. *The output o of \mathcal{E} as per Definition 4.21 for input σ at date t is such that $((\sigma, t), o) \in E_\varphi$.*

As in the untimed setting, in Proposition 4.10, the output of the enforcement monitor is the concatenation of the outputs of the word labelling the path followed by the enforcement monitor when reading σ . A formal definition is given in the proof of this proposition, in appendix A.2.2.

Sketch of proof. The proof is done by induction on σ . When receiving a new event, rule *store-cont()* can be applied if it is controllable, or rule *pass-uncont()* if it is uncontrollable. Doing so, the two first members of the configuration are recomputed, and they correspond exactly to the values of σ'_s (or σ''_s) and σ'_c (or σ''_c) in the definition of *store_φ*. Then, rule *delay()* can be applied, to reach the date of the first event in the second member of the current configuration, and then rule *dump()* can be applied to output it. This process can be repeated until the desired date is reached. Thus, when date t is reached, what has been emitted since the last rule *store-cont()* or *pass-uncont()* is *obs*(σ_b, t), where σ_b is the value of σ'_s (or σ''_s) as previously mentioned. Considering the definition of *store_φ*, it follows that the output of \mathcal{E} with input σ at date t is the exact same output as the one of E_φ .

Remark 5. As in the untimed setting (Section 4.2.3), for a configuration of the enforcement monitor $c = \langle \sigma_b, \sigma_c, q, \delta \rangle$, we can associate a node of the game graph $(\Pi_1(q), Z, w, 0)$, such that $\Pi_2(q) \in Z$, and w is the longest prefix of the buffer of the enforcement monitor, *i.e.* $\Pi_\Sigma(\sigma_b) . \sigma_c$, such that $(\Pi_1(q), Z, w, 0)$ is a node of the game graph. Then, if $(\sigma, t) \in \text{Pre}(\varphi)$, and c is the configuration reached by the enforcement monitor with input σ at date t , then $(\Pi_1(q), Z, w, 0)$ is a winning node for player P_0 .

t = 0	$\epsilon / \langle \epsilon, \epsilon, (l_0, 0), 0 \rangle / (1, \text{Auth}).(1, \text{on}).(2, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(1)$
t = 1	$\epsilon / \langle \epsilon, \epsilon, (l_0, 1), 1 \rangle / (0, \text{Auth}).(1, \text{on}).(2, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{pass-uncont}(\text{Auth})$
t = 1	$(1, \text{Auth}) / \langle \epsilon, \epsilon, (l_1, 0), 0 \rangle / (1, \text{on}).(2, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(1)$
t = 2	$(1, \text{Auth}) / \langle \epsilon, \epsilon, (l_1, 1), 1 \rangle / (0, \text{on}).(2, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{pass-uncont}(\text{on})$
t = 2	$(1, \text{Auth}).(1, \text{on}) / \langle \epsilon, \epsilon, (l_2, 1), 0 \rangle / (2, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(2)$
t = 4	$(1, \text{Auth}).(1, \text{on}) / \langle \epsilon, \epsilon, (l_2, 3), 2 \rangle / (0, \mathbf{w}).(1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{store-cont}(\mathbf{w})$
t = 4	$(1, \text{Auth}).(1, \text{on}) / \langle \epsilon, \mathbf{w}, (l_2, 3), 2 \rangle / (1, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(1)$
t = 5	$(1, \text{Auth}).(1, \text{on}) / \langle \epsilon, \mathbf{w}, (l_2, 4), 3 \rangle / (0, \text{off}).(1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{pass-uncont}(\text{off})$
t = 5	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}) / \langle (2, \mathbf{w}), \epsilon, (l_1, 0), 0 \rangle / (1, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(1)$
t = 6	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}) / \langle (2, \mathbf{w}), \epsilon, (l_1, 1), 1 \rangle / (0, \text{on}).(1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{pass-uncont}(\text{on})$
t = 6	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}) / \langle \epsilon, \mathbf{w}, (l_2, 1), 0 \rangle / (1, \mathbf{w}).(1, \text{off})$ $\downarrow \text{delay}(1)$
t = 7	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}) / \langle \epsilon, \mathbf{w}, (l_2, 2), 1 \rangle / (0, \mathbf{w}).(1, \text{off})$ $\downarrow \text{store-cont}(\mathbf{w})$
t = 7	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}) / \langle \epsilon, \mathbf{w.w}, (l_2, 2), 1 \rangle / (1, \text{off})$ $\downarrow \text{delay}(1)$
t = 8	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}) / \langle \epsilon, \mathbf{w.w}, (l_2, 3), 2 \rangle / (0, \text{off})$ $\downarrow \text{pass-uncont}(\text{off})$
t = 8	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}).(2, \text{off}) / \langle (2, \mathbf{w}).(0, \mathbf{w}), \epsilon, (l_1, 0), 0 \rangle / \epsilon$ $\downarrow \text{delay}(2)$
t = 10	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}).(2, \text{off}) / \langle (2, \mathbf{w}).(0, \mathbf{w}), \epsilon, (l_1, 2), 2 \rangle / \epsilon$ $\downarrow \text{dump}((2, \mathbf{w}))$
t = 10	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}).(2, \text{off}).(2, \mathbf{w}) / \langle (0, \mathbf{w}), \epsilon, (l_1, 2), 0 \rangle / \epsilon$ $\downarrow \text{dump}((0, \mathbf{w}))$
t = 10	$(1, \text{Auth}).(1, \text{on}).(3, \text{off}).(1, \text{on}).(2, \text{off}).(2, \mathbf{w}).(0, \mathbf{w}) / \langle \epsilon, \epsilon, (l_1, 2), 0 \rangle / \epsilon$

Figure 4.4 – Execution of an enforcement monitor with input $(1, \text{Auth}) . (1, \text{LockOn}) . (2, \text{Write}) . (1, \text{LockOff}) . (1, \text{LockOn}) . (1, \text{Write}) . (1, \text{LockOff})$

Conclusion

In this chapter, we have presented enforcement mechanisms that are sound, compliant, and optimal, and that use a Büchi game to ensure soundness. The outputs of the enforcement mechanisms in this chapter are the same as the outputs of enforcement mechanisms defined in Chapter 3. Nevertheless, the computation methods in these chapters are different, and the mechanisms presented in this chapter have better computation times in practice, and yield better performance. Moreover, we believe that algorithms are simpler using the method presented in this chapter rather than using the one in Chapter 3. A tool called GREP has been designed that constructs the game graph as described in Section 4.3.2, and uses it to modify its input so that it is a sound, compliant and optimal enforcement mechanism. This tool is described in the next chapter.

Chapter 5

GREP: Games for Runtime Enforcement of Properties

Introduction

In this chapter, we present GREP, the tool developed using the approach described in Section 4.3 to enforce timed properties. GREP is a sound, compliant and optimal enforcement mechanism.

We give a description of GREP in Section 5.1, detailing the different modules it is made of and the way it is used in Section 5.2, before presenting some performance evaluations, comparing GREP to TiPEX, another tool enforcing timed properties in Section 5.3.

The work described in this chapter has been published in [Renard *et al.* \[2017b\]](#).

5.1 Description of the approach

The strategy of GREP is the one described in Section 4.3. Given a timed regular property φ , and a partition of its alphabet into a set of controllable events Σ_c and a set of uncontrollable events Σ_u , GREP first builds a symbolic graph that is compatible with Büchi games, as per Definition 4.16. The graph used is the one described in [Alur *et al.* \[1992\]](#). Then, GREP builds a game graph as per Definition 4.17, using Σ_c as the set of controllable events and Σ_u as the set of uncontrollable events. Once the game graph is constructed, GREP computes the set W_0 of winning nodes for player P_0 (the enforcement mechanism).

Then, GREP can follow a real execution on the game graph, by watching the node that has been reached so far by its output, and the nodes that can be reached by emitting stored controllable actions (*i.e.* following the corresponding edges in the game graph). Whenever a winning node is reached by P_0 , the

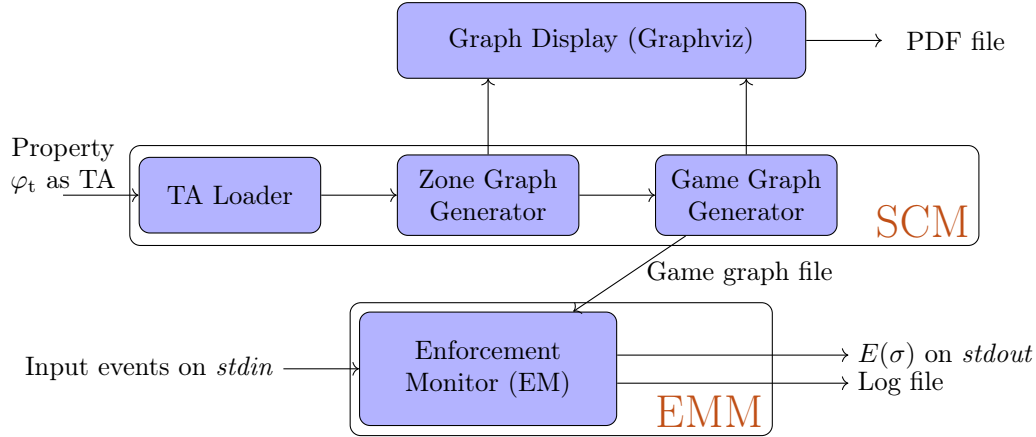


Figure 5.1 – General architecture of GREP

strategy is to emit as many events as possible, remaining in a winning node all the time. Since the game played is a Büchi game, it is always possible for P_0 to stay in a winning node whenever one is reached. Whenever a winning node is reached, the output of the EM is then guaranteed to satisfy the property.

5.2 General Functioning of GREP

GREP is a tool of about 6,000 lines of code¹ developed using the C language, available at <https://github.com/matthieurenard/GREP>. GREP is essentially composed of two modules (cf Fig. 5.1): the Symbolic Computing Module (SCM) and the Enforcement Monitor Module (EMM). It loads a TA file describing the desired property, and reads the inputs directly from *stdin*. The output of the EM is sent to *stdout*. This approach allows one to use GREP with off-the-shelf applications.

5.2.1 Symbolic Computing Module (SCM)

The Symbolic Computing Module is composed of three main components: a TA loader, the zone graph generator, and the game graph generator.

TA Loader

The TA loader is the component that parses a file containing the description of a timed automaton and loads it into a C structure. The file of the automaton is a textual description following a grammar designed for this purpose. An example file, that loads property φ_t (see Fig. 3.8), is provided in Listing 5.1. The file is parsed using a custom grammar, implemented using `lex` and `yacc`.

¹calculated with `cloc` (<https://github.com/AlDanial/cloc>)

```

automaton
{
    // Write
    cont {w}
    // Auth, LockOn, LockOff
    uncont {a, n, f}

    nodes
    {
        10 [initial];
        11 [accepting];
        12 [accepting];
        13;
    }

    clocks {x}

    edges
    {
        10 ->{a}{x}{} 11;
        10 ->{w}{}{} 13;
        10 ->{n}{}{} 13;
        10 ->{f}{}{} 13;
        11 ->{n}{}{} 12;
        11 ->{w}{}{x >= 2} 11;
        11 ->{f}{x}{} 11;
        11 ->{a}{}{} 11;
        11 ->{w}{}{x < 2} 13;
        12 ->{a}{}{} 12;
        12 ->{n}{}{} 12;
        12 ->{f}{x}{} 11;
        12 ->{w}{}{} 13;
        13 ->{w}{}{} 13;
        13 ->{a}{}{} 13;
        13 ->{n}{}{} 13;
        13 ->{f}{}{} 13;
    }
}

```

Listing 5.1 – Automaton file for φ_t

The automaton must also be deterministic and complete (see [Alur and Dill \[1992\]](#)). If the automaton is not deterministic, the behaviour is undefined. Once the timed automaton is loaded, a symbolic graph is computed by the Zone Graph Generator to abstract its infinite semantics into a finite graph that is compatible with Büchi games, as per Definition 4.16. The graph that is built is actually the one described in [Alur et al. \[1992\]](#).

Zone Graph Generator

From the timed automaton, a symbolic graph is constructed using zones. This zone graph must be compatible with Büchi games, as per Definition 4.16. An algorithm to compute a symbolic graph compatible with Büchi games is given in [Alur et al. \[1992\]](#). This algorithm has been implemented to compute the symbolic graph in this module.

In GREP, zones are represented by Difference Bound Matrices (DBMs), using the UPPAAL DBM library (UDBM, see [UDBM \[2011\]](#)), and its C API. The algorithm requires some functionality that is not provided by this C API (some of them exist in some higher-level wrappers), such as complementing zones into a list of zones. This functionality was added to our own wrapper of UDBM. No other third-party library was needed to compute the symbolic graph. This symbolic graph is used to build the final game graph, that will be used by the enforcement monitor.

Game Graph Generator

Using the symbolic graph, the Game Graph Generator builds a graph over which to play a Büchi game whose strategy is the one to be followed by the enforcement monitor. The graph is constructed as described in Definition 4.17. Once the graph is constructed, the Büchi game is solved for player P_0 (the enforcement monitor), with the set of Büchi nodes being the set of nodes whose location is accepting. The winning nodes are then the nodes from which the enforcement monitor ensures that its output will satisfy the property.

Following a path of winning nodes in the graph gives a strategy to follow such that the final output satisfies the property. This is how the EM uses the graph to actually enforce the property.

5.2.2 Enforcement Monitor Module (EMM)

The EMM uses the SCM to compute the output for a given input. It has five main public functions: `init(G)`, `getStrat()`, `delay(t)`, `eventRcvd(e)`, and `emit()`. Function `init(G)` initialises the EMM following the strategy from graph G . Function `getStrat()` gives the strategy to follow, *i.e.* whether the first action of the buffer should be output or not. Since time is abstracted by the zone graph for the SCM, it needs to be notified that some time has passed, which is

done by the use of function $\text{delay}(t)$, where t is the number of time units that have elapsed since the last call to delay , or the creation of the enforcement mechanism for the first call. Time units only need to be consistent with the ones used in the property. Function $\text{eventRcvd}(e)$ is used to inform the EMM that an event e has been read from the input. In this case, the EMM acts differently depending on the controllability of the event. Function $\text{emit}()$ is used to output the first action of the buffer. Uncontrollable events are output by function $\text{eventRcvd}()$, as required by compliance.

Note that these functions allow to use the EMM in both online (real-time) and offline (with a trace as input) settings. All these functions, except function $\text{getStrat}()$, return the number of time units required to reach the time successor of the current node (∞ if there is no time successor). It is the number of time units given to function $\text{delay}()$ if no event is received before and the strategy is not to emit.

Thus, the general algorithm to use the EMM in the offline setting is given in Algorithm 1. Basically, the EMM follows a path in the game graph. Thus, it considers the current node as the node reached by its output, and explores the strategy tree from this node. The EMM also stores the controllable actions that have not been output yet, and uses them to compute the possible output. Since the output should be the longest possible, with minimal possible delays, computing the strategy requires to explore the tree of all possible strategies. This is done by exploring the game graph, simulating the emission of the controllable actions of the buffer at all possible time instants. In each node belonging to P_0 , if the successor by emitting, *i.e.* green with empty triangular head arrow ($\xrightarrow{\text{green}}$) in the game graph, is winning, then it is explored, and if the time successor is also winning, it is explored as well, since waiting before emitting could allow the EMM to output more events. Each node is then associated with a score, corresponding to the number of actions that have been emitted to reach the node. Then, the EMM stores the node that has the biggest score, and the strategy to follow to reach it. If two nodes have the same score, then the first common ancestor is computed, and the one node that can be reached by emitting from this ancestor (the other node can be reached from this ancestor by waiting) is kept as the node to reach. This corresponds to computing the lexicographical order. This process is repeated for each node with the same score, with the previous stored node, such that in the end the stored node is the minimal node (for the lexicographic order) of all the nodes with the highest score.

Note that computing an output such that all actions are emitted whenever it is possible to emit them does not require to explore the strategy tree. Depending on the property, the two outputs could be the same, *i.e.* if the property is such that letting time elapse never enables a transition that eventually allows the EMM to output more events. Then the EMM can work faster by using an optimisation that does not compute any tree, but outputs actions

input : The game graph \mathcal{G} , the input sequence of events, through function **read** ()
output: The output of the enforcer mechanism, through function **emit**()

```
1 init( $G$ );
2  $\text{del} \leftarrow \infty$ ;
3 while The input sequence has not been read entirely do
4    $(\delta, a) \leftarrow \text{read}()$ ;
5   while  $\text{del} \leq \delta$  do
6      $\delta \leftarrow \delta - \text{del}$ ;
7      $\text{del} \leftarrow \text{delay}(\text{del})$ ;
8     while  $\text{getStrat}() = \text{EMIT}$  do
9       emit();
10    end
11  end
12   $\text{delay}(\delta)$ ;
13   $\text{del} \leftarrow \text{eventRcvd}(a)$ ;
14 end
15 while  $\text{del} < \infty$  or  $\text{getStrat}() = \text{EMIT}$  do
16   while  $\text{getStrat}() = \text{EMIT}$  do
17     emit();
18   end
19   if  $\text{del} < \infty$  then
20      $\text{del} \leftarrow \text{delay}(\text{del})$ ;
21   end
22 end
```

Algorithm 1: Main algorithm to enforce a property in offline mode

whenever possible, *i.e.* when the successor node by emitting is winning, if it is specified to do so.

To visualise the difference between the two computations, consider the property described in Fig. 3.10. For this property, considering for instance that the input word is $(0, \text{Write}) . (1, \text{Write})$, the output of GREP when exploring the execution tree would be (using delays): $(4, \text{Write}) . (4, \text{Write})$, whereas using the other algorithm, that emits events as soon as possible, the output would be $(2, \text{Write})$. The first one outputs more events, but the second one outputs its first event before the first one.

5.2.3 Running GREP

GREP is shipped with two executables: one to use the enforcement mechanism in offline mode, and the other in the online mode. Both of them take their input on the standard input. In the offline mode, the input is composed of events in the form (t, a) , where t is a date and a is an action, controllable or uncontrollable. In the online mode, only the action is given, the date is computed from the real time through a call to `gettimeofday()`. Note that these executables may build only on UNIX-like systems because of some system calls such as `gettimeofday()` and `clock_gettime()`. Excepting this, the tool is not system-dependent. The output (events with their dates) is printed on the standard output. Several options may be used:

- One of the two options `-a <automatonFile>` or `-g <graphFile>` must be passed to specify the property. The file `<automatonFile>` should be in the same format as the file shown in Listing 5.1. The file `<graphFile>` should be a file saved by this executable (see option `-s`), loading this kind of file should be faster than loading an automaton file since it contains the graph, which does not need to be computed again.
- `-s <graphFile>` saves the game graph in `<graphFile>`, to be loaded in another execution (see option `-g`).
- `-z <zoneGraphFile>` draws the zone graph using graphviz and store it (as PDF) in `<zoneGraphFile>`.
- `-d <gameGraphFile>` draws the game graph using graphviz and store it (as PDF) in `<gameGraphFile>`.
- `-t <timeFile>` logs times between the reception of two events in the file `<timeFile>`. This option is used to benchmark the program.
- `-l <logFile>` prints all the logs in `<logFile>`.

- **-f (fast)** use the optimised version, where actions are output whenever they can be instead of outputting the longest word possible with minimal dates.

If options **-s**, **-z**, **-d**, or **-t** are not given, then the corresponding action will just not happen. For example, without **-z**, the zone graph will not be saved. If none of the options among **-a** and **-g** is given, the program will print an error and abort. If both are given, then the automaton file is used. If option **-l** is not given, then the standard error is used as log file, which is not recommended (we recommend always using the option **-l**). If the option **-f** is not given, then the enforcement mechanism will output as many events as possible, with the lowest possible dates; enabling the option will make it output actions as soon as possible (*i.e.* if the node of the game graph reached by outputting is winning). Using option **-f** is usually faster, but the outputs might differ depending on the property.

For instance, the command:

```
game_enf_offline -a phit.tmtn -l log -d gameGraph.pdf < input
```

will enforce the property described in the file `phit.tmtn`, logging in the file `log`, reading its events from the file `input`. It will also draw the game graph in the file `gameGraph.pdf`.

The enforcement mechanism logs the mode in which it runs (default or fast) at the beginning, and when it stops, it logs the input, its output, the controllable actions that have not been output (what remains in its buffer), and a verdict that is WIN if its output satisfies the property, or LOSS otherwise (remember that some properties are not enforceable, see Example 3.1).

```
Enforcer initialized in default mode.
Shutting down the enforcer...
Summary of the execution:
Input: (0, Write) (1, Auth) (2, Write) (3, LockOn)
       (4, Write) (5, LockOff) (6, LockOn) (7, LockOff)
Output: (1, Auth) (2, Write) (2, Write) (3, LockOn)
       (5, LockOff) (6, LockOn) (7, LockOff) (9, Write)
Remaining events in the buffer:
VERDICT: WIN
Enforcer shutdown.
```

Listing 5.2 – Log file produced by GREP

For example, considering that `phit.tmtn` is the file given in Listing 5.1, the previous command with the input file containing the sequence:

```
(0,Write) . (1,Auth) . (2,Write) . (3,LockOn) . (4,Write) . (5,LockOff)
(6,LockOn) . (7,LockOff), produces the output:
(1,Auth) . (2,Write) . (2,Write) . (3,LockOn) . (5,LockOff) . (6,LockOn)
(7,LockOff) . (9,Write). The produced log file is given in Listing 5.2.
```

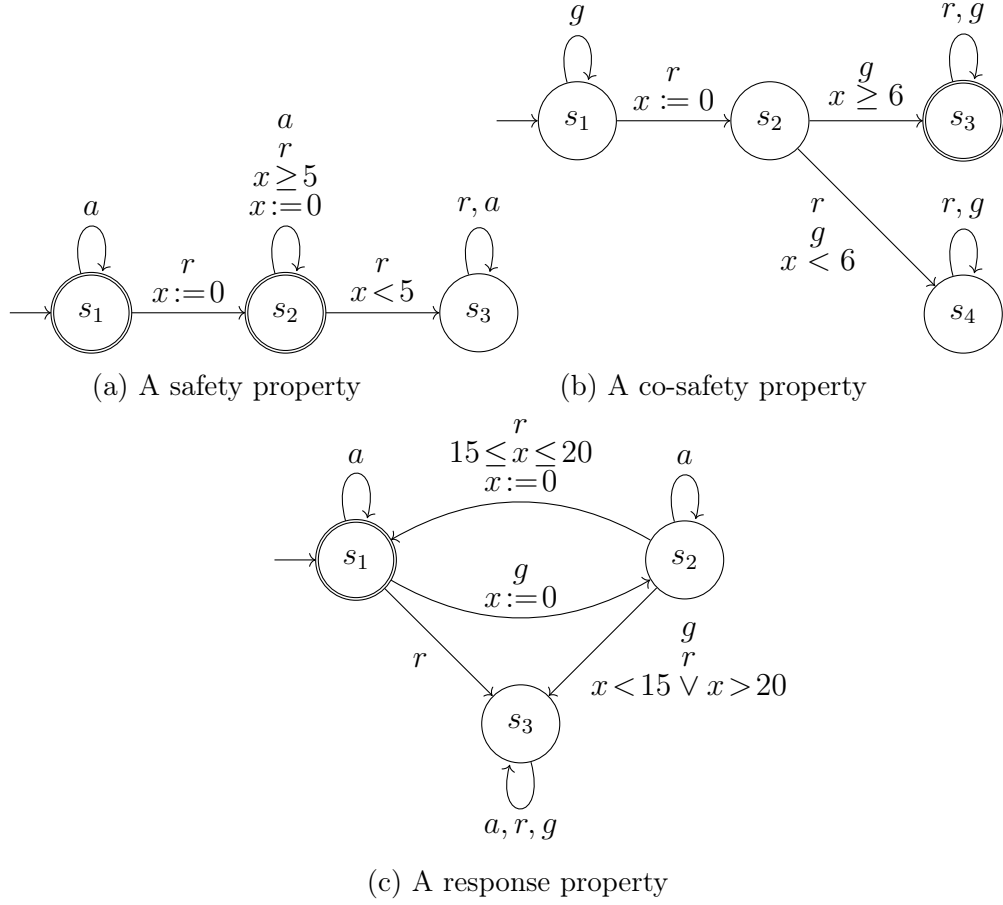


Figure 5.2 – Properties used to benchmark GREP

5.3 Performance Evaluation

5.3.1 Comparison with TiPEX

The performance of GREP has been evaluated on three properties that come along with TiPEX, the tool to which we compare. TiPEX (see [Pinisetty et al. \[2015b\]](#)) is, to our knowledge, the only other tool that acts as an enforcement mechanism for timed regular properties. These properties are described in Fig. 5.2. The safety property states that there should always be 5 time units between two r actions. The co-safety property states that the first r action should be followed by a g action, with a delay of at least 6 time units. The response property states that every grant (g) action should be followed by a release (r) action within 15 to 20 time units, without any grant action occurring between them.

For each of these properties, GREP has been run 100 times on every input among 100 inputs of 1000 events randomly generated. The time between the reception of two events has been saved for all of these executions. The

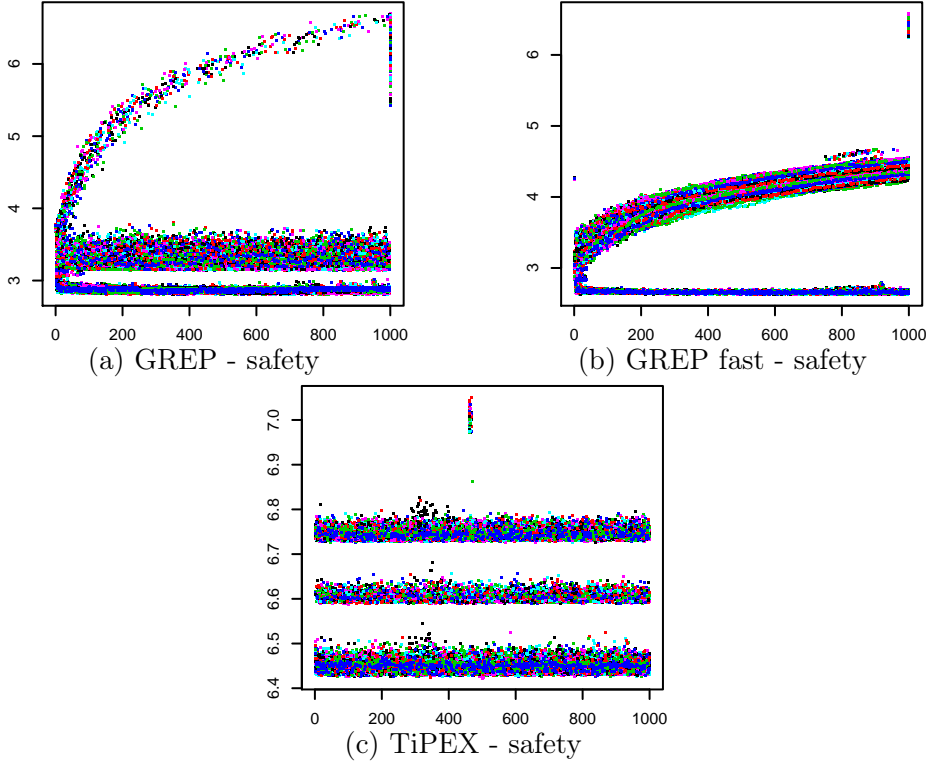


Figure 5.3 – Comparison of timings of GREP and TiPEX on the safety property. “GREP fast” means that option `-f` is used. The x axis corresponds to the events of the input (from 1 to 1000), and the y axis corresponds to the logarithm of the timings (in nanoseconds) between the reads of the events.

same times have been computed for TiPEX², reducing the number of inputs and iterations to have the benchmarks run in a reasonable amount of time. Figures 5.3 and 5.4 give a graphical visualisation of the performance of GREP and TiPEX.

Figures 5.3 and 5.4 are obtained as follows: each input is iterated several times (100 for GREP, less for TiPEX³), and the computation times (in nanoseconds) of the tool between the reads of two consecutive events of the input are stored. Then, the median time is computed for each of these times between all the iterations. We then plot the logarithm (in base 10) of these times against the reads of the events. We use a logarithmic scale because many values are low, and they would be merged in a line when using a linear scale. The results for GREP with option `-f` are given only for the safety property because they are similar to the results without the option for the two other properties. We can see that GREP is faster than TiPEX by several orders of

²We patched TiPEX to retrieve the times as we do in our tool, only modifying it to get times properly, and did not change the behaviour inside the part that is being measured.

³For some properties, running TiPEX was too long to run it as many times as GREP.

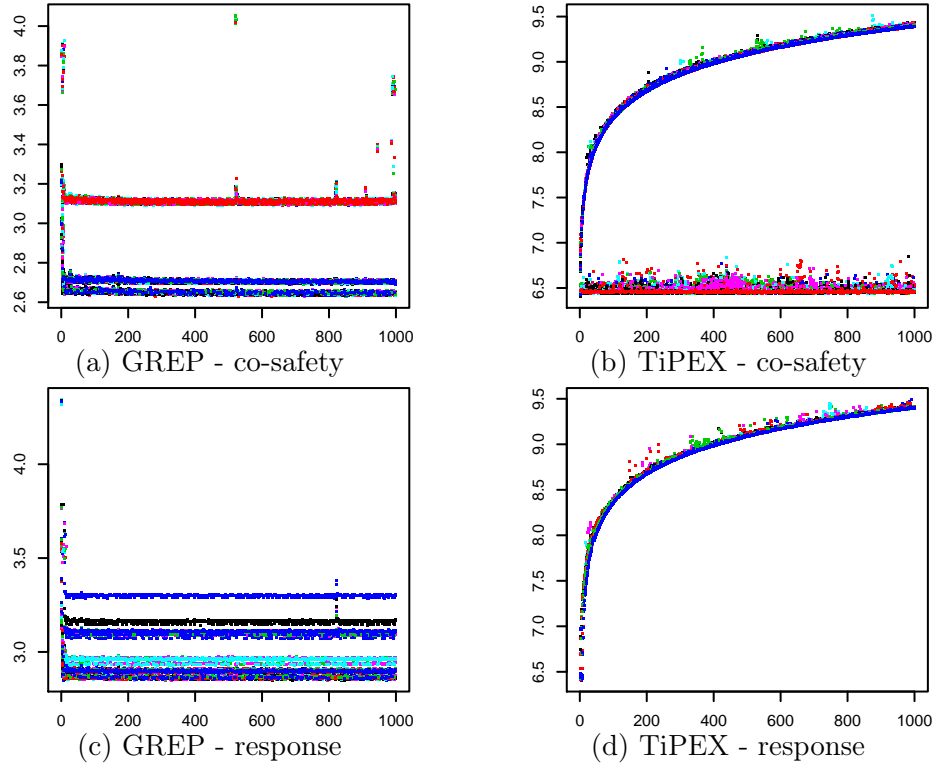


Figure 5.4 – Timings of GREP and TiPEX on the response and co-safety properties. The x axis corresponds to the events of the input (from 1 to 1000), and the y axis corresponds to the logarithm of the timings (in nanoseconds) between the reads of the events.

magnitude. GREP outputs many events in less than $10\mu s$ (4 on the scale of the graphs), whereas TiPEX takes at least 1 ms (6 on the scale of the graph) to output them. For the safety property, we can see that for some inputs, GREP takes an increasing amount of time to compute the strategy. This is due to the exploration of the strategy tree, which grows with the number of stored controllable actions. Using the optimised setting (-f) allows GREP to compute its output faster, as shown in Fig. 5.3b. The last vertical line has also many high values, because it represents the time to emit all the remaining actions after the last event from the input was read. For the co-safety and response properties, the time GREP takes between two events is less variable than for the safety property, mainly because the strategy of GREP is simpler: it consists in either emitting everything for the co-safety property (once state s_3 is reached) or emitting nothing for the response property, if the first stored controllable is an r while in state s_1 . TiPEX, on the other hand, takes a linearly-increasing amount of time to emit some events.

If GREP performs better than TiPEX on these properties, another improvement of GREP over TiPEX is that it can handle uncontrollable events. Using uncontrollable events can lower the performance of GREP, as is shown in Section 5.3.2.

5.3.2 Performance Evaluation with Uncontrollable Events

In this section, we show the limits of GREP when using uncontrollable events, with a property that is designed to be hard to be enforced by GREP, at least in its default mode.

Consider property φ_u described in Fig. 5.5, with u an uncontrollable event and c a controllable one. This property has two locations, s_1 and s_2 that are symmetrical: both of them require that a certain delay (15 time units for s_1 and 10 time units for s_2) has elapsed since the last event to emit a c event. As in Section 5.3.1, GREP has been tested for this property, using 100 random inputs of 1000 events. The results are presented in Fig. 5.6. As in Section 5.3.1, the x-axis of the plots represents the events of the inputs, from 1 to 1000, and the y-axis is the logarithm of the timings, in nanoseconds, between the reads of two consecutive events. The timings have been plotted with (Fig. 5.6b) and without (Fig. 5.6a) option -f.

Considering Fig. 5.6a, one can note that there seems to be four different behaviours: for some inputs, the timings between events is constant, and can be low, *i.e.* of about one microsecond, or a little bit higher, *i.e.* of about 10 microseconds; for some other inputs the timings are increasing, up to about 10 microseconds, or up to about 10 milliseconds for the last events. This difference between runs can be explained by the randomness of the events of the inputs. This benchmark has been made to show the limitation of GREP,

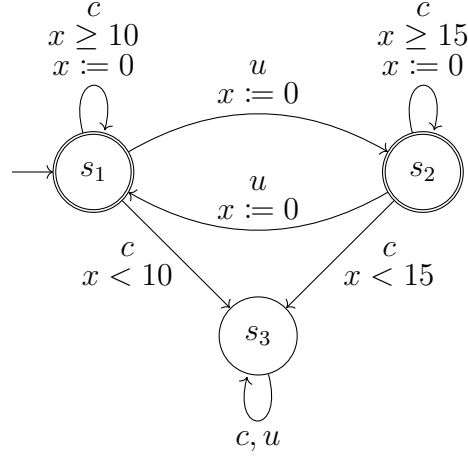
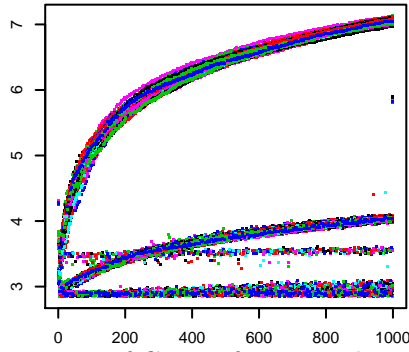
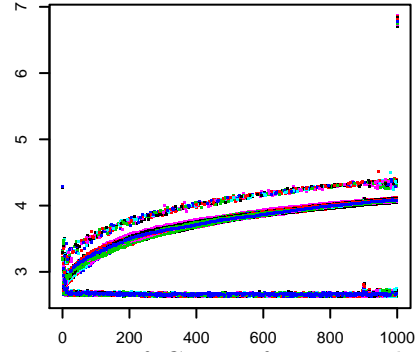


Figure 5.5 – Property φ_u .



(a) Timings of GREP for φ_u without option -f.



(b) Timings of GREP for φ_u with option -f.

Figure 5.6 – Timings of GREP for property φ_u with and without option -f.

thus the delays between events have been taken randomly between 0 and 3, meaning that events are received faster than it is possible to output controllable events (remember that c events must have a delay greater than 10 time units). Thus, depending on the proportion of uncontrollable events, that are emitted immediately, the buffer of stored controllable events grows as events are read. Property φ_u has been specifically designed to increase the number of stored controllable events.

Thus, in the worst case, the computation time of GREP increases with the size of its buffer. For some properties such as φ_u , receiving events with small delays (compared to guards) increases the size of the buffer, meaning that the computational overhead introduced by GREP could become too high for a use in online mode.

However, considering Fig. 5.6b, we can see that GREP performs better with option `-f`. Note that for φ_u , the outputs are the same with or without option `-f`. In the worst case, where GREP used 10 milliseconds without option `-f`, it only requires about 100 microseconds with option `-f`. In both cases, the timings increase with the size of the buffer, but option `-f` reduces the growth of the timings, and may allow using GREP in online mode where it is not possible without option `-f`.

This difference between the use of option `-f` and not using it can be explained by the fact that with option `-f`, GREP does not explore all the possible executions to output the longest word possible, but only decides if it is possible to emit a limited number of events.

Conclusion

In this chapter, we have presented GREP, a tool implementing an enforcement mechanism using the technique described in Chapter 4. Thus, GREP takes a timed automaton as input, and an execution, that is given on its standard input. GREP writes on its standard output the modified execution, that should satisfy the property. When it has read all the input, GREP outputs all the remaining possible events and then stops, outputting a summary of its run, including a verdict stating if the property is satisfied by its output. GREP can run in offline mode, reading delays with the events on its standard input, or in online mode, computing delays with the real time. Finally, GREP can run in its default mode, in which case its output is the same as the output of the enforcement mechanism described in Chapter 4, or in a “fast” mode, in which case it outputs events as soon as possible, reducing its computational time.

Conclusion

We give a summary of this thesis, as well as some potential future work and improvements.

This thesis falls in the domain of Runtime Verification, that aims at deciding whether a system's execution satisfies a desired property, at runtime. More than outputting a verdict indicating if the property is satisfied, runtime enforcement aims at modifying the execution of the running system to constrain it to satisfy the property. We have considered properties that were timed regular properties, *i.e.* represented by timed automata (Alur and Dill [1992]). The main contribution of this thesis is to consider some events as being uncontrollable, meaning that they are only observable by an enforcement mechanism, but can not be modified. We have formally described enforcement mechanisms in this context, with two different methods to compute the modifications made to the execution of the system, and implemented a tool using one of these two methods.

Summary

Chapter 3: Enforcing Properties with Uncontrollable Events: A First Approach first defines enforcement mechanisms with a functional point of view. An enforcement mechanism is represented by a function taking an execution (seen as a word over the alphabet of all possible events) and returning another execution. The argument given to the function corresponds to the input and the image corresponds to the output of the enforcement mechanism for that input. Requirements of enforcement mechanisms, such as soundness, compliance, and optimality, are given as constraints on such enforcement functions. Then, given a regular property, a sound, compliant, and optimal enforcement function is built for this property. To finish, a transition system is described, that builds the same output as the enforcement function previously defined. This scheme is then repeated to build an enforcement function and a transition system for a timed regular property.

Chapter 4: Enforcing Properties using a Büchi Game follows the same scheme as Chapter 3, but using a set-theoretic approach of functions.

It redefines the requirements expected of enforcement mechanisms using this formalism, and describes an enforcement function that is sound, compliant and optimal. This function is actually similar to the one that is defined in Chapter 3, but the computation method of the output has changed. The output is computed using a Büchi game over a graph representing the enforcement mechanism and its possibilities. Compared to Chapter 3, this allows the enforcement mechanism to precompute some of its decisions prior to its execution, thus trading some time complexity with space complexity at runtime. For a real-time use of enforcement mechanisms, time complexity seems to be the major concern, thus such trade-offs are worth doing. An equivalent transition system is also described, as in Chapter 3. Again, this is done for regular properties, and in a second section, for timed regular properties.

Chapter 5: GREP: Games for Runtime Enforcement of Properties

describes the implementation we made of the enforcement mechanism defined in Chapter 4, *i.e.* building a graph over which we solve a Büchi game. This implementation, called GREP, takes a timed automaton as input, using a custom grammar. It reads the input execution on its standard input, and outputs an execution that has been corrected to satisfy the property if possible on its standard output. GREP can work both in online and in offline mode (*i.e.* calculating delays between events based on the real time, or taking delays as inputs with actions). Two output modes are available: the default one is to emit as many events as possible, lowering delays next, the other one is to emit an event as soon as possible. The latter may output less events, but its output is faster to compute. Depending on the property, both output modes can be equivalent.

GREP has been compared to TiPEX, another tool implementing an enforcement mechanism, on properties provided by TiPEX. These properties do not have uncontrollable events, since TiPEX does not handle them. Overall, GREP performs better than TiPEX, and can handle the use of uncontrollable events. The computation overhead introduced by GREP seems adequate for a real-time use.

Future Work

Using such enforcement mechanisms in real-time applications. We have presented a formal construction of enforcement mechanisms for timed regular properties, and a tool acting as a proof of concept. Nevertheless, all the performance tests were made in offline mode, *i.e.* with dates given in the input with actions. In online mode, one should consider the computation overhead added by the tool itself in order to compute correct dates, because it could want to emit an event but by the time its computation ends, the

real date could become invalid. Working with enforcement mechanisms thus is challenging, since it requires some adaptations depending on the hardware used.

Instrumenting such enforcement mechanisms. As mentioned in the previous paragraph, instrumentation for real-time scenarios is not straightforward. Thus, it would be interesting to evaluate the limitations of the instrumentation of our enforcement mechanisms. Knowing the limitations of the instrumentation could also allow us to automate the instrumentation process. The instrumentation process would need to evaluate the performance of the enforcement mechanism to determine an upper bound of the overhead it introduces. This upper bound would then need to be taken into account by the enforcement mechanism itself to avoid violating the property due to the overhead in computational time it introduced.

Improve GREP. We have presented GREP, and some performance evaluation. We have seen that using GREP in online mode can be difficult if the number of stored controllable events is increasing, because the computational overhead increases simultaneously. Using option `-f`, *i.e.* changing the computation of the output can help reducing this overhead, but it can change the output for some properties. One way to improve GREP would thus be to detect automatically when it is possible to use option `-f` without changing the output, so that GREP can decide to use the better alternative. Another way to improve GREP could be to compute other outputs, with other enforcement primitives for example.

Enforcing other properties. We have been interested in this thesis only in the enforcement of timed regular properties, *i.e.* properties that can be represented by timed automata as described in [Alur and Dill \[1992\]](#). One could build enforcement mechanisms for properties with different formalisms. For instance, in [Bauer *et al.* \[2011\]](#), the authors build verification monitors for TLTL properties, using event-clocks automata (see [Alur *et al.* \[1999\]](#)). Thus, it should be possible to combine their verification monitors and the technique we presented to enforce these properties.

Enforcement on more complex systems. We have only considered simple systems, that produce a sequence of events given as the input of the enforcement mechanisms. One could be interested in enforcing properties on more complex systems, such as multi-threaded ones. Enforcing in a multi-threading context raises multiple problems: should enforcement mechanisms be themselves multi-threaded? If they are not, how can we not lose the interest of having a multi-threaded system, since enforcement mechanisms would act

as serialisers? What kind of property would be enforceable in this context? In particular, would such mechanisms be able to detect and prevent data races and deadlocks?

Some questions also naturally arise when dealing with enforcement on distributed systems. Would an enforcement mechanism for such systems be centralised, or distributed? What are the properties that can be enforced for such systems? Distributed systems usually communicate using some network, such as the internet, thus some latency must appear. Messages from different parts of the system may occur in any order, so enforcement mechanisms for distributed systems may consider enforcing each communication independently. Some work on the monitoring of decentralised systems has recently been done, for both decentralised and centralised specifications, for example in [Bauer and Falcone \[2012\]](#); [El-Hokayem and Falcone \[2017a,b\]](#). The decentralised enforcement of policies has also been studied in [Hallé *et al.* \[2016\]](#), where the authors build some kind of blockchain to ensure that the history of some file satisfies a given property.

Using techniques from control theory in runtime enforcement. As stated in the [Introduction](#) of this thesis (see the [Runtime Enforcement](#) section), runtime enforcement and control theory are close fields. It would be interesting to draw a precise boundary between these fields, because it would help understand better their differences, but also their resemblance. It would then be possible to deduce some techniques of control theory that could be used in runtime enforcement, and vice versa.

Appendix A

Proofs

A.1 Proofs of Chapter 3

A.1.1 Proofs for the Untimed Setting (Section 3.1)

In all this section, we will use the notations from Section 3.1, meaning that φ is a property whose associated automaton is $\mathcal{A}_\varphi = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$. In some proofs, we also use notations from Definition 3.10.

Proposition 3.1. *E_φ as per Definition 3.10 is an enforcement function as per Definition 3.1.*

Proof. We have to show that for σ and σ' in Σ^* , if $\sigma \preceq \sigma'$, then $E_\varphi(\sigma) \preceq E_\varphi(\sigma')$.

To do this, we just have to show that for all $a \in \Sigma$, $E_\varphi(\sigma) \preceq E_\varphi(\sigma . a)$. Indeed, if this holds for any $\sigma \in \Sigma^*$ and any $a \in \Sigma$, then if $\sigma \preceq \sigma'$, for any $i \in [|\sigma|; |\sigma'| - 1]$, $E_\varphi(\sigma'_{[..i]}) \preceq E_\varphi(\sigma'_{[..(i+1)]})$. Thus, by transitivity $E_\varphi(\sigma) \preceq E_\varphi(\sigma')$.

Let us consider $\sigma \in \Sigma_c^*$, $a \in \Sigma$, $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma . a)$. Then:

- if $a \in \Sigma_u$, $\sigma_t = \sigma_s . a . \sigma'_s$, where σ'_s is defined in Definition 3.10, meaning that $\sigma_s \preceq \sigma_t$.
- Otherwise, $a \in \Sigma_c$, and then $\sigma_t = \sigma_s . \sigma''_s$, where σ''_s is defined in Definition 3.10, thus again, $\sigma_s \preceq \sigma_t$.

In both cases, $E_\varphi(\sigma) = \sigma_s \preceq \sigma_t = E_\varphi(\sigma . a)$, meaning that $E_\varphi(\sigma) \preceq E_\varphi(\sigma')$ if $\sigma \preceq \sigma'$. Thus E_φ is an enforcement function. \square

Lemma A.1. $\forall \sigma \in \Sigma_c^*, \forall a \in \Sigma_c, I(\sigma) \subseteq I(\sigma . a)$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall a \in \Sigma_c, I(\sigma) \subseteq I(\sigma . a)$ ”. Let us show by induction that $P(\sigma)$ holds for every $\sigma \in \Sigma_c^*$.

Induction basis: if $a \in \Sigma_c$, then since $I(\epsilon) = \emptyset$, $I(\epsilon) \subseteq I(a)$. Thus, $P(\epsilon)$ holds.

Induction step: let us suppose that for $n \in \mathbb{N}$, for all $\sigma \in \Sigma_c^*$ such that $|\sigma| \leq n$, $P(\sigma)$ holds. Let us then consider $\sigma \in \Sigma_c^*$ such that $|\sigma| = n + 1$, and $a \in \Sigma_c$. Let $(h, \sigma_0) \in \Sigma_c \times \Sigma_c^*$ be such that $\sigma = h \cdot \sigma_0$ (they must exist since $|\sigma| > 0$).

Then, $|\sigma_0| = n$, and by induction hypothesis, $P(\sigma_0)$ holds, meaning that $I(\sigma_0) \subseteq I(\sigma_0 \cdot a)$. Moreover, following the definition of $S(\sigma_0 \cdot a)$, $S(\sigma_0) \subseteq S(\sigma_0 \cdot a)$. It follows that $S(\sigma_0) \cup I(\sigma_0) \subseteq S(\sigma_0 \cdot a) \cup I(\sigma_0 \cdot a)$, and thus $I(\sigma) = I(h \cdot \sigma_0) = \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0)) \subseteq \text{Pred}_h(S(\sigma_0 \cdot a) \cup I(\sigma_0 \cdot a)) = I(h \cdot \sigma_0 \cdot a) = I(\sigma \cdot a)$. This means that $P(\sigma \cdot a)$ holds.

Thus, by induction on the size of $\sigma \in \Sigma_c^*$, for all $\sigma \in \Sigma_c^*$, $P(\sigma)$ holds. This means that for all $\sigma \in \Sigma_c^*$, for all $a \in \Sigma_c$, $I(\sigma) \subseteq I(\sigma \cdot a)$. \square

Lemma A.2. $\forall \sigma \in \Sigma_c^*, \forall q \in Q, \forall u \in \Sigma_u, (q \in S(\sigma)) \implies (q \text{ after } u \in S(\sigma) \cup I(\sigma))$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall q \in Q, \forall u \in \Sigma_u, (q \in S(\sigma)) \implies (q \text{ after } u \in S(\sigma) \cup I(\sigma))$ ”. Let us show by induction that $P(\sigma)$ holds for any $\sigma \in \Sigma_c^*$.

Induction basis: let us consider $u \in \Sigma_u$ and $q \in S(\epsilon)$. Then, since $u \in \Sigma_u$, $u \in \Sigma_u^*$, and following the definition of $S(\epsilon)$, $q \text{ after } u \in S(\epsilon)$. Thus, $q \text{ after } u \in S(\epsilon) \cup I(\epsilon)$.

Induction step: let us suppose that for $\sigma \in \Sigma_c^*$, $P(\sigma)$ holds. Let us then consider $u \in \Sigma_u$, $a \in \Sigma_c$, and $q \in S(\sigma \cdot a)$.

Then, either $q \in S(\sigma)$ or $q \in \max_{\subseteq}(\{Y \subseteq F_G \mid Y \cap u\text{Pred}(\overline{Y \cup I(\sigma \cdot a)}) = \emptyset\})$. If $q \in S(\sigma)$, then by induction hypothesis, $P(\sigma)$ holds, meaning that $q \text{ after } u \in S(\sigma) \cup I(\sigma)$. Following lemma A.1, $I(\sigma) \subseteq I(\sigma \cdot a)$, and since $S(\sigma) \subseteq S(\sigma \cdot a)$, it follows that $S(\sigma) \cup I(\sigma) \subseteq S(\sigma \cdot a) \cup I(\sigma \cdot a)$. Thus, $q \text{ after } u \in S(\sigma \cdot a) \cup I(\sigma \cdot a)$. Otherwise, $q \in \max_{\subseteq}(\{Y \subseteq F_G \mid Y \cap u\text{Pred}(\overline{Y \cup I(\sigma \cdot a)}) = \emptyset\})$, and thus $q \text{ after } u \in S(\sigma \cdot a) \cup I(\sigma \cdot a)$. Thus, $P(\sigma \cdot a)$ holds.

By induction on σ , it follows that $P(\sigma)$ holds for any $\sigma \in \Sigma_c^*$. Thus, for all $\sigma \in \Sigma_c^*$, for all $u \in \Sigma_u$, for all $q \in Q$, $(q \in S(\sigma)) \implies (q \text{ after } u \in S(\sigma) \cup I(\sigma))$. \square

Lemma A.3. $\forall \sigma \in \Sigma_c^*, \forall q \in S(\sigma) \cup I(\sigma), G(q, \sigma) \neq \emptyset$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall q \in S(\sigma) \cup I(\sigma), G(q, \sigma) \neq \emptyset$ ”. Let us show by induction that $P(\sigma)$ holds for any $\sigma \in \Sigma_c^*$.

Induction basis: let us consider $q \in S(\epsilon) \cup I(\epsilon)$. Then, since $I(\epsilon) = \emptyset$, $q \in S(\epsilon)$. Following the definition of $S(\epsilon)$, this means that ϵ is such that $\epsilon \preceq \epsilon$ and q after $\epsilon = q \in S(\epsilon) = S(\epsilon^{-1} \cdot \epsilon)$. Following Definition 3.9, this means that $\epsilon \in G(q, \epsilon)$, and thus $G(q, \epsilon) \neq \emptyset$, and thus that $P(\epsilon)$ holds.

Induction step: let us suppose that for $n \in \mathbb{N}$, for all $\sigma \in \Sigma_c^*$ such that $|\sigma| \leq n$, $P(\sigma)$ holds. Let us then consider $\sigma \in \Sigma_c^*$ such that $|\sigma| = n$, $a \in \Sigma_c$ and $q \in S(\sigma \cdot a) \cup I(\sigma \cdot a)$. Then, we consider two cases:

- $q \in S(\sigma \cdot a)$, then ϵ is such that $\epsilon \preceq \sigma \cdot a$ and q after $\epsilon \in S(\sigma \cdot a) = S(\epsilon^{-1} \cdot (\sigma \cdot a))$, thus $\epsilon \in G(q, \sigma \cdot a)$.
- $q \in I(\sigma \cdot a)$, then let $(h, \sigma_0) \in \Sigma_c \times \Sigma_c^*$ be such that $h \cdot \sigma_0 = \sigma \cdot a$ (they must exist since $|\sigma \cdot a| > 0$). Then, $I(\sigma \cdot a) = I(h \cdot \sigma_0) = \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0))$, meaning that $q \in \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0))$. By induction hypothesis, since $|\sigma_0| = |\sigma| = n$, $P(\sigma_0)$ holds, meaning that $G(q \text{ after } h, \sigma_0) \neq \emptyset$. Let us consider $w \in G(q \text{ after } h, \sigma_0)$. Then, w is such that $w \preceq \sigma_0$ and $(q \text{ after } h) \text{ after } w \in S(w^{-1} \cdot \sigma_0)$. Thus, $h \cdot w \preceq h \cdot \sigma_0$ and $q \text{ after } (h \cdot w) = (q \text{ after } h) \text{ after } w \in S(w^{-1} \cdot \sigma_0) = S((h \cdot w)^{-1} \cdot (h \cdot \sigma_0))$. Thus, $h \cdot w \in G(q, h \cdot \sigma_0) = G(q, \sigma \cdot a)$.

In both cases, $G(q, \sigma \cdot a) \neq \emptyset$, meaning that $P(\sigma \cdot a)$ holds.

By induction on the size of $\sigma \in \Sigma_c^*$, it follows that $P(\sigma)$ holds for any $\sigma \in \Sigma_c^*$, meaning that for all $\sigma \in \Sigma_c^*$, for all $q \in S(\sigma) \cup I(\sigma)$, $G(q, \sigma) \neq \emptyset$. \square

Lemma A.4. $\forall \sigma \in \Sigma^*, (\sigma \notin \text{Pre}(\varphi) \wedge (\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)) \implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_c = \sigma|_{\Sigma_c})$.

Proof. For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate “ $(\sigma \notin \text{Pre}(\varphi) \wedge (\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)) \implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_c = \sigma|_{\Sigma_c})$ ”. Let us show by induction that $P(\sigma)$ holds for any $\sigma \in \Sigma^*$.

Induction basis: $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$, and since $\epsilon|_{\Sigma_u} = \epsilon|_{\Sigma_c} = \epsilon$, $P(\epsilon)$ holds.

Induction step: let us suppose that for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us then consider $a \in \Sigma$, $(\sigma_s, \sigma_b) = \text{store}_\varphi(\sigma)$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma \cdot a)$.

Then, if $\sigma \cdot a \in \text{Pre}(\varphi)$, $P(\sigma \cdot a)$ holds.

Let us now consider that $\sigma \cdot a \notin \text{Pre}(\varphi)$. Then, since $\text{Pre}(\varphi)$ is extension-closed, $\sigma \notin \text{Pre}(\varphi)$, and thus, by induction hypothesis, $\sigma_s = \sigma|_{\Sigma_u}$ and $\sigma_c = \sigma|_{\Sigma_c}$. We consider two cases:

- $a \in \Sigma_u$, then $\sigma_t = \sigma_s \cdot a \cdot \sigma'_s$, with $\sigma'_s \in G(\text{Reach}(\sigma_s \cdot a), \sigma_c) \cup \{\epsilon\}$. Since $\sigma \cdot a \notin \text{Pre}(\varphi)$, $G(\text{Reach}((\sigma \cdot a)|_{\Sigma_u}), (\sigma \cdot a)|_{\Sigma_c}) = \emptyset$. Moreover, since $a \in \Sigma_u$, $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} \cdot a = \sigma_s \cdot a$ and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} = \sigma_c$,

thus $G(\text{Reach}(\sigma_s . a), \sigma_c) = \emptyset$. It follows that $\sigma'_s \in \{\epsilon\}$, meaning that $\sigma_t = \sigma_s . a = \sigma_{|\Sigma_u} . a = (\sigma . a)_{|\Sigma_u}$, and $\sigma_d = \sigma'^{-1}_s . \sigma_c = \sigma_c = \sigma_{|\Sigma_c} = (\sigma . a)_{|\Sigma_c}$.

- $a \in \Sigma_c$, then $\sigma_t = \sigma_s . \sigma''_s$, with $\sigma''_s \in G(\sigma_s, \sigma_c . a) \cup \{\epsilon\}$. Since $\sigma . a \notin \text{Pre}(\varphi)$, $G(\text{Reach}((\sigma . a)_{|\Sigma_u}), (\sigma . a)_{|\Sigma_c}) = \emptyset$. Moreover, since $a \in \Sigma_c$, $(\sigma . a)_{|\Sigma_u} = \sigma_{|\Sigma_u} = \sigma_s$ and $(\sigma . a)_{|\Sigma_c} = \sigma_{|\Sigma_c} . a = \sigma_c . a$. Thus, $G(\text{Reach}(\sigma_s), \sigma_c . a) = \emptyset$, meaning that $\sigma''_s = \epsilon$. Thus, $\sigma_t = \sigma_s = \sigma_{|\Sigma_u} = (\sigma . a)_{|\Sigma_u}$ and $\sigma_d = \sigma''^{-1}_s . (\sigma_c . a) = \sigma_c . a = \sigma_{|\Sigma_c} . a = (\sigma . a)_{|\Sigma_c}$.

In both cases, $P(\sigma . a)$ holds.

By induction on $\sigma \in \Sigma^*$, for all $\sigma \in \Sigma^*$, if $\sigma \notin \text{Pre}(\varphi)$ and $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, then $\sigma_s = \sigma_{|\Sigma_u}$ and $\sigma_c = \sigma_{|\Sigma_c}$. \square

Proposition 3.2. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$, as per Definition 3.2.

Proof. We have to show that for any $\sigma \in \text{Pre}(\varphi)$, $E_\varphi(\sigma) \models \varphi$. Let $P(\sigma)$ be the predicate: “ $(\sigma \in \text{Pre}(\varphi) \wedge (\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)) \implies (E_\varphi(\sigma) \models \varphi \wedge \text{Reach}(\sigma_s) \in S(\sigma_c))$ ”. Let us prove by induction that for any $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis: If $\epsilon \in \text{Pre}(\varphi)$, then following the definition of $\text{Pre}(\varphi)$, $G(\text{Reach}(\epsilon), \epsilon) \neq \emptyset$. Thus $\epsilon \in G(\text{Reach}(\epsilon), \epsilon)$ (since ϵ is the only word satisfying $\epsilon \preceq \epsilon$). This means that $\text{Reach}(\epsilon)$ after $\epsilon = \text{Reach}(\epsilon) \in S(\epsilon)$. Considering that $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$, it follows that $E_\varphi(\epsilon) = \epsilon$, and thus, since $S(\epsilon) \subseteq F_G$, $E_\varphi(\epsilon) \models \varphi$. Thus $P(\epsilon)$ holds.

Induction step: Suppose now that, for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $a \in \Sigma$, $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma . a)$. Let us prove that $P(\sigma . a)$ holds. We consider three different cases:

- $(\sigma . a) \notin \text{Pre}(\varphi)$. Then $P(\sigma . a)$ holds.
- $(\sigma . a) \in \text{Pre}(\varphi) \wedge \sigma \notin \text{Pre}(\varphi)$. Then, since $\text{Pre}(\varphi)$ is extension-closed, it follows that $\sigma . a \in \{w \in \Sigma^* \mid G(\text{Reach}(w_{|\Sigma_u}), w_{|\Sigma_c}) \neq \emptyset\}$, meaning that $G(\text{Reach}((\sigma . a)_{|\Sigma_u}), (\sigma . a)_{|\Sigma_c}) \neq \emptyset$. Moreover, since $\sigma \notin \text{Pre}(\varphi)$, following lemma A.4, $\sigma_s = \sigma_{|\Sigma_u}$ and $\sigma_c = \sigma_{|\Sigma_c}$. Now, we consider two cases:
 - If $a \in \Sigma_u$, then $(\sigma . a)_{|\Sigma_u} = \sigma_{|\Sigma_u} . a = \sigma_s . a$, and $(\sigma . a)_{|\Sigma_c} = \sigma_{|\Sigma_c} = \sigma_c$. Thus, $G(\text{Reach}(\sigma_s . a), \sigma_c) \neq \emptyset$, meaning that $\sigma'_s = (\sigma_s . a)^{-1} . \sigma_t \in G(\text{Reach}(\sigma_s . a), \sigma_c)$. Thus, following the definition of G , $\text{Reach}(\sigma_s . a)$ after $\sigma'_s = \text{Reach}(\sigma_s . a . \sigma'_s) = \text{Reach}(\sigma_t) \in S(\sigma'^{-1}_s . \sigma_c) = S(\sigma_d)$. Moreover, since $S(\sigma_d) \subseteq F_G$, $E_\varphi(\sigma . a) = \sigma_t \models \varphi$. This means that $P(\sigma . a)$ holds.

- If $a \in \Sigma_c$, then $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$, and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} \cdot a = \sigma_c \cdot a$. Thus, $G(\text{Reach}(\sigma_s), \sigma_c \cdot a) \neq \emptyset$, meaning that $\sigma_s'' = \sigma_s^{-1} \cdot \sigma_t \in G(\text{Reach}(\sigma_s), \sigma_c \cdot a)$. As in the case where $a \in \Sigma_u$, it follows that $\text{Reach}(\sigma_t) \in S(\sigma_d)$ and thus $E_\varphi(\sigma \cdot a) \models \varphi$. This means that $P(\sigma \cdot a)$ holds.

Thus, if $\sigma \cdot a \in \text{Pre}(\varphi)$ but $\sigma \notin \text{Pre}(\varphi)$, $P(\sigma \cdot a)$ holds.

- $\sigma \in \text{Pre}(\varphi)$ (and then $(\sigma \cdot a) \in \text{Pre}(\varphi)$ since $\text{Pre}(\varphi)$ is extension-closed). Then, by induction hypothesis, $P(\sigma)$ holds, meaning that $\text{Reach}(\sigma_s) \in S(\sigma_b)$ and $E_\varphi(\sigma) \models \varphi$. Again, we consider two cases:
 - If $a \in \Sigma_u$, then, since $\text{Reach}(\sigma_s) \in S(\sigma_c)$, following lemma A.2, $\text{Reach}(\sigma_s)$ after $a = \text{Reach}(\sigma_s \cdot a) \in S(\sigma_c) \cup I(\sigma_c)$. Then, following lemma A.3, $G(\text{Reach}(\sigma_s \cdot a), \sigma_b) \neq \emptyset$. Thus, $\sigma_s' = (\sigma_s \cdot a)^{-1} \cdot \sigma_t \in G(\text{Reach}(\sigma_s \cdot a), \sigma_c)$. It follows that $\text{Reach}(\sigma_s \cdot a \cdot \sigma_s') = \text{Reach}(\sigma_t) \in S(\sigma_s'^{-1} \cdot \sigma_c) = S(\sigma_d)$, and thus, since $S(\sigma_d) \subseteq F_G$, $E_\varphi(\sigma \cdot a) = \sigma_t \models \varphi$. Henceforth, $P(\sigma \cdot a)$ holds.
 - If $a \in \Sigma_c$, then, since $\text{Reach}(\sigma_s) \in S(\sigma_c)$ and $S(\sigma_c) \subseteq S(\sigma_c \cdot a)$, $\text{Reach}(\sigma_s) \in S(\sigma_c \cdot a)$. Following lemma A.3, $G(\text{Reach}(\sigma_s), \sigma_c \cdot a) \neq \emptyset$. Thus, $\sigma_s'' = \sigma_s^{-1} \cdot \sigma_t \in G(\text{Reach}(\sigma_s), \sigma_c \cdot a)$. As in the case where $a \in \Sigma_u$, this leads to $\sigma_t \in S(\sigma_d)$ and $E_\varphi(\sigma \cdot a) \models \varphi$. Henceforth, $P(\sigma \cdot a)$ holds.

Thus, if $\sigma \in \text{Pre}(\varphi)$, $P(\sigma \cdot a)$ holds.

In all cases, $P(\sigma \cdot a)$ holds.

Thus, $P(\sigma) \implies P(\sigma \cdot a)$. By induction on σ , $\forall \sigma \in \Sigma^*$, $(\sigma \in \text{Pre}(\varphi) \wedge (\sigma_s, \sigma_b) = \text{store}_\varphi(\sigma)) \implies (E_\varphi(\sigma) \models \varphi \wedge \text{Reach}(\sigma_s) \in S(\sigma_b))$. In particular, for all $\sigma \in \Sigma^*$, $(\sigma \in \text{Pre}(\varphi)) \implies (E_\varphi(\sigma) \models \varphi)$. This means that E_φ is sound with respect to φ in $\text{Pre}(\varphi)$. \square

Proposition 3.3. E_φ is compliant, as per Definition 3.3.

Proof. For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate: “ $((\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)) \implies (\sigma_s|_{\Sigma_c} \cdot \sigma_c = \sigma|_{\Sigma_c} \wedge \sigma_s|_{\Sigma_u} = \sigma|_{\Sigma_u})$ ”. Let us prove that for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis : $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$, and $\epsilon|_{\Sigma_c} = \epsilon|_{\Sigma_c} \cdot \epsilon$, and $\epsilon|_{\Sigma_u} = \epsilon|_{\Sigma_u}$. Thus $P(\epsilon)$ holds.

Induction step : Let us suppose that for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, $a \in \Sigma$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma \cdot a)$. Let us prove that $P(\sigma \cdot a)$ holds. We distinguish two cases:

- If $a \in \Sigma_u$, then $\sigma_t = \sigma_s \cdot a \cdot \sigma'_s$, where σ'_s is defined in Definition 3.10, and $\sigma_t \cdot \sigma_d = \sigma_s \cdot a \cdot \sigma_c$. Therefore, $\sigma_{t|\Sigma_c} \cdot \sigma_d = (\sigma_t \cdot \sigma_d)_{|\Sigma_c}$, since $\sigma_d \in \Sigma_c^*$. Thus, $\sigma_{t|\Sigma_c} \cdot \sigma_d = \sigma_{s|\Sigma_c} \cdot \sigma_c$. Since $P(\sigma)$ holds, $\sigma_{t|\Sigma_c} \cdot \sigma_d = \sigma_{|\Sigma_c} = (\sigma \cdot a)_{|\Sigma_c}$. Moreover, since $\sigma'_s \in \Sigma_c^*$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u} \cdot a$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_u} = \sigma_{|\Sigma_u} \cdot a = (\sigma \cdot a)_{|\Sigma_u}$. Thus $P(\sigma \cdot a)$ holds.
- Otherwise, $a \in \Sigma_c$, and then $\sigma_t = \sigma_s \cdot \sigma''_s$, where σ''_s is defined in Definition 3.10, and $\sigma_t \cdot \sigma_d = \sigma_s \cdot \sigma_c \cdot a$. Therefore, $\sigma_{t|\Sigma_c} \cdot \sigma_d = (\sigma_t \cdot \sigma_d)_{|\Sigma_c} = (\sigma_s \cdot \sigma_c \cdot a)_{|\Sigma_c} = \sigma_{s|\Sigma_c} \cdot \sigma_c \cdot a$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_c} \cdot \sigma_d = \sigma_{\Sigma_c} \cdot a = (\sigma \cdot a)_{|\Sigma_c}$. Moreover, since $\sigma''_s \in \Sigma_c^*$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u}$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_u} = \sigma_{|\Sigma_u} = (\sigma \cdot a)_{|\Sigma_u}$. Thus $P(\sigma \cdot a)$ holds.

In both cases, $P(\sigma \cdot a)$ holds. Thus, for all $\sigma \in \Sigma^*$, for all $a \in \Sigma$, $P(\sigma) \implies P(\sigma \cdot a)$.

By induction on σ , for any $\sigma \in \Sigma^*$, if $((\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma))$, then $(\sigma_{s|\Sigma_c} \cdot \sigma_c = \sigma_{|\Sigma_c}$ and $\sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u})$.

Moreover, if $\sigma \in \Sigma^*$, $u \in \Sigma_u$, $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma \cdot u)$, then $\sigma_t = \sigma_s \cdot u \cdot \sigma'_s$, where σ'_s is defined in Definition 3.10. Thus $\sigma_s \cdot u \preceq \sigma_t$, and since $\sigma_s = E_\varphi(\sigma)$, and $\sigma_t = E_\varphi(\sigma \cdot u)$, it follows that $E_\varphi(\sigma) \cdot u \preceq E_\varphi(\sigma \cdot u)$. Thus, for any $\sigma \in \Sigma^*$, $E_\varphi(\sigma)_{|\Sigma_c} \preceq \sigma_{|\Sigma_c}$, $E_\varphi(\sigma)_{|\Sigma_u} = \sigma_{|\Sigma_u}$, and $\forall u \in \Sigma_u$, $E_\varphi(\sigma) \cdot u \preceq E_\varphi(\sigma \cdot u)$, meaning that E_φ is compliant. \square

Lemma A.5. $\forall \sigma \in \Sigma_c^*, \forall q \in Q, (q \notin S(\sigma)) \implies (\exists \sigma_u \in \Sigma_u^*, q \text{ after } \sigma_u \notin F \wedge \forall \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma)).$

Proof. For $\sigma \in \Sigma_c^*$ and $q \in Q$, let $P(\sigma, q)$ be the predicate: “ $\forall \sigma_u \in \Sigma_u^*, q \text{ after } \sigma_u \in F \vee \exists \sigma'_u \preceq \sigma_u, (\sigma'_u \neq \epsilon \wedge q \text{ after } \sigma'_u \in S(\sigma) \cup I(\sigma))$ ”. Let us show the contrapositive of the lemma, that is that for all $\sigma \in \Sigma_c^*$ and $q \in Q$, $P(\sigma, q) \implies q \in S(\sigma)$. We consider two cases:

- If $\sigma = \epsilon$, let us consider $q \in Q$ such that $P(\epsilon, q)$ holds. Then, since $\epsilon \in \Sigma_u^*$ and there does not exist a word w satisfying $w \preceq \epsilon \wedge w \neq \epsilon$, it follows that $q = q \text{ after } \epsilon \in F$. Let us consider $\sigma_u \in \Sigma_u^*$. Then, since $P(\epsilon, q)$ holds, either $q \text{ after } \sigma_u \in F$, or there exists $\sigma'_u \preceq \sigma_u$ such that $\sigma'_u \neq \epsilon$ and $q \text{ after } \sigma'_u \in S(\epsilon) \cup I(\epsilon)$. In this last case, since $I(\epsilon) = \emptyset$, $q \text{ after } \sigma'_u \in S(\epsilon)$. Following the definition of $S(\epsilon)$, since $\sigma_u'^{-1} \cdot \sigma_u \in \Sigma_u^*$, $(q \text{ after } \sigma'_u) \text{ after } (\sigma_u'^{-1} \cdot \sigma_u) = q \text{ after } \sigma_u \in F$. Thus, in all cases, $q \text{ after } \sigma_u \in F$. Thus, for all $\sigma_u \in \Sigma_u^*$, $q \text{ after } \sigma_u \in F$, meaning that $q \in S(\epsilon)$.

- If $\sigma \neq \epsilon$, there exists $\sigma' \in \Sigma_c^*$ and $a \in \Sigma$ such that $\sigma = \sigma'.a$, meaning that $S(\sigma)$ is such that $S(\sigma) = S(\sigma') \cup \max_{\subseteq}(\{Z \subseteq F \mid Z \cap \text{uPred}(\overline{Z \cup I(\sigma)}) = \emptyset\})$. Let us consider $q \in Q$ such that $P(\sigma, q)$ holds. Then, we define

$$Y = \{q \text{ after } \sigma_u \mid \sigma_u \in \Sigma_u^* \wedge \forall \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma)\}.$$

Since $P(\sigma, q)$ holds, $Y \subseteq F$. Moreover, if $y \in Y$ and $u \in \Sigma_u$, then:

- either y after $u \in S(\sigma) \cup I(\sigma)$, and then y after $u \in (Y \cup S(\sigma)) \cup I(\sigma)$,
- or y after $u \notin S(\sigma) \cup I(\sigma)$. Then, if $\sigma_u \in \Sigma_u^*$ is such that $y = q$ after σ_u (σ_u exists since $y \in Y$), then y after $u = (q \text{ after } \sigma_u) \text{ after } u = q \text{ after } (\sigma_u . u) \notin S(\sigma) \cup I(\sigma)$. Since $\sigma_u . u \in \Sigma_u^*$, y after $u \in Y \subseteq (Y \cup S(\sigma)) \cup I(\sigma)$.

Thus, y after $u \in (Y \cup S(\sigma)) \cup I(\sigma)$, and since following lemma A.2, $S(\sigma) \cap \text{uPred}(\overline{S(\sigma) \cup I(\sigma)}) = \emptyset$, this means that $(Y \cup S(\sigma)) \cap \text{uPred}(\overline{(Y \cup S(\sigma)) \cup I(\sigma)}) = \emptyset$. It follows that $(Y \cup S(\sigma)) \subseteq \max_{\subseteq}(\{Z \subseteq F \mid Z \cap \text{uPred}(\overline{Z \cup I(\sigma)}) = \emptyset\}) \subseteq S(\sigma)$. Since $q \in Y \subseteq S(\sigma)$, this means that $q \in S(\sigma)$.

Thus, for $\sigma \in \Sigma_c^*$ and $q \in Q$, $P(\sigma, q) \implies q \in S(\sigma)$. This means that the contrapositive also holds, thus $q \notin S(\sigma) \implies \neg P(\sigma, q)$, meaning that $q \notin S(\sigma) \implies (\exists \sigma_u \in \Sigma_u^*, q \text{ after } \sigma_u \notin F \wedge \forall \sigma'_u \preceq \sigma_u, q \text{ after } \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma))$. \square

Proposition 3.4. E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 3.7.

Proof. Let E be an enforcement function such that $\text{compliant}(E, \Sigma_c, \Sigma_u)$, and let us consider $\sigma \in \text{Pre}(\varphi)$ and $a \in \Sigma$ such that $E(\sigma) = E_\varphi(\sigma)$ and $|E(\sigma.a)| > |E_\varphi(\sigma.a)|$. We have to prove that there exists $\sigma_u \in \Sigma_u^*$ such that $E(\sigma.a.\sigma_u) \not\models \varphi$. Let us consider $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$. We consider two cases:

- $a \in \Sigma_u$. Then, since E is compliant, and $E(\sigma) = E_\varphi(\sigma) = \sigma_s$, there exists $\sigma_{s1} \preceq \sigma_c$ such that $E(\sigma.a) = E(\sigma).a.\sigma_{s1} = \sigma_s.a.\sigma_{s1}$. Moreover, there exists $\sigma'_s \preceq \sigma_c$ such that $E_\varphi(\sigma.a) = E_\varphi(\sigma).a.\sigma'_s = \sigma_s.a.\sigma'_s$. Since $|E(\sigma.a)| > |E_\varphi(\sigma.a)|$, $|\sigma_{s1}| > |\sigma'_s|$. Considering that $\sigma'_s = \max_{\preceq}(\text{G}(\text{Reach}(\sigma_s.a), \sigma_c) \cup \{\epsilon\})$, it follows that $\sigma_{s1} \notin \text{G}(\text{Reach}(\sigma_s.a), \sigma_c)$. Following the definition of G , this means that either $\sigma_{s1} \not\preceq \sigma_c$, but since E' is compliant, this is not possible, or that $\text{Reach}(\sigma_s.a)$ after $\sigma_{s1} \notin S(\sigma_{s1}^{-1} . \sigma_c)$. Let us consider $q = \text{Reach}(\sigma_s.a.\sigma_{s1})$ and $\sigma_{c1} = \sigma_{s1}^{-1} . \sigma_c$. Then, $q \notin S(\sigma_{c1})$. Following lemma A.5, this means that there exists $\sigma_u \in \Sigma_u^*$ such that q after $\sigma_u \notin F$ and for all $\sigma'_u \preceq \sigma_u$, $\sigma'_u \neq \epsilon \implies q$ after $\sigma'_u \notin S(\sigma_{c1}) \cup I(\sigma_{c1})$. Then, we consider two cases:

- If $E(\sigma.a.\sigma_u) = \sigma_s.a.\sigma_{s1}.\sigma_u$, then $\text{Reach}(E(\sigma.a.\sigma_u)) \notin F$, meaning that $E(\sigma.a.\sigma_u) \not\models \varphi$.
- Otherwise, since E is compliant, there exists $\sigma_{s2} \preceq \sigma_{c1}$ and $\sigma_{u1} \preceq \sigma_u$ such that $\sigma_{s2} \neq \epsilon$, $\sigma_{u1} \neq \epsilon$, and $E(\sigma.a.\sigma_{u1}) = \sigma_s.a.\sigma_{s1}.\sigma_{u1}.\sigma_{s2}$. Let us consider $q' = q$ after $\sigma_{u1}.\sigma_{s2}$ and $\sigma_{c2} = \sigma_{s2}^{-1}.\sigma_{c1}$. Then, since $\sigma_{u1} \preceq \sigma_u$ and $\sigma_{u1} \neq \epsilon$, q after $\sigma_{u1} \notin S(\sigma_{c1}) \cup I(\sigma_{c1})$. Thus, $q' = q$ after $\sigma_{u1}.\sigma_{s2} \notin S(\sigma_{c2}) \cup I(\sigma_{c2})$, because otherwise, q after $\sigma_{u1} = \text{Pred}_{\sigma_{s2}}(q') \in \text{Pred}_{\sigma_{s2}}(S(\sigma_{c2}) \cup I(\sigma_{c2})) \subseteq I(\sigma_{c1})$, which is absurd. Then, we can again use lemma A.5 to find a word $\sigma_{u2} \in \Sigma_u^*$ such that q' after $\sigma_{u2} \notin F$ and for any $\sigma'_u \preceq \sigma_{u2}$, q' after $\sigma'_u \notin S(\sigma_{c2}) \cup I(\sigma_{c2})$. Since $\sigma_{s2} \neq \epsilon$, $|\sigma_{c2}| < |\sigma_{c1}|$, thus the operation can be repeated a finite number of times (at most until all the controllable events of σ appear in the output of E). Thus, there exists $n \in \mathbb{N}$, there exists $(\sigma_{u1}, \sigma_{u2}, \dots, \sigma_{un})$, and $(\sigma_{s1}, \sigma_{s2}, \dots, \sigma_{sn})$, such that $E(\sigma.a.\sigma_{u1}.\sigma_{u2}.\dots.\sigma_{un}) = \sigma_s.a.\sigma_{s1}.\sigma_{u1}.\sigma_{s2}.\sigma_{u2}.\dots.\sigma_{sn}.\sigma_{un}$, and $\text{Reach}(\sigma_s.a.\sigma_{s1}.\sigma_{u1}.\sigma_{s2}.\sigma_{u2}.\dots.\sigma_{sn}.\sigma_{un}) \notin F$. This means that, if $\sigma_u = \sigma_{u1}.\sigma_{u2}.\dots.\sigma_{un}$, then $\sigma_u \in \Sigma_u^*$ and $E(\sigma.a.\sigma_u) \not\models \varphi$.

Thus, in all cases, there exists $\sigma_u \in \Sigma_u^*$ such that $E(\sigma.a.\sigma_u) \not\models \varphi$.

- $a \in \Sigma_c$. The proof is the same as in the case where $a \in \Sigma_u$, by replacing occurrences of “ $\sigma_s.a$ ” by “ σ_s ”, and occurrences of “ σ_b ” by “ $\sigma_b.a$ ”.

Thus, if E is an enforcement function such that there exists $\sigma \in \text{Pre}(\varphi)$, and $a \in \Sigma$ such that $\text{compliant}(E, \Sigma_u, \Sigma_c)$, $E(\sigma) = E_\varphi(\sigma)$, and $|E(\sigma.a)| > |E_\varphi(\sigma.a)|$, then there exists $\sigma_u \in \Sigma_u^*$ such that $E(\sigma.a.\sigma_u) \not\models \varphi$. This means that E_φ is optimal in $\text{Pre}(\varphi)$. \square

Proposition 3.5. *The output of the enforcement monitor \mathcal{E} as per Definition 3.12 for input σ is $E_\varphi(\sigma)$ as per Definition 3.10.*

Proof. Let us introduce some notation for this proof: for a word $w \in \Gamma^{\mathcal{E}*}$, we note $\text{input}(w) = \Pi_1(w(1)) . \Pi_1(w(2)) \dots \Pi_1(w(|w|))$, the word obtained by concatenating the first members (the inputs) of w . In a similar way, we note $\text{output}(w) = \Pi_3(w(1)) . \Pi_3(w(2)) \dots \Pi_3(w(|w|))$, the word obtained by concatenating all the third members (outputs) of w . Since all configurations are not reachable from $c_0^\mathcal{E}$, for $w \in \Gamma^{\mathcal{E}*}$, we note $\text{Reach}^\mathcal{E}(w) = c$ whenever $c_0^\mathcal{E} \xrightarrow{w}_\mathcal{E} c$, and $\text{Reach}^\mathcal{E}(w) = \perp$ if such a c does not exist. We also define the Rules function as follows:

$$\text{Rules} : \begin{cases} \Sigma^* & \rightarrow \Gamma^{\mathcal{E}*} \\ \sigma & \mapsto \max_{\preceq}(\{w \in \Gamma^{\mathcal{E}*} \mid \text{input}(w) = \sigma \wedge \text{Reach}^\mathcal{E}(w) \neq \perp\}) \end{cases}$$

For a word $\sigma \in \Sigma^*$, $\text{Rules}(\sigma)$ is the trace of the longest valid run in \mathcal{E} , *i.e.* the sequence of all the rules that can be applied with input σ . We then extend the

definition of output to words in Σ^* : for $\sigma \in \Sigma^*$, $\text{output}(\sigma) = \text{output}(\text{Rules}(\sigma))$. We also note ϵ the empty word of Σ^* , and $\epsilon^\mathcal{E}$ the empty word of $\Gamma^\mathcal{E}$.

For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate: “ $E_\varphi(\sigma) = \text{output}(\sigma) \wedge (((\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma) \wedge \text{Reach}^\mathcal{E}(\text{Rules}(\sigma)) = \langle q, \sigma_c^\mathcal{E} \rangle) \implies (q = \text{Reach}(\sigma_s) \wedge \sigma_c = \sigma_c^\mathcal{E}))$ ”.

Let us prove by induction that for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis: $E_\varphi(\epsilon) = \epsilon = \text{output}(\epsilon)$. Moreover, $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon)$, and $\text{Reach}^\mathcal{E}(\epsilon^\mathcal{E}) = c_0^\mathcal{E}$. Therefore, as $c_0^\mathcal{E} = \langle q_0, \epsilon \rangle$, $P(\epsilon)$ holds, because $\text{Reach}(\epsilon) = q_0$.

Induction step: Let us suppose now that for some $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $(\sigma_s, \sigma_c) = \text{store}_\varphi(\sigma)$, $q = \text{Reach}(\sigma_s)$, $a \in \Sigma$, and $(\sigma_t, \sigma_d) = \text{store}_\varphi(\sigma . a)$. Let us prove that $P(\sigma . a)$ holds.

Since $P(\sigma)$ holds, $\text{Reach}^\mathcal{E}(\text{Rules}(\sigma)) = \langle q, \sigma_c \rangle$ and $\sigma_s = \text{output}(\sigma)$. We consider two cases:

- $a \in \Sigma_u$. Then, considering $\sigma'_s = (\sigma_s . a)^{-1} . \sigma_t$, $\sigma_t = \sigma_s . a . \sigma'_s$. Since $a \in \Sigma_u$, rule `pass-uncont` can be applied: let us consider $q' = q$ after a . Then, $\langle q, \sigma_c \rangle \xrightarrow{a/\text{pass-uncont}(a)/a}_\mathcal{E} \langle q', \sigma_c \rangle$.

If $\sigma'_s = \epsilon$, $G(q', \sigma_c) = \emptyset$ or $G(q', \sigma_c) = \{\epsilon\}$, meaning that no other rule can be applied, and thus $P(\sigma . a)$ holds.

Otherwise, $\sigma'_s \neq \epsilon$, and thus $\sigma'_s \in G(q', \sigma_c)$, meaning that $G(q', \sigma_c) \neq \emptyset$ and $G(q', \sigma_c) \neq \{\epsilon\}$, thus rule `dump`($\sigma_c(1)$) can be applied. Since $\sigma'_s \preceq \sigma_c$, $\sigma'_s(1) = \sigma_c(1)$, thus if $q_1 = q'$ after $\sigma_c(1)$, $q_1 = q'$ after $\sigma'_s(1)$. If $\sigma'_s(1)^{-1} . \sigma'_s \neq \epsilon$, then $\sigma'_s(1)^{-1} . \sigma'_s \in G(q_1, \sigma_c(1)^{-1} . \sigma_c)$, meaning that rule `dump` can be applied again. Rule `dump` can actually be applied $|\sigma'_s|$ times, since for all $w \preceq \sigma'_s$, if $w \neq \sigma'_s$, then $w^{-1} . \sigma'_s \neq \epsilon$ and $w^{-1} . \sigma'_s \in G(q' \text{ after } w, w^{-1} . \sigma_c)$. Thus, after rule `dump` has been applied $|\sigma'_s|$ times, the configuration reached is $\langle q' \text{ after } \sigma'_s, \sigma'^{-1}_s . \sigma_c \rangle$. Moreover, the output produced by all the rules `dump` is σ'_s . Since no rule can be applied after the $|\sigma'_s|$ applications of the rule `dump`, $\text{output}(\sigma . a) = \text{output}(\sigma) . a . \sigma'_s = \sigma_t$, and $\text{Reach}^\mathcal{E}(\text{Rules}(\sigma . a)) = \langle q' \text{ after } \sigma'_s, \sigma'^{-1}_s . \sigma_c \rangle = \langle q \text{ after } a \text{ after } \sigma'_s, \sigma_d \rangle = \langle \text{Reach}(\sigma_s) \text{ after } a \text{ after } \sigma'_s, \sigma_d \rangle = \langle \text{Reach}(\sigma_s . a . \sigma'_s), \sigma_d \rangle = \langle \text{Reach}(\sigma_t), \sigma_d \rangle$.

Thus, if $a \in \Sigma_u$, $P(\sigma . a)$ holds.

- $a \in \Sigma_c$. Then, considering $\sigma''_s = \sigma_s^{-1} . \sigma_t$, $\sigma_t = \sigma_s . \sigma''_s$. Since $a \in \Sigma_c$, it is possible to apply the `store-cont` rule, and $\langle q, \sigma_c \rangle \xrightarrow{a/\text{store-cont}(a)/\epsilon}_\mathcal{E} \langle q, \sigma_c . a \rangle$.

Then, as in the case where $a \in \Sigma_u$, rule `dump` can be applied $|\sigma''_s|$ times, meaning that the configuration reached is $\langle q \text{ after } (\sigma_c . a)(1) . (\sigma_c . a)(2) . \dots . (\sigma_c . a)(|\sigma''_s|), (\sigma_c . a)(|\sigma''_s| + 1) . (\sigma_c . a)(|\sigma''_s| + 2) . \dots . (\sigma_c . a)(|\sigma_c . a|) \rangle$. Since $\sigma''_s \preceq \sigma_c . a$, $(\sigma_c . a)(1) . (\sigma_c . a)(2) . \dots . (\sigma_c . a)(|\sigma''_s|) = \sigma''_s$,

thus $\text{Reach}(\text{Rules}(\sigma . a)) = \langle q \text{ after } \sigma_s'', \sigma_s''^{-1} . (\sigma_c . a) \rangle = \langle \text{Reach}(\sigma_t), \sigma_d \rangle$.
Moreover, $\text{output}(\sigma . a) = \text{output}(\sigma) . \sigma_s'' = \sigma_s . \sigma_s'' = \sigma_t = E_\varphi(\sigma . a)$.

Thus, if $a \in \Sigma_c$, $P(\sigma . a)$ holds.

Thus, in all cases, $P(\sigma . a)$ holds.

This means that $P(\sigma) \implies P(\sigma . a)$. Thus, by induction on σ , for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds. In particular, for all $\sigma \in \Sigma^*$, $E_\varphi(\sigma) = \text{output}(\sigma)$. \square

A.1.2 Proofs for the Timed Setting (Section 3.2)

In all this section, notation from Section 3.2 is used, meaning that φ is represented by a TA $\mathcal{A}_\varphi = \langle L, l_0, X, \Sigma, \Delta, G \rangle$ whose semantics is $\llbracket \mathcal{A}_\varphi \rrbracket = \langle Q, q_0, \Gamma, \rightarrow, F_G \rangle$. Timed word use dates and not delays.

Proposition 3.6. *E_φ as defined in Definition 3.19 is an enforcement function, as per Definition 3.13.*

Proof. We have to show the two following propositions:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t, E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall (t, a) \in \mathbb{R}_{\geq 0} \times \Sigma, \sigma . (t, a) \in \text{tw}(\Sigma) \implies E_\varphi(\sigma, t) \preceq E_\varphi(\sigma . (t, a), t)$.

We first show that item 1 holds.

For $\sigma \in \text{tw}(\Sigma)$, let $P(\sigma)$ be the predicate: “ $\forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t, E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$ ”. Let us show by induction that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis: if $\sigma = \epsilon$, then let us consider $t \in \mathbb{R}_{\geq 0}$, and $t' \geq t$. Then, $E_\varphi(\epsilon, t) = \epsilon \preceq \epsilon = E_\varphi(\epsilon, t')$. Thus, $P(\epsilon)$ holds.

Induction step: let us suppose that, for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider (t'', a) such that $\sigma . (t'', a) \in \text{tw}(\Sigma)$, $t \in \mathbb{R}_{\geq 0}$, and $t' \geq t$.

- If $t \geq t''$, then let us consider $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t'')$, $(\sigma_{t1}, \sigma_{d1}, \sigma_{e1}) = \text{store}_\varphi(\sigma . (t'', a), t)$, and $(\sigma_{t2}, \sigma_{d2}, \sigma_{e2}) = \text{store}_\varphi(\sigma . (t'', a), t')$. Then, $E_\varphi(\sigma . (t'', a), t) = \sigma_{t1}$ and $E_\varphi(\sigma . (t'', a), t') = \sigma_{t2}$.

- If $a \in \Sigma_u$, then considering t_1 as defined in Definition 3.19, $t_1 = \min(\{t_0 \in \mathbb{R}_{\geq 0} \mid t_0 \geq t'' \wedge G(\text{Reach}(\sigma_s . (t'', a), t_0), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) . \sigma_c) \neq \emptyset\})$. Then,

$$\begin{aligned} \sigma_{d1} &= \min_{\text{lex}}(\max(\text{G}(\text{Reach}(\sigma_s . (t'', a), \min(\{t, t_1\})), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) . \sigma_c) \\ &\quad \cup \{\epsilon\})) +_t \min(\{t, t_1\}) \\ \sigma_{d2} &= \min_{\text{lex}}(\max(\text{G}(\text{Reach}(\sigma_s . (t'', a), \min(\{t', t_1\})), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) . \sigma_c) \\ &\quad \cup \{\epsilon\})) +_t \min(\{t', t_1\}). \end{aligned}$$

Case 1: $t \geq t_1$. Since $t' \geq t$, then $t' \geq t_1$, thus $\min(\{t', t_1\}) = \min(\{t, t_1\}) = t_1$, thus $\sigma_{d1} = \sigma_{d2}$. It follows that:

$$\sigma_{t1} = \sigma_s \cdot (t'', a) \cdot \text{obs}(\sigma_{d1}, t) \preceq \sigma_s \cdot (t'', a) \cdot \text{obs}(\sigma_{d1}, t') = \sigma_s \cdot (t'', a) \cdot \text{obs}(\sigma_{d2}, t') = \sigma_{t2}.$$

Case 2: $t < t_1$. Then, $\min(\{t, t_1\}) = t$. Since $t < t_1$, by definition of t_1 , this means that $G(\text{Reach}(\sigma_s \cdot (t'', a), t), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) \cdot \sigma_c) = \emptyset$, and thus $\sigma_{d1} = \epsilon$. Since $\sigma_{d1} = \epsilon$, $\sigma_{t1} = \sigma_s \cdot (t'', a) \preceq \sigma_s \cdot (t'', a) \cdot \text{obs}(\sigma_{d2}, t') = \sigma_{t2}$.

Thus, if $t' \geq t \geq t''$ and $a \in \Sigma_u$, $P(\sigma) \implies E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

- Otherwise, $a \in \Sigma_c$, and then considering t_2 as defined in Definition 3.19, $t_2 = \min(\{t_0 \in \mathbb{R}_{\geq 0} \mid t_0 \geq t'' \wedge G(\text{Reach}(\sigma_s, t_0), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) \cdot \sigma_c \cdot a) \neq \emptyset\})$. Then,

$$\begin{aligned} \sigma_{d1} &= \min_{\text{lex}}(\max_{\preceq}(G(\text{Reach}(\sigma_s, \min(\{t, t_2\})), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) \cdot \sigma_c \cdot a) \\ &\quad \cup \{\epsilon\})) +_t \min(\{t, t_2\}) \\ \sigma_{d2} &= \min_{\text{lex}}(\max_{\preceq}(G(\text{Reach}(\sigma_s, \min(\{t', t_2\})), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) \cdot \sigma_c \cdot a) \\ &\quad \cup \{\epsilon\})) +_t \min(\{t', t_2\}). \end{aligned}$$

Case 1: $t \geq t_2$. Since $t' \geq t$, $t' \geq t_2$, meaning that $\min(\{t, t_2\}) = \min(\{t', t_2\}) = t_2$, and thus $\sigma_{d1} = \sigma_{d2}$. It follows that $\sigma_{t1} = \sigma_s \cdot \text{obs}(\sigma_{d1}, t) \preceq \sigma_s \cdot \text{obs}(\sigma_{d1}, t') = \sigma_s \cdot \text{obs}(\sigma_{d2}, t') = \sigma_{t2}$.

Case 2: $t < t_2$. Then, $G(\text{Reach}(\sigma_s, \min(\{t, t_2\})), \Pi_\Sigma(\text{nobs}(\sigma_b, t'')) \cdot \sigma_c \cdot a) = \emptyset$, meaning that $\sigma_{d1} = \epsilon$. Thus, $\sigma_{t1} = \sigma_s \preceq \sigma_s \cdot \text{obs}(\sigma_{d2}, t') = \sigma_{t2}$.

Thus, if $t' \geq t \geq t''$ and $a \in \Sigma_c$, $P(\sigma) \implies E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

Therefore, if $t' \geq t \geq t''$, for all $a \in \Sigma$, $P(\sigma) \implies E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

- If $t' < t''$, then $t < t''$, $\text{obs}(\sigma \cdot (t'', a), t) = \text{obs}(\sigma, t)$, and $\text{obs}(\sigma \cdot (t'', a), t') = \text{obs}(\sigma, t')$. Thus,

$$\begin{aligned} E_\varphi(\sigma \cdot (t'', a), t) &= \text{store}_\varphi(\text{obs}(\sigma \cdot (t'', a), t), t) \\ &= \text{store}_\varphi(\text{obs}(\sigma, t), t) \\ &= E_\varphi(\sigma, t), \end{aligned}$$

and

$$\begin{aligned} E_\varphi(\sigma \cdot (t'', a), t') &= \text{store}_\varphi(\text{obs}(\sigma \cdot (t'', a), t'), t') \\ &= \text{store}_\varphi(\text{obs}(\sigma, t'), t') \\ &= E_\varphi(\sigma, t'). \end{aligned}$$

Since $P(\sigma)$ holds, then $E_\varphi(\sigma \cdot (t'', a), t) = E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t') = E_\varphi(\sigma \cdot (t'', a), t')$.

- If $t < t'' \leq t'$, then $\text{obs}(\sigma \cdot (t'', a), t) = \text{obs}(\sigma, t)$. Since $P(\sigma)$ holds, then $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t'')$. Let $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t'')$ and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t'', a), t')$.

Then, $\sigma_t = \sigma_s \cdot (t'', a) \cdot \text{obs}(\sigma_e, t')$ if $a \in \Sigma_u$, and $\sigma_t = \sigma_s \cdot \text{obs}(\sigma_e, t')$ if $a \in \Sigma_c$. In both cases, $\sigma_s \preceq \sigma_t$. This means that $E_\varphi(\sigma, t'') \preceq E_\varphi(\sigma \cdot (t'', a), t')$. Thus, $E_\varphi(\sigma \cdot (t'', a), t) = E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t'') \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

Thus, if $t < t'' \leq t'$, then $P(\sigma) \implies E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$.

Consequently, in all cases, if $t \leq t'$, then $P(\sigma) \implies E_\varphi(\sigma \cdot (t'', a), t) \preceq E_\varphi(\sigma \cdot (t'', a), t')$. This means that $P(\sigma) \implies P(\sigma \cdot (t'', a))$.

Thus, by induction on σ , for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Thus, for all $\sigma \in \text{tw}(\Sigma)$, for all $t \in \mathbb{R}_{\geq 0}$, for all $t' \geq t$, $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$.

Now, let us prove item 2. Let us consider $\sigma \in \text{tw}(\Sigma)$, and (t, a) such that $\sigma \cdot (t, a) \in \text{tw}(\Sigma)$. Then, if $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t)$, and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t, a), t)$, then either $\sigma_t = \sigma_s \cdot (t, a) \cdot \sigma'_s$, or $\sigma_t = \sigma_s \cdot \sigma''_s$, whether a is controllable or uncontrollable respectively, where σ'_s and σ''_s are defined in Definition 3.19. In both cases, $\sigma_s \preceq \sigma_t$. Thus, $E_\varphi(\sigma, t) = \Pi_1(\text{store}_\varphi(\text{obs}(\sigma, t), t)) = \sigma_s \preceq \sigma_t = \Pi_1(\text{store}_\varphi(\text{obs}(\sigma \cdot (t, a), t))) = E_\varphi(\sigma \cdot (t, a), t)$. This holds because, since $\sigma \cdot (t, a) \in \text{tw}(\Sigma)$, $\text{time}(\sigma) \leq t$, thus $\text{obs}(\sigma, t) = \sigma$. Thus, for all $\sigma \in \text{tw}(\Sigma)$, for all $t \in \mathbb{R}_{\geq 0}$ and $t' \geq t$, $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma, t')$ and $E_\varphi(\sigma, t) \preceq E_\varphi(\sigma \cdot (t, a), t)$.

This means that E_φ is an enforcement function. \square

Lemma A.6. $\forall t \in \mathbb{R}_{\geq 0}, \forall \sigma \in \text{tw}(\Sigma),$
 $(\sigma \notin \text{Pre}(\varphi, t) \wedge (\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t))$
 $\implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_b = \epsilon \wedge \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})).$

Proof. For $\sigma \in \text{tw}(\Sigma)$, let $P(\sigma)$ be the predicate “ $\forall t \geq \text{time}(\sigma), (\sigma \notin \text{Pre}(\varphi, t) \wedge (\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t)) \implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_b = \epsilon \wedge \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c}))$ ”. Let us prove by induction that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis: for $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$. Then, $\text{store}_\varphi(\epsilon, t) = (\epsilon, \epsilon, \epsilon)$. Considering that $\epsilon \in \text{tw}(\Sigma_u)$, and $\epsilon = \Pi_\Sigma(\epsilon|_{\Sigma_c})$, $P(\epsilon)$ trivially holds (whether $\epsilon \in P(\varphi, t)$ or not).

Induction step: suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider (t', a) such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$, and $t \geq t'$. Let us also consider $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$ and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t', a), t)$.

Then, if $\sigma \cdot (t', a) \in \text{Pre}(\varphi, t)$, $P(\sigma \cdot (t', a))$ trivially holds. Thus, let us suppose that $\sigma \cdot (t', a) \notin \text{Pre}(\varphi, t)$. Since $\sigma \preceq \sigma \cdot (t', a)$ and $t \geq t'$, it follows that $\sigma \notin \text{Pre}(\varphi, t')$. By induction hypothesis, this means that $\sigma_s = \sigma|_{\Sigma_u}$, $\sigma_b = \epsilon$,

and $\sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})$. Then, since $\sigma \cdot (t', a) \notin \text{Pre}(\varphi, t)$, following the definition of $\text{Pre}(\varphi, t)$ (Definition 3.20), this means that for all $t'' \leq t$, $G(\text{Reach}(\text{obs}(\sigma \cdot (t', a), t'')|_{\Sigma_u}, t''), \Pi_\Sigma(\text{obs}(\sigma \cdot (t', a), t'')|_{\Sigma_c})) = \emptyset$. In particular, $G(\text{Reach}((\sigma \cdot (t', a))|_{\Sigma_u}, t), \Pi_\Sigma((\sigma \cdot (t', a))|_{\Sigma_c})) = \emptyset$ (since $t \geq t'$, $\text{obs}(\sigma \cdot (t', a), t) = \sigma \cdot (t', a)$). Then, there are two cases:

- If $a \in \Sigma_u$, then, since $(\sigma \cdot (t', a))|_{\Sigma_u} = \sigma|_{\Sigma_u} \cdot (t', a) = \sigma_s \cdot (t', a)$, and $\Pi_\Sigma((\sigma \cdot (t', a))|_{\Sigma_c}) = \Pi_\Sigma(\sigma|_{\Sigma_c}) = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c$, we have $G(\text{Reach}(\sigma_s \cdot (t', a), t), \Pi_\Sigma(\sigma_b, t') \cdot \sigma_c) = \emptyset$. This means that $t < t_1$, where t_1 is defined in Definition 3.19, and thus $\sigma_d = \epsilon$. Since $\sigma_t = \sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t)$, $\sigma_t = \sigma_s \cdot (t', a) = (\sigma \cdot (t', a))|_{\Sigma_u}$, and $\sigma_e = \sigma_c = \sigma|_{\Sigma_c} = (\sigma \cdot (t', a))|_{\Sigma_c}$. Thus, $P(\sigma \cdot (t', a))$ holds if $a \in \Sigma_u$.
- If $a \in \Sigma_c$, then, $(\sigma \cdot (t', a))|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$, and $\Pi_\Sigma((\sigma \cdot (t', a))|_{\Sigma_c}) = \Pi_\Sigma(\sigma|_{\Sigma_c}) \cdot a = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a$. Thus, $G(\text{Reach}(\sigma_s, t), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a) = \emptyset$. This means that $t < t_2$, where t_2 is defined in Definition 3.19, and thus $\sigma_d = \epsilon$. Since $\sigma_t = \sigma_s \cdot \text{obs}(\sigma_d, t)$, $\sigma_t = \sigma_s = \sigma|_{\Sigma_u} = (\sigma \cdot (t', a))|_{\Sigma_u}$, and $\sigma_e = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a = \Pi_\Sigma(\sigma|_{\Sigma_c}) \cdot a = \Pi_\Sigma((\sigma \cdot (t', a))|_{\Sigma_c})$. Thus, $P(\sigma \cdot (t', a))$ holds if $a \in \Sigma_c$.

Thus, $P(\sigma) \implies P(\sigma \cdot (t', a))$.

By induction on σ , for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Thus, for all $\sigma \in \text{tw}(\Sigma)$, for all $t \in \mathbb{R}_{\geq 0}$, if $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t)$ and $(\sigma, t) \notin \text{Pre}(\varphi)$, then $\sigma_s = \sigma|_{\Sigma_u}$, $\sigma_b = \epsilon$, and $\sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})$. \square

Lemma A.7. $\forall \sigma \in \Sigma_c^*, \forall a \in \Sigma_c, I(\sigma) \subseteq I(\sigma \cdot a)$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall a \in \Sigma_c, I(\sigma) \subseteq I(\sigma \cdot a)$ ”. Let us show by induction that $P(\sigma)$ holds for all $\sigma \in \Sigma_c^*$.

Induction basis: let us consider $a \in \Sigma_c$. Then, $I(\epsilon) = \emptyset \subseteq I(a)$.

Induction step: suppose now that for $\sigma \in \Sigma_c^*$, and for any $\sigma' \in \Sigma_c^*$, if $|\sigma'| \leq |\sigma|$, then $P(\sigma')$ holds. Let us then consider $a \in \Sigma_c$, $a' \in \Sigma_c$, and $(h, \sigma_0) \in \Sigma_c \times \Sigma_c^*$ such that $h \cdot \sigma_0 = \sigma \cdot a$ (h and σ_0 exist because $\sigma \cdot a \neq \epsilon$).

Then, $I(\sigma \cdot a \cdot a') = I(h \cdot \sigma_0 \cdot a') = \text{Pred}_h(S(\sigma_0 \cdot a') \cup I(\sigma_0 \cdot a'))$, and $I(\sigma \cdot a) = I(h \cdot \sigma_0) = \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0))$. Following the definition of S (Definition 3.17), $S(\sigma_0) \subseteq S(\sigma_0 \cdot a')$. Moreover, by induction hypothesis, since $|\sigma_0| \leq |\sigma|$, $P(\sigma_0)$ holds, meaning that $I(\sigma_0) \subseteq I(\sigma_0 \cdot a')$. Thus, $S(\sigma_0) \cup I(\sigma_0) \subseteq S(\sigma_0 \cdot a') \cup I(\sigma_0 \cdot a')$. It follows that $I(\sigma \cdot a) = \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0)) \subseteq \text{Pred}_h(S(\sigma_0 \cdot a') \cup I(\sigma_0 \cdot a')) = I(\sigma \cdot a \cdot a')$. Thus, for all $a' \in \Sigma_c$, $I(\sigma \cdot a) \subseteq I(\sigma \cdot a \cdot a')$, meaning that $P(\sigma \cdot a)$ holds.

Thus, $(\forall \sigma', |\sigma'| \leq |\sigma| \implies P(\sigma')) \implies P(\sigma \cdot a)$.

By induction on the size of σ , $P(\sigma)$ holds for every $\sigma \in \Sigma_c^*$, meaning that for all $\sigma \in \Sigma_c^*$, for all $a \in \Sigma_c$, $I(\sigma) \subseteq I(\sigma . a)$. \square

Lemma A.8. $\forall q \in Q, \forall \sigma \in \Sigma_c^*, (q \in S(\sigma)) \implies (\forall u \in \Sigma_u, q \text{ after } (0, u) \in S(\sigma) \cup I(\sigma))$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall q \in Q, (q \in S(\sigma)) \implies (\forall u \in \Sigma_u, q \text{ after } (0, u) \in S(\sigma) \cup I(\sigma))$ ”. Let us show by induction on σ that $P(\sigma)$ holds for every $\sigma \in \Sigma_c^*$.

Induction basis: let us consider $q \in S(\epsilon)$. Then, for any $u \in \Sigma_u$, since $(0, u) \in \text{tw}(\Sigma_u)$, considering the definition of $S(\epsilon)$, $q \text{ after } (0, u) \in S(\epsilon)$. Thus, $q \in S(\epsilon) \cup I(\epsilon)$. Thus, $P(\epsilon)$ holds.

Induction step: let us suppose that for $\sigma \in \Sigma_c^*$, $P(\sigma)$ holds. Let us consider $a \in \Sigma_c$ and $q \in S(\sigma . a)$. Then, considering the definition of $S(\sigma . a)$, two cases are possible:

- If $q \in S(\sigma)$, then, by induction hypothesis, for all $u \in \Sigma_u$, $q \text{ after } (0, u) \in S(\sigma) \cup I(\sigma)$. $S(\sigma) \subseteq S(\sigma . a)$, and following lemma A.7, $I(\sigma) \subseteq I(\sigma . a)$, thus, $q \text{ after } (0, u) \in S(\sigma . a) \cup I(\sigma . a)$.
- Otherwise, $q \in S(\sigma . a) \setminus S(\sigma)$, and then, considering the definition of S (Definition 3.17), $(S(\sigma . a) \setminus S(\sigma)) \cap \text{uPred}(\overline{(S(\sigma . a) \setminus S(\sigma)) \cup I(\sigma . a)}) = \emptyset$. Thus, if $u \in \Sigma_u$, $q \text{ after } (0, u) \in (S(\sigma . a) \setminus S(\sigma)) \cup I(\sigma . a) \subseteq S(\sigma . a) \cup I(\sigma . a)$.

In both cases, for all $u \in \Sigma_u$, $q \text{ after } (0, u) \in S(\sigma . a) \cup I(\sigma . a)$, meaning that $P(\sigma . a)$ holds.

Thus, for all $a \in \Sigma_c$, $P(\sigma) \implies P(\sigma . a)$.

Thus, by induction on σ , for all $\sigma \in \Sigma_c^*$, $P(\sigma)$ holds, meaning that for all $\sigma \in \Sigma_c^*$, for all $q \in S(\sigma)$, for all $u \in \Sigma_u$, $q \text{ after } (0, u) \in S(\sigma) \cup I(\sigma)$. \square

Lemma A.9. $\forall \sigma \in \Sigma_c^*, \forall q \in Q, (q \in S(\sigma) \cup I(\sigma)) \implies (G(q, \sigma) \neq \emptyset)$.

Proof. For $\sigma \in \Sigma_c^*$, let $P(\sigma)$ be the predicate “ $\forall q \in Q, (q \in S(\sigma) \cup I(\sigma)) \implies (G(q, \sigma) \neq \emptyset)$ ”. Let us then prove by induction on σ that $P(\sigma)$ holds for every $\sigma \in \Sigma_c^*$.

Induction basis: let us consider $q \in S(\epsilon) \cup I(\epsilon)$. Since $I(\epsilon) = \emptyset$, this means that $q \in S(\epsilon)$.

Following the definition of $S(\epsilon)$ (see Definition 3.17), since $\epsilon \in \text{tw}(\Sigma_u)$, this means that ϵ satisfies $\epsilon \preceq \Pi_\Sigma(\epsilon)$, $q \text{ after } \epsilon = q \in F_G$ (since $S(\epsilon) \subseteq F_G$), and for any $t \in \mathbb{R}_{\geq 0}$, $q \text{ after } (\epsilon, t) \in S(\epsilon)$. Thus, considering the definition of G (Definition 3.18), this means that $\epsilon \in G(q, \epsilon)$, thus $G(q, \epsilon) \neq \emptyset$.

Thus $P(\epsilon)$ holds.

Induction step: let us suppose that for $n \in \mathbb{N}$, for all $\sigma \in \Sigma_c^*$, $|\sigma| \leq n \implies P(\sigma)$. Let us consider $\sigma \in \Sigma_c^*$ such that $|\sigma| = n$, $a \in \Sigma_c$, and $q \in S(\sigma.a) \cup I(\sigma.a)$.

Then, we distinguish two cases, whether $q \in S(\sigma.a)$ or $q \in I(\sigma.a)$:

- If $q \in I(\sigma.a)$, let us consider $(h, \sigma_0) \in \Sigma_c \times \Sigma_c^*$ such that $\sigma.a = h.\sigma_0$. Then, $q \in I(h.\sigma_0) = \text{Pred}_h(S(\sigma_0) \cup I(\sigma_0))$, and since $|\sigma_0| = |\sigma| = n \leq n$, by induction hypothesis, $G(q \text{ after } (0, h), \sigma_0) \neq \emptyset$. Let us consider $w \in G(q \text{ after } (0, h), \sigma_0)$. Then, $(0, h).w$ satisfies $\Pi_\Sigma((0, h).w) \preceq h.\sigma_0$, $q \text{ after } ((0, h).w) = q \text{ after } (0, h) \text{ after } w \in F_G$, and for any $t \in \mathbb{R}_{\geq 0}$, $q \text{ after } ((0, h).w, t) = q \text{ after } (0, h) \text{ after } (w, t) \in S(\Pi_\Sigma(w)^{-1}.\sigma_0) = S(\Pi_\Sigma((0, h).w)^{-1}.(h.\sigma_0))$. Thus, $(0, h).w \in G(q, h.\sigma_0) = G(q, \sigma.a)$. Thus, $G(q, \sigma.a) \neq \emptyset$.
- If $q \in S(\sigma.a)$, then there are again two cases:
 - if $q \in S(\sigma)$, then by induction hypothesis, $G(q, \sigma) \neq \emptyset$. Since $G(q, \sigma) \subseteq G(q, \sigma.a)$, it follows that $G(q, \sigma.a) \neq \emptyset$.
 - Otherwise, $q \in X \cup Y$, where X and Y are defined in the definition of $S(\sigma.a)$ (Definition 3.17).
 - * If $q \in X$, then there exists $i \in I(\sigma.a)$ and $\delta \in \mathbb{R}_{\geq 0}$ such that $q \text{ after } (\epsilon, \delta) = i$, and for all $t \leq \delta$, $q \text{ after } (\epsilon, t) \in X \subseteq S(\sigma.a)$. Since $i \in I(\sigma.a)$, we showed previously that $G(i, \sigma.a) \neq \epsilon$. Let us consider $w \in G(i, \sigma.a)$. Then, $w +_t \delta$ satisfies $\Pi_\Sigma(w +_t \delta) \preceq \sigma.a$, $q \text{ after } (w +_t \delta) = i \text{ after } w \in F_G$, and for all $t \in \mathbb{R}_{\geq 0}$, if $t < \delta$, then $q \text{ after } (w +_t \delta, t) = q \text{ after } (\epsilon, t) \in X \subseteq S(\sigma.a)$, otherwise, $q \text{ after } (w +_t \delta, t) = i \text{ after } (w, t - \delta) \in S(\sigma.a)$. Thus, $w +_t \delta \in G(q, \sigma.a)$. Thus, $G(q, \sigma.a) \neq \emptyset$.
 - * Otherwise, $q \in Y$, and then ϵ satisfies $\Pi_\Sigma(\epsilon) \preceq \sigma.a$, $q \text{ after } \epsilon \in F_G$, and for all $t \in \mathbb{R}_{\geq 0}$, $q \text{ after } (\epsilon, t) \in \text{up}(q) \subseteq \text{up}(Y) = Y \subseteq S(\sigma.a)$. Thus, $\epsilon \in G(q, \sigma.a)$. Thus, $G(q, \sigma.a) \neq \emptyset$.

Thus, for all $q \in S(\sigma.a) \cup I(\sigma.a)$, $G(q, \sigma.a) \neq \emptyset$, meaning that $P(\sigma.a)$ holds.

Thus, $P(\sigma) \implies P(\sigma.a)$.

By induction on σ , $P(\sigma)$ holds for every $\sigma \in \Sigma_c^*$, meaning that for all $\sigma \in \Sigma_c^*$, for all $q \in S(\sigma) \cup I(\sigma)$, $G(q, \sigma) \neq \emptyset$. \square

Proposition 3.7. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$ as per Definition 3.14.

Proof. Notation from Definition 3.19 is to be used in this proof: for $q \in Q$ and $w \in \Sigma_c^*$,

$$\begin{aligned}
 \kappa_\varphi(q, w) &= \min_{\leq_{\text{lex}}}(\max_{\preceq}(\text{G}(q, w) \cup \{\epsilon\})), \\
 \text{buf}_c &= \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c, \\
 t_1 &= \min(\{t'' \in \mathbb{R}_{\geq 0} \mid t'' \geq t' \wedge \\
 &\quad \text{G}(\text{Reach}(\sigma_s \cdot (t', a), t''), \text{buf}_c) \neq \emptyset\} \cup \{+\infty\}), \\
 \sigma'_b &= \kappa_\varphi(\text{Reach}(\sigma_s \cdot (t', a), \min(\{t, t_1\})), \text{buf}_c) +_t \min(\{t, t_1\}), \\
 \sigma'_c &= \Pi_\Sigma(\sigma'_b)^{-1} \cdot \text{buf}_c, \\
 t_2 &= \min(\{t'' \in \mathbb{R}_{\geq 0} \mid t'' \geq t' \wedge \\
 &\quad \text{G}(\text{Reach}(\sigma_s, t''), \text{buf}_c \cdot a) \neq \emptyset\} \cup \{+\infty\}), \\
 \sigma''_b &= \kappa_\varphi(\text{Reach}(\sigma_s, \min(\{t, t_2\})), \text{buf}_c \cdot a) +_t \min(\{t, t_2\}), \\
 \sigma''_c &= \Pi_\Sigma(\sigma''_b)^{-1} \cdot (\text{buf}_c \cdot a).
 \end{aligned}$$

We have to prove that for any $\sigma \in \text{tw}(\Sigma)$, for any $t \in \mathbb{R}_{\geq 0}$, $(\sigma, t) \in \text{Pre}(\varphi) \implies E_\varphi(\sigma, t) \models \varphi$.

For $\sigma \in \text{tw}(\Sigma)$, and $t \geq \text{time}(\sigma)$, let $P(\sigma, t)$ be the predicate “ $(\sigma \in \text{Pre}(\varphi, t) \wedge (\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t)) \implies (E_\varphi(\sigma) \models \varphi \wedge \text{nobs}(\sigma_b, t) -_t t \in \text{G}(\text{Reach}(\sigma_s, t), \Pi_\Sigma(\text{nobs}(\sigma_b, t)) \cdot \sigma_c))$ ”. Let also $P(\sigma)$ be the predicate: “ $\forall t \geq \text{time}(\sigma), P(\sigma, t)$ ”. Let us show by induction that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis: for $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$. We consider two cases:

- if $\epsilon \notin \text{Pre}(\varphi, t)$, then $P(\epsilon)$ trivially holds.
- Otherwise, $\epsilon \in \text{Pre}(\varphi, t)$, and then, following Definition 3.20, there exists $t' \leq t$ such that $\text{G}(\text{Reach}(\text{obs}(\epsilon, t')|_{\Sigma_u}, t'), \epsilon) \neq \emptyset$, meaning that $\text{G}(\text{Reach}(\epsilon, t'), \epsilon) \neq \emptyset$. Thus, following the definition of $\text{G}(\text{Reach}(\epsilon, t'), \epsilon)$, (Definition 3.18), $\epsilon \in \text{G}(\text{Reach}(\epsilon, t'), \epsilon)$, and $\text{Reach}(\epsilon) \in F_G$. Since $E_\varphi(\epsilon) = \epsilon$, and $\text{Reach}(\epsilon) \in F_G$, $E_\varphi(\epsilon) \models \varphi$. Thus, because $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon, \epsilon)$, $P(\epsilon, t)$ holds.

Thus, in both cases, $P(\epsilon, t)$ holds, meaning that $P(\epsilon)$ holds.

Induction step: suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider (t', a) such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$, and $t \geq t' = \text{time}(\sigma \cdot (t', a))$. Let us also consider $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$ and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t', a), t)$.

Then, we distinguish three cases:

- If $\sigma \cdot (t', a) \notin \text{Pre}(\varphi, t)$, then, $P(\sigma \cdot (t', a), t)$ trivially holds.

- If $\sigma.(t', a) \in \text{Pre}(\varphi, t) \wedge \sigma \notin \text{Pre}(\varphi, t')$, then since $\sigma \notin \text{Pre}(\varphi, t')$, following lemma A.6, $\sigma_s = \sigma|_{\Sigma_u}$, $\sigma_b = \epsilon$, and $\sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})$.

Since $\sigma.(t', a) \in \text{Pre}(\varphi, t)$, and $\sigma \notin \text{Pre}(\varphi, t')$, following Definition 3.20, there exists $t'' \in \mathbb{R}_{\geq 0}$ such that $t' \leq t'' \leq t$, and $G(\text{Reach}(\text{obs}(\sigma.(t', a), t'')|_{\Sigma_u}, t''), \Pi_\Sigma(\text{obs}(\sigma.(t', a), t'')|_{\Sigma_c})) \neq \emptyset$. Since $t'' \geq t' = \text{time}(\sigma.(t', a))$, $\text{obs}(\sigma.(t', a), t'') = \sigma.(t', a)$. Thus:

$$G(\text{Reach}((\sigma.(t', a))|_{\Sigma_u}, t''), \Pi_\Sigma((\sigma.(t', a))|_{\Sigma_c})) \neq \emptyset. \quad (\text{A.1})$$

- If $a \in \Sigma_u$, then considering that $(\sigma.(t', a))|_{\Sigma_u} = \sigma|_{\Sigma_u}.(t', a) = \sigma_s.(t', a)$, $\sigma_b = \epsilon$, and $\sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})$, (A.1) becomes:

$$G(\text{Reach}(\sigma_s.(t', a), t''), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c) \neq \emptyset.$$

Thus, $t_1 \leq t'' \leq t$, meaning that $\sigma_d -_t t_1 \in G(\text{Reach}(\sigma_s.(t', a), t_1), \Pi_\Sigma(\sigma_b). \sigma_c)$. Thus, considering the definition of G (Definition 3.18), it follows that $\text{nobs}(\sigma_d, t) -_t t \in G(\text{Reach}(\sigma_s.(t', a). \text{obs}(\sigma_d, t), t), \Pi_\Sigma(\text{obs}(\sigma_d, t))^{-1} . (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c))$.

Moreover, $\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c = \sigma|_{\Sigma_c}$, thus $\Pi_\Sigma(\text{obs}(\sigma_d, t))^{-1} . (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c) = \Pi_\Sigma(\text{nobs}(\sigma_d, t)) . \sigma_e$, meaning that $\text{nobs}(\sigma_d, t) -_t t \in G(\text{Reach}(\sigma_t, t), \Pi_\Sigma(\text{nobs}(\sigma_d, t)) . \sigma_e)$.

Thus, $P(\sigma.(t', a), t)$ holds.

- Otherwise, $a \in \Sigma_c$. Then, $(\sigma.(t', a))|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$, $\sigma_b = \epsilon$, and $\sigma_c = \Pi_\Sigma((\sigma.(t', a))|_{\Sigma_c}) = \Pi_\Sigma(\sigma|_{\Sigma_c}).a$. This means that (A.1) becomes:

$$G(\text{Reach}(\sigma_s, t''), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c . a) \neq \emptyset.$$

Thus, $t_2 \leq t'' \leq t$, therefore $\sigma_d -_t t_2 \in G(\text{Reach}(\sigma_s, t_2), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c . a)$. It follows that $\text{nobs}(\sigma_d, t) -_t t \in G(\text{Reach}(\sigma_s . \text{obs}(\sigma_d, t), t), \Pi_\Sigma(\text{obs}(\sigma_d, t))^{-1} . (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c . a))$.

Moreover, $\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c . a = \Pi_\Sigma((\sigma.(t', a))|_{\Sigma_c}) = \Pi_\Sigma(\sigma_d) . \sigma_e$. Thus, $\Pi_\Sigma(\text{obs}(\sigma_d, t))^{-1} . (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c . a) = \Pi_\Sigma(\text{nobs}(\sigma_d, t)) . \sigma_e$. Thus, $\text{nobs}(\sigma_d, t) -_t t \in G(\text{Reach}(\sigma_t, t), \Pi_\Sigma(\text{nobs}(\sigma_d, t)) . \sigma_e)$. This means that $P(\sigma.(t', a), t)$ holds.

Thus, if $\sigma.(t', a) \in \text{Pre}(\varphi, t)$ and $\sigma \notin \text{Pre}(\varphi, t')$, $P(\sigma, t) \implies P(\sigma.(t', a), t)$.

- If $\sigma.(t', a) \in \text{Pre}(\varphi, t)$ and $\sigma \in \text{Pre}(\varphi, t')$, then, let us consider $w_b = \text{nobs}(\sigma_b, t') -_t t'$. By induction hypothesis, since $\sigma \in \text{Pre}(\varphi, t')$, we know that $E_\varphi(\sigma) \models \varphi$, and $w_b \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) . \sigma_c)$.

- If $a \in \Sigma_u$, then, since $w_b \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$, $\text{Reach}(\sigma_s, t')$ after $(w_b, 0) = \text{Reach}(\sigma_s, t') \in S(\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$. Thus, following lemma A.8, since $a \in \Sigma_u$, $\text{Reach}(\sigma_s, t')$ after $(0, a) = \text{Reach}(\sigma_s \cdot (t', a)) \in S(\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c) \cup I(\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$. Then, following lemma A.9, this means that $G(\text{Reach}(\sigma_s \cdot (t', a)), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c) \neq \emptyset$.

It follows that $t_1 = t'$, thus $\min(\{t, t_1\}) = t_1 = t'$, and $\sigma_d \dashv_t t' \in G(\text{Reach}(\sigma_s \cdot (t', a), t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$. This implies that $\text{Reach}(\sigma_s \cdot (t', a) \cdot \sigma_d) = \text{Reach}(E_\varphi(\sigma \cdot (t', a))) \in F_G$, meaning that $E_\varphi(\sigma \cdot (t', a)) \models \varphi$. Moreover, following the definition of G (Definition 3.18), $\text{nobs}(\sigma_d, t) \dashv_t t \in G(\text{Reach}(\sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t), t), \Pi_\Sigma(\text{obs}(\sigma_d, t))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c))$. Thus, since $\sigma_t = \sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t)$, and $\Pi_\Sigma(\sigma_d) \cdot \sigma_e = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c$, it follows that $\text{nobs}(\sigma_d, t) \dashv_t t \in G(\text{Reach}(\sigma_t, t), \Pi_\Sigma(\text{nobs}(\sigma_d, t)) \cdot \sigma_e)$. This means that $P(\sigma \cdot (t', a), t)$ holds.

- Otherwise, $a \in \Sigma_c$. Since $w_b \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$, w_b satisfies $\Pi_\Sigma(w_b) \preceq \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \preceq \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a$, $\text{Reach}(\sigma_s, t')$ after $w_b \in F_G$, and for all $t'' \in \mathbb{R}_{\geq 0}$, $\text{Reach}(\sigma_s, t')$ after $(w_b, t'') \in S(\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c))$.

Since $\Pi_\Sigma(w_b) \preceq \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \preceq \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a$, it follows that, for any $t'' \in \mathbb{R}_{\geq 0}$, $\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a) = (\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)) \cdot a$. Thus, $S(\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)) \subseteq S(\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a))$. Thus for all $t'' \in \mathbb{R}_{\geq 0}$, $\text{Reach}(\sigma_s, t')$ after $(w_b, t'') \in S(\Pi_\Sigma(\text{obs}(w_b, t''))^{-1} \cdot (\Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a))$.

This means that $w_b \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a)$. It follows that $G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a) \neq \emptyset$, and thus, using the same reasoning as in the case where $a \in \Sigma_u$, $t_2 = t'$, and σ_d is such that $\text{Reach}(\sigma_s, t')$ after $\sigma_d \in F_G$, meaning that $E_\varphi(\sigma \cdot (t', a)) \models \varphi$, and $\text{nobs}(\sigma_d, t) \dashv_t t \in G(\text{Reach}(\sigma_t, t), \Pi_\Sigma(\text{nobs}(\sigma_d, t)) \cdot \sigma_e)$. Thus, $P(\sigma \cdot (t', a), t)$ holds.

Thus, in all cases, for all $t \geq t'$, $P(\sigma) \implies P(\sigma \cdot (t', a), t)$. This means that $P(\sigma) \implies \forall t \geq t', P(\sigma \cdot (t', a), t)$. Thus, for all $t' \geq \text{time}(\sigma)$, for all $a \in \Sigma$, $P(\sigma) \implies P(\sigma \cdot (t', a))$.

Thus, by induction on σ , for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. In particular, for all $(\sigma, t) \in \text{Pre}(\varphi)$, $E_\varphi(\sigma) \models \varphi$. This means that E_φ is sound in $\text{Pre}(\varphi)$. \square

Proposition 3.8. E_φ is compliant, as per Definition 3.15.

Proof. We have to prove the three following properties:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, E_\varphi(\sigma, t) \preceq_{d\Sigma_c} \sigma$

2. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, E_\varphi(\sigma, t) =_{\Sigma_u} \text{obs}(\sigma, t)$
3. $\forall \sigma \in \text{tw}(\Sigma), \forall (t, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u, \sigma \cdot (t, u) \in \text{tw}(\Sigma) \implies E_\varphi(\sigma, t) \cdot (t, u) \preceq E_\varphi(\sigma \cdot (t, u), t)$.

We start by proving items 1 and 2.

For $\sigma \in \text{tw}(\Sigma)$, let $P(\sigma)$ be the predicate: “ $\forall t \geq \text{time}(\sigma), (\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t) \implies \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u} \wedge \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \text{nobs}(\sigma_b, t)) \cdot \sigma_c = \Pi_\Sigma(\sigma_{|\Sigma_c}) \wedge \sigma_{s|\Sigma_c} \preceq_d \sigma_{|\Sigma_c}$ ”. Let us prove by induction that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis: for $\sigma = \epsilon$. $\text{store}_\varphi(\epsilon) = (\epsilon, \epsilon, \epsilon)$, and $\epsilon_{|\Sigma_c} = \epsilon_{|\Sigma_u} = \Pi_\Sigma(\epsilon) = \epsilon$. Thus, $P(\epsilon)$ trivially holds.

Induction step: suppose now that for some $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider (t', a) such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$, $t \geq t' = \text{time}(\sigma \cdot (t', a))$, $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$, and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t', a), t)$. Then, by induction hypothesis, $\sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u}$, $\Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \text{nobs}(\sigma_b, t')) \cdot \sigma_c = \Pi_\Sigma(\sigma_{|\Sigma_c})$, and $\sigma_{s|\Sigma_c} \preceq_d \sigma_{|\Sigma_c}$.

- If $a \in \Sigma_u$, then, by construction, σ_d satisfies $\Pi_\Sigma(\sigma_d) \preceq \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c$ and $\sigma_d \neq \epsilon \implies \text{date}(\sigma_d(1)) \geq t'$.
 - *Projection on Σ_u :* Since $a \in \Sigma_u$, $\sigma_{t|\Sigma_u} = (\sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t))_{|\Sigma_u}$. Since $\sigma_d \in \text{tw}(\Sigma_c)$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u} \cdot (t', a) = \sigma_{|\Sigma_u} \cdot (t', a) = (\sigma \cdot (t', a))_{|\Sigma_u}$.
 - *Projection on Σ_c :* $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \text{nobs}(\sigma_d, t)) \cdot \sigma_e = \Pi_\Sigma((\sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t))_{|\Sigma_c} \cdot \text{nobs}(\sigma_d, t)) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \sigma_d) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c}) \cdot \Pi_\Sigma(\sigma_d) \cdot \sigma_e$. By construction, $\Pi_\Sigma(\sigma_d) \cdot \sigma_e = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c$. Thus, $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \sigma_d) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c}) \cdot \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c = \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \text{nobs}(\sigma_b, t')) \cdot \sigma_c = \Pi_\Sigma(\sigma_{|\Sigma_c}) = \Pi_\Sigma((\sigma \cdot (t', a))_{|\Sigma_c})$. Moreover, $\sigma_t \in \text{tw}(\Sigma)$, and since $\sigma_t = \sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t)$, it follows that for all $i \in [1; |\text{obs}(\sigma_d, t)|]$, $\text{date}(\sigma_d(i)) \geq t'$. Since $\sigma_{s|\Sigma_c} \preceq_d \sigma_{|\Sigma_c}$, for all $i \in [1; |\sigma_{s|\Sigma_c}|]$, $\text{date}(\sigma_{s|\Sigma_c}(i)) \geq \text{date}(\sigma_{|\Sigma_c}(i))$. Thus, for all $i \in [1; |\sigma_{t|\Sigma_c}|]$, $\text{date}(\sigma_{t|\Sigma_c}(i)) \geq \text{date}(\sigma_{|\Sigma_c}(i))$. Since $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \sigma_d) \cdot \sigma_e = \Pi_\Sigma(\sigma_{|\Sigma_c})$, $\Pi_\Sigma(\sigma_{t|\Sigma_c}) \preceq \Pi_\Sigma(\sigma_{|\Sigma_c})$. Thus $\sigma_{t|\Sigma_c} \preceq_d \sigma_{|\Sigma_c} = (\sigma \cdot (t', a))_{|\Sigma_c}$.

This means that if $a \in \Sigma_u$, $P(\sigma \cdot (t', a))$ holds.

- If $a \in \Sigma_c$, then, by construction, σ_d satisfies $\Pi_\Sigma(\sigma_d) \preceq \Pi_\Sigma(\sigma_b) \cdot \sigma_c \cdot a$, and $\sigma_d \neq \epsilon \implies \text{date}(\sigma_d(1)) \geq t'$.
 - *Projection on Σ_u :* $\sigma_{t|\Sigma_u} = (\sigma_s \cdot \text{obs}(\sigma_d, t))_{|\Sigma_u}$. Since $\sigma_d \in \text{tw}(\Sigma_c)$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u} = (\sigma \cdot (t', a))_{|\Sigma_u}$.
 - *Projection on Σ_c :* $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \text{nobs}(\sigma_d, t)) \cdot \sigma_e = \Pi_\Sigma((\sigma_s \cdot \text{obs}(\sigma_d, t))_{|\Sigma_c} \cdot \text{nobs}(\sigma_d, t)) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \sigma_d) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c}) \cdot \Pi_\Sigma(\sigma_d) \cdot \sigma_e$. By construction, it is ensured that $\Pi_\Sigma(\sigma_d) \cdot \sigma_e = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a$.

It follows that $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \sigma_d) \cdot \sigma_e = \Pi_\Sigma(\sigma_{s|\Sigma_c}) \cdot \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a = \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a = \Pi_\Sigma(\sigma_{|\Sigma_c}) \cdot a = \Pi_\Sigma((\sigma \cdot (t', a))_{|\Sigma_c})$.

Moreover, considering t_2 as defined in Definition 3.19, $t_2 \geq t'$, and $t \geq t'$, thus $\min(\{t, t_2\}) \geq t'$, which means that since there exists $w_d \in \text{tw}(\Sigma)$ such that $\sigma_d = w_d +_t \min(\{t, t_2\})$, if $\sigma_d \neq \epsilon$, then $\text{date}(\sigma_d(1)) \geq t'$. Thus, for all $i \in [1; |\sigma_d|]$, $\text{date}(\sigma_d(i)) \geq t' = \text{time}(\sigma \cdot (t', a))$. This still holds if $\sigma_d = \epsilon$, because then $[1; |\sigma_d|] = \emptyset$. Since $\sigma_{s|\Sigma_c} \preceq_d \sigma_{|\Sigma_c}$, for all $i \in [1; |\sigma_{s|\Sigma_c}|]$, $\text{date}(\sigma_{s|\Sigma_c}(i)) \geq \text{date}(\sigma_{|\Sigma_c}(i))$. Thus, for all $i \in [1; |\sigma_{t|\Sigma_c}|]$, $\text{date}(\sigma_{t|\Sigma_c}(i)) \geq \text{date}((\sigma \cdot (t', a))_{|\Sigma_c}(i))$. Since $\Pi_\Sigma(\sigma_{t|\Sigma_c} \cdot \text{nobs}(\sigma_d, t)) \cdot \sigma_e = \Pi_\Sigma((\sigma \cdot (t', a))_{|\Sigma_c})$, $\Pi_\Sigma(\sigma_{t|\Sigma_c}) \preceq_d \Pi_\Sigma((\sigma \cdot (t', a))_{|\Sigma_c})$. Thus $\sigma_{t|\Sigma_c} \preceq_d (\sigma \cdot (t', a))_{|\Sigma_c}$.

Thus if $a \in \Sigma_c$, $P(\sigma \cdot (t, a))$ holds.

Thus for any $a \in \Sigma$ and $t \geq \text{time}(\sigma)$, $P(\sigma) \implies P(\sigma \cdot (t, a))$.

Thus, by induction on σ , for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds, meaning that for all $t \geq \text{time}(\sigma)$, $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t) \implies \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u} \wedge \Pi_\Sigma(\sigma_{s|\Sigma_c} \cdot \text{nobs}(\sigma_b, t)) \cdot \sigma_c = \Pi_\Sigma(\sigma_{|\Sigma_c}) \wedge \sigma_{s|\Sigma_c} \preceq_d \sigma_{\Sigma_c}$. Since $E_\varphi(\sigma) = \sigma_s$, this means that $E_\varphi(\sigma) \preceq_{d\Sigma_c} \sigma$ and $E_\varphi(\sigma)_{|\Sigma_u} = \sigma_{|\Sigma_u}$.

Now what remains to be proved is item 3.

Let us consider $\sigma \in \text{tw}(\Sigma)$, and $(t, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u$ such that $\sigma \cdot (t, u) \in \text{tw}(\Sigma)$. Then, considering the definition of store_φ (Definition 3.19), $E_\varphi(\sigma \cdot (t, u), t) = E_\varphi(\sigma, t) \cdot (t, u) \cdot \text{obs}(\sigma'_s, t)$, where σ'_s is defined in Definition 3.19. This means that $E_\varphi(\sigma, t) \cdot (t, u) \preceq E_\varphi(\sigma \cdot (t, u))$.

Thus, items 1 to 3 hold. Thus E_φ is compliant with respect to Σ_u and Σ_c . \square

Lemma A.10. $\forall \sigma \in \Sigma_c^*, \forall q \in Q, (q \notin S(\sigma)) \implies (\exists \sigma_u \in \text{tw}(\Sigma_u), (q \text{ after } \sigma_u \notin F_G) \wedge (\forall t > 0, q \text{ after } (\sigma_u, t) \notin S(\sigma) \cup I(\sigma)) \wedge (\forall \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma)))$

Proof. For $\sigma \in \Sigma_c^*$ and $q \in Q$, let $P(\sigma, q)$ be the predicate “ $\forall \sigma_u \in \text{tw}(\Sigma_u), (q \text{ after } \sigma_u \in F_G) \vee (\exists t > 0, q \text{ after } (\sigma_u, t) \in S(\sigma) \cup I(\sigma)) \vee (\exists \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \wedge q \text{ after } \sigma'_u \in S(\sigma) \cup I(\sigma))$ ”. Let us show the contrapositive of the proposition, that is that for all $\sigma \in \Sigma_c^*$, for all $q \in Q$, $(P(\sigma, q)) \implies (q \in S(\sigma))$.

- If $\sigma = \epsilon$, let us consider $q \in Q$ such that $P(\epsilon, q)$ holds. Then, since $\epsilon \in \text{tw}(\Sigma_u)$, $q \text{ after } \epsilon = q \in F_G$, or there exists $t > 0$ such that $q \text{ after } (\epsilon, t) \in S(\epsilon) \cup I(\epsilon)$, or there exists $\sigma'_u \preceq \epsilon$ such that $\sigma'_u \neq \epsilon$ and $q \text{ after } \sigma'_u \in S(\epsilon) \cup I(\epsilon)$. Since $\sigma'_u \preceq \epsilon$, $\sigma'_u = \epsilon$, meaning that this last condition does not hold for $\sigma_u = \epsilon$. Thus, $q \in F_G$ or there exists $t \in \mathbb{R}_{\geq 0}$

such that q after $(\epsilon, t) \in S(\epsilon) \cup I(\epsilon)$. Since $I(\epsilon) = \emptyset$ and $S(\epsilon) \subseteq F_G$, if the second condition holds, then q after $(\epsilon, t) \in F_G$, meaning that $q \in F_G$. Thus, $q \in F_G$.

Moreover, since $P(\epsilon, q)$ holds, for any $\sigma_u \in \text{tw}(\Sigma_u)$, q after $\sigma_u \in F_G$ or there exists $t \in \mathbb{R}_{\geq 0}$ such that q after $(\sigma_u, t) \in S(\epsilon) \cup I(\epsilon) \subseteq F_G$, meaning that q after $\sigma_u \in F_G$, or there exists $\sigma'_u \preceq \sigma_u$ such that q after $\sigma'_u \in S(\epsilon) \cup I(\epsilon)$. If the last condition holds, since $I(\epsilon) = \emptyset$, then q after $\sigma'_u \in S(\epsilon)$. Then, following the definition of $S(\epsilon)$, since $\sigma'_u{}^{-1} \cdot \sigma_u \in \text{tw}(\Sigma_u)$, it follows that q after σ'_u after $\sigma'_u{}^{-1} \cdot \sigma_u = q$ after $\sigma_u \in F_G$. Thus, for all $\sigma_u \in \text{tw}(\Sigma_u)$, q after $\sigma_u \in F_G$, meaning that $q \in S(\epsilon)$.

- If $\sigma \neq \epsilon$, there exists $(\sigma', a) \in \Sigma_c^* \times \Sigma_c$ such that $\sigma = \sigma' . a$. Let us consider $q \in Q$ such that $P(\sigma, q)$ holds. Then, for all $\sigma_u \in \text{tw}(\Sigma_u)$, q after $\sigma_u \in F_G$, or there exists $t > 0$ such that q after $(\sigma_u, t) \in S(\sigma) \cup I(\sigma)$, or there exists $\sigma'_u \preceq \sigma_u$ such that $\sigma'_u \neq \epsilon$ and q after $\sigma'_u \in S(\sigma) \cup I(\sigma)$. Let X_s and Y_s be such that $S(\sigma) = S(\sigma' . a) = S(\sigma') \cup X_s \cup Y_s$, with:

- $\forall x \in X_s, \exists i \in I(\sigma' . a), \exists \delta \in \mathbb{R}_{\geq 0}, x$ after $(\epsilon, \delta) = i \wedge \forall t \leq \delta, x$ after $(\epsilon, t) \in X_s$,
- $Y_s \subseteq F_G \wedge \text{up}(Y_s) = Y_s$, and
- $(X_s \cup Y_s) \cap \text{uPred}(\overline{X_s \cup Y_s \cup I(\sigma' . a)}) = \emptyset$.

X_s and Y_s correspond to the sets X and Y in the definition of $S(\sigma' . a)$, respectively. Let us consider $X_0 = \{q \text{ after } (\sigma_u, t) \mid \sigma_u \in \text{tw}(\Sigma_u) \wedge t \in \mathbb{R}_{\geq 0} \wedge \forall t' \in]0; t], q \text{ after } (\sigma_u, t') \notin S(\sigma) \cup I(\sigma) \wedge \forall \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma)\}$, and $Y_0 = \{y \in X_0 \mid \text{up}(y) \subseteq X_0 \cup Y_s\}$. Then, $Y_0 \subseteq X_0$, and $\text{up}(Y_0) = Y_0$. Moreover, if $y \in Y_0$, then $\text{up}(y) \subseteq X_0 \cup Y_s$, and more precisely, $\text{up}(y) \subseteq Y_0 \cup Y_s$, since all the states in $\text{up}(y)$ are also in Y_0 if $y \in Y_0$. Since $\text{up}(Y_s) = Y_s$, either $\text{up}(y) \subseteq Y_0$ or there exists $t \in \mathbb{R}_{\geq 0}$ such that for all $t' < t$, y after $(\epsilon, t') \in X_0$ and $\text{up}(y \text{ after } (\epsilon, t)) \subseteq Y_s$. Since $P(\sigma, q)$ holds, and $Y_s \subseteq F_G$, in both cases, $y \in F_G$, meaning that $Y_0 \subseteq F_G$. Let us now consider $Y = Y_s \cup Y_0$, $X = X_s \cup (X_0 \setminus Y_0)$, and $x \in X$. Let us suppose that $x \notin X_s$, meaning that $x \in X_0 \setminus Y_0$. Following the definition of X_0 and Y_0 , this means that there exists $\delta > 0$ and $i \in S(\sigma) \cup I(\sigma)$ such that x after $(\epsilon, \delta) = i$, and they can be chosen such that for all $t < \delta$, x after $(\epsilon, t) \in X_0$. Suppose now that $i \in S(\sigma)$, and more precisely that $i \in Y_s$. Then, $\text{up}(i) \subseteq Y_s$ and $\text{up}(i) \cap \text{uPred}(\overline{X_s \cup Y_s \cup I(\sigma)}) = \emptyset$, and since for all $t < \delta$, x after $(\epsilon, t) \in X_0$, it follows that $\text{up}(x) \subseteq X_0 \cup Y_s$, meaning that $x \in Y_0$, which is absurd. Thus, $i \notin Y_s$. This means that either $i \in I(\sigma)$, or $i \in X_s$. Thus, there exists $\delta' \in \mathbb{R}_{\geq 0}$ such that i after $(\epsilon, \delta') \in I(\sigma)$ and for all $t < \delta'$, i after $(\epsilon, t) \in X_s \subseteq X$ (if $i \in I(\sigma)$, then $\delta' = 0$). Then, x after $(\epsilon, \delta + \delta') = i$, and for all $t < \delta + \delta'$, x after $(\epsilon, t) \in X$. Moreover,

$(X \cup Y) \cap \text{uPred}(\overline{X \cup Y \cup I(\sigma)}) = \emptyset$ since $Y = Y_s \cup Y_0 \subseteq S(\sigma) \cup X_0$, $X \subseteq X_s \cup X_0$, and $X \cup Y = X_0 \cup S(\sigma)$. This means that $X \cup Y \subseteq S(\sigma)$, and since $X_0 \subseteq X \cup Y$, $X_0 \subseteq S(\sigma)$. Since $q = q$ after $(\epsilon, 0)$, with $\epsilon \in \text{tw}(\Sigma_u)$ and $t \in \mathbb{R}_{\geq 0}$, $q \in X_0$, and thus $q \in S(\sigma)$. Thus, if $\sigma \neq \epsilon$ and $q \in Q$, $P(\sigma, q) \implies q \in S(\sigma)$.

Thus, for all $\sigma \in \Sigma_c^*$, for all $q \in Q$, $P(\sigma, q) \implies q \in S(\sigma)$. Thus, the contrapositive also holds, meaning that for all $\sigma \in \Sigma_c^*$, for all $q \in Q$, $q \notin S(\sigma) \implies \neg P(\sigma, q)$, that is $q \notin S(\sigma) \implies (\exists \sigma_u \in \text{tw}(\Sigma_u), q \text{ after } \sigma_u \notin F_G \wedge \forall t > 0, q \text{ after } (\sigma_u, t) \notin S(\sigma) \cup I(\sigma) \wedge \forall \sigma'_u \preceq \sigma_u, \sigma'_u \neq \epsilon \implies q \text{ after } \sigma'_u \notin S(\sigma) \cup I(\sigma))$. \square

Proposition 3.9. E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 3.16.

Proof. Let us consider $E' : \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \text{tw}(\Sigma)$, that is compliant with respect to Σ_c and Σ_u . Let us also consider $\sigma \in \text{tw}(\Sigma)$, and (t', a) such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$. Suppose now that $(\sigma, t') \in \text{Pre}(\varphi)$, $E'(\sigma, t') = E_\varphi(\sigma, t')$, and that $E_\varphi(\sigma \cdot (t', a)) \prec_d E'(\sigma \cdot (t', a))$.

We then have to show that there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that $E'(\sigma \cdot (t', a) \cdot \sigma_u) \not\models \varphi$.

Let us consider $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$, and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\sigma \cdot (t', a), t)$, where t is such that $\sigma_t = E_\varphi(\sigma \cdot (t', a))$ (i.e. t is sufficiently big). Then, considering proof of soundness (appendix A.1.2), since $(\sigma, t') \in \text{Pre}(\varphi)$, $\text{nobs}(\sigma_b, t') \dashv_t t' \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$.

- If $a \in \Sigma_u$, this means that $\sigma_d \dashv_t t' \in G(\text{Reach}(\sigma_s \cdot (t', a)), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$. Let us consider $q = \text{Reach}(\sigma_s \cdot (t', a))$, and $\text{buf}_c = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c$. Then, $\sigma_d \dashv_t t' = \min_{\text{lex}}(\max_{\preceq}(G(q, \text{buf}_c)))$. E' is compliant with respect to Σ_u and Σ_c , thus, since $E_\varphi(\sigma, t') = E'(\sigma, t')$, there exists $\sigma_{d2} \in \text{tw}(\Sigma)$ such that $E'(\sigma \cdot (t', a)) = \sigma_s \cdot (t', a) \cdot \sigma_{d2}$. Since $E_\varphi(\sigma \cdot (t', a)) \prec_d E'(\sigma \cdot (t', a))$, then $\sigma_d \prec_d \sigma_{d2}$, thus $\sigma_d \dashv_t t' \prec_d \sigma_{d2} \dashv_t t' = w_{d2}$, meaning that $w_{d2} \notin G(q, \text{buf}_c)$. Then, following the definitions of G and S , there are several cases:

- $\Pi_\Sigma(w_{d2}) \not\preceq \text{buf}_c$. Since E' is compliant, and $E'(\sigma) = E_\varphi(\sigma)$, this is not possible.
- $q \text{ after } w_{d2} \notin F_G$, meaning that $E'(\sigma \cdot (t', a)) \not\models \varphi$.
- Or there exists $t'' \in \mathbb{R}_{\geq 0}$ such that $q \text{ after } (w_{d2}, t'') \notin S(\Pi_\Sigma(\text{obs}(w_{d2}, t''))^{-1} \cdot \text{buf}_c)$. Let us then note $\text{buf}_{c2} = \Pi_\Sigma(\text{obs}(w_{d2}, t''))^{-1} \cdot \text{buf}_c$, and $q_2 = q \text{ after } (w_{d2}, t'')$. Then, following lemma A.10, there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that $q_2 \text{ after } \sigma_u \notin F_G$, for all $t_0 > 0$, $q_2 \text{ after } (\sigma_u, t_0) \notin S(\text{buf}_{c2}) \cup I(\text{buf}_{c2})$, and for all $\sigma'_u \preceq \sigma_u$, $\sigma'_u \neq \epsilon \implies q_2 \text{ after } \sigma'_u \notin S(\text{buf}_{c2}) \cup I(\text{buf}_{c2})$.

Then, considering that E' is compliant, either $E'(\sigma \cdot (t', a) \cdot (\sigma_u +_t (t' + t'')))) = \sigma_s \cdot (t', a) \cdot \text{obs}(w_{d2} +_t t', t'') \cdot (\sigma_u +_t (t' + t''))$, meaning that $E'(\sigma \cdot (t', a) \cdot (\sigma_u +_t (t' + t'')))) \not\models \varphi$.

$(t', a).(\sigma_u +_t(t' + t'')) \not\models \varphi$, or there exists $\sigma'_u \preceq \sigma_u$, $w_{d3} \neq \epsilon$ such that $\Pi_\Sigma(w_{d3}) \preceq \Pi_\Sigma(\text{buf}_{c2})$ and $\text{Reach}(E'(\sigma.(t', a).(\sigma'_u +_t(t' + t'')))) = q_2$ after σ'_u after w_{d3} . Since $\sigma'_u \preceq \sigma_u$, q_2 after $(\sigma'_u, \text{date}(w_{d3}(1))) \notin S(\text{buf}_{c2}) \cup I(\text{buf}_{c2})$. Considering the definition of I , q_2 after σ'_u after $w_{d3}(1) \notin S(\Pi_\Sigma(w_{d3}(1))^{-1} \cdot \text{buf}_{c2}) \cup I(\Pi_\Sigma(w_{d3}(1))^{-1} \cdot \text{buf}_{c2})$, because otherwise q_2 after $\sigma'_u \in \text{Pred}_{w_{d3}(1)}(S(\Pi_\Sigma(w_{d3}(1))^{-1} \cdot \text{buf}_{c2}) \cup I(\Pi_\Sigma(w_{d3}(1))^{-1} \cdot \text{buf}_{c2})) = I(\text{buf}_{c2})$, which does not hold. It follows that, by iterating the previous reasoning on the first events of w_{d3} that share the same date,

$$\begin{aligned} q_2 \text{ after } \sigma'_u \text{ after } (w_{d3}, \text{date}(w_{d3}(1))) &\notin \\ S(\Pi_\Sigma(\text{obs}(w_{d3}, \text{date}(w_{d3}(1))))^{-1} \cdot \text{buf}_{c2}) &\cup \\ I(\Pi_\Sigma(\text{obs}(w_{d3}, \text{date}(w_{d3}(1))))^{-1} \cdot \text{buf}_{c2}). \end{aligned}$$

Thus, using again lemma A.10, we can find a word in $\text{tw}(\Sigma_u)$ such that the output of E' will never be in S nor I , and end up outside of F_G . Whatever controllable events E' will output, its output will never reach S nor I , and since E' can only output a limited number of controllable events (no more than $|\text{buf}_c|$), at some point it will not be able to output controllable events anymore, and then there will be an uncontrollable word leading its output outside of F_G . Concatenating all the uncontrollable words obtained from lemma A.10, there would be $\sigma_{\text{ug}} \in \text{tw}(\Sigma_u)$ such that $E'(\sigma.(t', a). \sigma_{\text{ug}}) \not\models \varphi$.

Thus, if $a \in \Sigma_u$, there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that $E'(\sigma.(t', a). \sigma_u) \not\models \varphi$.

- If $a \in \Sigma_c$, then since $(\sigma, t') \in \text{Pre}(\varphi)$, following the proof of soundness (appendix A.1.2), $\sigma_d -_t t' \in G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_b, t')). \sigma_c.a)$. Then, we can do the same proof as in the case where $a \in \Sigma_u$, but considering that $q = \text{Reach}(\sigma_s)$ and $\text{buf}_c = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a$.

Thus, if $a \in \Sigma_c$, there also exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that $E'(\sigma.(t', a). \sigma_u) \not\models \varphi$.

This means that whenever $E'(\sigma) = E_\varphi(\sigma)$, and $E_\varphi(\sigma.(t', a)) \prec_d E'(\sigma.(t', a))$, then there exists $\sigma_u \in \Sigma_u$ such that $E'(\sigma.(t', a). \sigma_u) \not\models \varphi$.

Thus, E_φ is optimal. \square

Proposition 3.10. *The output of \mathcal{E} as per Definition 3.21 for input σ is $E_\varphi(\sigma)$ as per Definition 3.19.*

Proof. In this proof, we use some notation from Definition 3.21:

- $C^\mathcal{E} = \text{tw}(\Sigma) \times \Sigma_c^* \times Q \times \mathbb{R}_{\geq 0} \times \{\top, \perp\}$ is the set of configurations,
- $c_0^\mathcal{E} = \langle \epsilon, \epsilon, q_0, 0, \perp \rangle \in C^\mathcal{E}$ 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, composed of an optional input, an operation and an optional output,
- The set of operations, to be applied in the given order, is:
 $\{\text{compute}, \text{dump}, \text{pass-uncont}, \text{store-cont}, \text{delay}\}.$

Let us also introduce some specific notation. For a sequence of rules $w \in \Gamma^{\mathcal{E}*}$, we note $\text{input}(w) = \Pi_1(w(1)) \cdot \Pi_1(w(2)) \dots \Pi_1(w(|w|))$ the concatenation of all inputs from w . In the same way, we define $\text{output}(w) = \Pi_3(w(1)) \cdot \Pi_3(w(2)) \dots \Pi_3(w(|w|))$ the concatenation of all outputs from w . Since all configurations are not reachable from $c_0^\mathcal{E}$, for a word $w \in \Gamma^{\mathcal{E}*}$, we will say that $\text{Reach}^\mathcal{E}(w) = c$ if $c_0^\mathcal{E} \xrightarrow{w}_\mathcal{E} c$, or $\text{Reach}^\mathcal{E}(w) = \perp$ if such a c does not exist. Let us also define function **Rules** which, given a timed word and a date, returns the longest sequence of rules that can be applied with the given word as input at the given date:

$$\text{Rules} : \begin{cases} \text{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \rightarrow \Gamma^\mathcal{E} \\ (\sigma, t) \mapsto \underset{\approx}{\max}(\{w \in \Gamma^\mathcal{E} \mid \text{input}(w) = \sigma \wedge \text{Reach}(w) \neq \perp \wedge \Pi_4(\text{Reach}(w)) = t\}) \end{cases}$$

Since time is not discrete, the rule delay can be applied an infinite number of times by slicing time. Thus, we consider that the rule delay is always applied a minimum number of times, *i.e.* when two rules delay are consecutive, they are merged into one rule delay, whose parameter is the sum of the parameters of the two rules. The runs obtained are equivalent, but it allows to consider the maximum (for prefix order) of the set used in the definition of **Rules**. We then extend **output** to timed words with a date: for $\sigma \in \text{tw}(\Sigma)$, and a date t , $\text{output}(\sigma, t) = \text{output}(\text{Rules}(\sigma, t))$. In the same way, we extend $\text{Reach}^\mathcal{E}$ to timed words with a date, such that $\text{Reach}^\mathcal{E}(\sigma, t) = \text{Reach}^\mathcal{E}(\text{Rules}(\sigma, t))$.

We have to prove that for any $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, $\text{output}(\sigma, t) = E_\varphi(\sigma, t)$.

For $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, let $P(\sigma, t)$ be the predicate: “ $E_\varphi(\sigma, t) = \text{output}(\sigma, t) \wedge (((\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\text{obs}(\sigma, t), t) \wedge \langle \sigma_b^\mathcal{E}, \sigma_c^\mathcal{E}, q^\mathcal{E}, t, b \rangle = \text{Reach}^\mathcal{E}(\sigma, t)) \implies \sigma_b^\mathcal{E} = \text{nobs}(\sigma_b, t) \wedge \sigma_c^\mathcal{E} = \sigma_c \wedge q^\mathcal{E} = \text{Reach}(\sigma_s, t) \wedge (b = \top \implies G(q^\mathcal{E}, \sigma_c^\mathcal{E}) \neq \emptyset))$ ”. Let $P(\sigma)$ be the predicate “ $\forall t \in \mathbb{R}_{\geq 0}, P(\sigma, t)$ holds”. Let us then prove that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis : For $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$. Then, $\text{store}_\varphi(\epsilon, t) = (\epsilon, \epsilon, \epsilon)$, and $\text{Reach}(\epsilon, t) = \langle l_0, v_0 + t \rangle$. On the other hand, the only rules that can be applied are delay, and possibly compute, since there is not any input, nor any element to dump. Thus, $\text{Rules}(\epsilon, t) = \epsilon / \text{delay}(t) / \epsilon$, or there exists $t' \geq t$ such that $\text{Rules}(\epsilon, t) = \epsilon / \text{delay}(t') / \epsilon \cdot \epsilon / \text{compute}() / \epsilon \cdot \epsilon / \text{delay}(t - t') / \epsilon$. Let us consider $c = \text{Reach}(\text{Rules}(\epsilon, t))$. Then, $c = \langle \epsilon, \epsilon, \langle l_0, v_0 + t \rangle, t, b \rangle$. If

rule compute appears in $\text{Rules}(\epsilon, t)$, then $b = \top$, meaning that $G(q_0 \text{ after } (\epsilon, t'), \epsilon) \neq \emptyset$, and thus that $G(q_0 \text{ after } (\epsilon, t), \epsilon) \neq \emptyset$ since $t \geq t'$. Otherwise $b = \perp$. All the other values remain unchanged between the two cases. In both cases, $\text{output}(\text{Rules}(\epsilon, t)) = \epsilon = E_\varphi(\epsilon, t)$. Thus, $P(\epsilon)$ holds.

Induction step : Let us suppose now that for some $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider $(t', a) \in \mathbb{R}_{\geq 0} \times \Sigma$ such that $\sigma \cdot (t', a) \in \text{tw}(\Sigma)$. Let us then prove that $P(\sigma \cdot (t', a))$ holds.

Let us consider $t \in \mathbb{R}_{\geq 0}$, $c = \langle \sigma_b^\mathcal{E}, \sigma_c^\mathcal{E}, q^\mathcal{E}, t', b \rangle = \text{Reach}(\text{Rules}(\sigma, t'))$, $(\sigma_s, \sigma_b, \sigma_c) = \text{store}_\varphi(\sigma, t')$, and $(\sigma_t, \sigma_d, \sigma_e) = \text{store}_\varphi(\text{obs}(\sigma \cdot (t', a), t), t)$.

If $t < t'$, then $\text{obs}(\sigma \cdot (t', a), t) = \text{obs}(\sigma, t)$, and $P(\sigma \cdot (t', a), t)$ trivially holds, since $P(\sigma)$ holds.

Thus, in the following, we consider that $t \geq t'$, so that $\text{store}_\varphi(\text{obs}(\sigma \cdot (t', a), t), t) = \text{store}_\varphi(\sigma \cdot (t', a), t)$:

- If $a \in \Sigma_u$, rule pass-uncont can be applied. Let us consider $c' = c$ after $((t', a) / \text{pass-uncont}((t', a)) / (t', a))$. Then, $c' = \langle \epsilon, \Pi_\Sigma(\sigma_b^\mathcal{E}) \cdot \sigma_c^\mathcal{E}, q', t', \perp \rangle$, with $q' = q^\mathcal{E}$ after $(0, a)$.

Then, if $t \geq t_1^\mathcal{E}$, where $t_1^\mathcal{E} = \min(\{t'' \mid t'' \geq t' \wedge G(q' \text{ after } (\epsilon, t'' - t'), \Pi_\Sigma(\sigma_b^\mathcal{E}) \cdot \sigma_c^\mathcal{E}) \neq \emptyset\})$, then rule $\text{delay}(t_1^\mathcal{E} - t')$ can be applied, followed by rule compute. Since $q^\mathcal{E} = \text{Reach}(\sigma_s, t')$, $\sigma_b^\mathcal{E} = \text{nobs}(\sigma_b, t')$, and $\sigma_c^\mathcal{E} = \sigma_c$ (by induction hypothesis), then $G(q' \text{ after } (\epsilon, t'' - t'), \Pi_\Sigma(\sigma_b^\mathcal{E}) \cdot \sigma_c^\mathcal{E}) = G(\text{Reach}(\sigma_s \cdot (t', a), t''), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c)$, thus $t_1^\mathcal{E} = t_1$, where t_1 is defined in Definition 3.19. Thus, c' after $((\epsilon / \text{delay}(t_1^\mathcal{E} - t') / \epsilon) \cdot (\epsilon / \text{compute} / \epsilon)) = \langle \sigma_d^\mathcal{E}, \sigma_e^\mathcal{E}, q' \text{ after } (\epsilon, t_1 - t'), t_1, \top \rangle$, with $\sigma_d^\mathcal{E} = \kappa_\varphi(q' \text{ after } (\epsilon, t_1 - t'), \Pi_\Sigma(\sigma_b^\mathcal{E}) \cdot \sigma_c^\mathcal{E}) +_t t_1 = \kappa_\varphi(\text{Reach}(\sigma_s \cdot (t', a), t_1), \Pi_\Sigma(\sigma_b) \cdot \sigma_c) +_t t_1 = \sigma_d$, and thus $\sigma_e^\mathcal{E} = \sigma_e$. Then, rules delay and dump can be applied until date t is reached. In the end, $\text{Reach}^\mathcal{E}(\sigma \cdot (t', a), t) = c'$ after w , where w is composed of an alternation of rules delay and dump, thus $\text{Reach}^\mathcal{E}(\sigma \cdot (t', a), t) = \langle \text{nobs}(\sigma_d^\mathcal{E}, t), \sigma_e^\mathcal{E}, q' \text{ after } (\text{obs}(\sigma_d^\mathcal{E}, t) -_t t', t - t'), t, \top \rangle = \langle \text{nobs}(\sigma_d, t), \sigma_e, \text{Reach}(\sigma_t, t), t, \top \rangle$. Then, $\text{output}(\sigma \cdot (t', a), t) = \text{output}(\sigma, t') \cdot (t', a) \cdot \text{obs}(\sigma_d^\mathcal{E}, t) = \sigma_s \cdot (t', a) \cdot \text{obs}(\sigma_d, t) = \sigma_t$.

Thus, if $t \geq t_1$, $P(\sigma \cdot (t', a), t)$ holds.

Otherwise, $t < t_1$, and then rule dump cannot be applied, since $\Pi_5(c') = \perp$, and rule compute also cannot be applied. Thus, the only rule that can be applied is delay, so that $\text{Reach}(\text{Rules}(\sigma \cdot (t', a), t)) = \langle \epsilon, \Pi_\Sigma(\sigma_b^\mathcal{E}) \cdot \sigma_c^\mathcal{E}, q' \text{ after } (\epsilon, t - t'), t', \perp \rangle$. Since $t < t_1$, this means that $\sigma_d = \epsilon$, and $\sigma_e = \Pi_\Sigma(\sigma_b) \cdot \sigma_c$. Thus, $\text{output}(\text{Rules}(\sigma \cdot (t', a), t)) = \text{output}(\text{Rules}(\sigma, t')) \cdot (t', a) = \sigma_s \cdot (t', a) = \sigma_t$, and $\sigma_d^\mathcal{E} = \sigma_d$, and $\sigma_e^\mathcal{E} = \sigma_e$. This means that $P(\sigma \cdot (t', a), t)$ holds when $t < t_1$.

Thus, if $a \in \Sigma_u$, then $P(\sigma \cdot (t', a), t)$ holds for all $t \geq t'$.

- Otherwise, $a \in \Sigma_c$. Then, rule store-cont can be applied. Let us consider $c' = c$ after $((t', a) / \text{store-cont}(a) / \epsilon)$. Then, $c' = \langle \epsilon, \Pi_\Sigma(\sigma_b^\epsilon) \cdot \sigma_c^\epsilon \cdot a, q^\epsilon, t', \perp \rangle$. Let us consider $t_2^\epsilon = \min(\{t'' \in \mathbb{R}_{\geq 0} \mid t'' \geq t' \wedge G(q^\epsilon \text{ after } (\epsilon, t'' - t'), \Pi_\Sigma(\sigma_b^\epsilon) \cdot \sigma_c^\epsilon \cdot a) \neq \emptyset\})$. Since $G(q^\epsilon \text{ after } (\epsilon, t'' - t'), \Pi_\Sigma(\sigma_b^\epsilon) \cdot \sigma_c^\epsilon \cdot a) = G(\text{Reach}(\sigma_s, t''), \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a)$, it follows that $t_2^\epsilon = t_2$ as defined in Definition 3.19.

If $t \geq t_2^\epsilon = t_2$, then rule delay($t_2 - t'$) can be applied, followed by rule compute. Then, c' after $((\epsilon / \text{delay}(t_2 - t') / \epsilon) \cdot (\epsilon / \text{compute}() / \epsilon)) = \langle \sigma_d^\epsilon, \sigma_e^\epsilon, q \text{ after } (\epsilon, t_2 - t'), t_2, \top \rangle$, where $\sigma_d^\epsilon = \kappa_\varphi(q \text{ after } (\epsilon, t_2 - t'), \Pi_\Sigma(\sigma_b^\epsilon) \cdot \sigma_c^\epsilon \cdot a) +_t t_2 = \kappa_\varphi(\text{Reach}(\sigma_s, t_2), \Pi_\Sigma(\sigma_b) \cdot \sigma_c \cdot a) +_t t_2 = \sigma_d$. Then, $\sigma_e^\epsilon = \sigma_e$. Then, an alternation of rules delay and dump can be applied until date t is reached. This leads to $\text{Reach}(\text{Rules}(\sigma \cdot (t', a), t)) = \langle \text{nobs}(\sigma_d^\epsilon, t), \sigma_e^\epsilon, q \text{ after } (\text{obs}(\sigma_d^\epsilon, t), t), t, \top \rangle = \langle \text{nobs}(\sigma_d, t), \sigma_e, \text{Reach}(\sigma_t, t), t, \top \rangle$. Moreover, $\text{output}(\text{Rules}(\sigma \cdot (t', a), t)) = \text{output}(\sigma, t') \cdot \text{obs}(\sigma_d, t) = \sigma_s \cdot \text{obs}(\sigma_d, t) = E_\varphi(\sigma \cdot (t', a), t)$.

Thus, if $t \geq t_2$, $P(\sigma \cdot (t', a), t)$ holds.

Otherwise, $t < t_2$, meaning that $\sigma_d^\epsilon = \epsilon = \sigma_d$, and $\sigma_e^\epsilon = \Pi_\Sigma(\sigma_b^\epsilon) \cdot \sigma_c^\epsilon \cdot a = \Pi_\Sigma(\text{nobs}(\sigma_b, t')) \cdot \sigma_c \cdot a = \sigma_e$, and $\text{output}(\sigma \cdot (t', a), t) = \text{output}(\sigma, t') = \sigma_s = E_\varphi(\sigma \cdot (t', a), t)$. Thus, $P(\sigma \cdot (t', a), t)$ holds.

Thus, $P(\sigma) \implies P(\sigma \cdot (t, a))$.

Thus, by induction on σ , for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. In particular, for all $\sigma \in \text{tw}(\Sigma)$, and for all $t \in \mathbb{R}_{\geq 0}$, $\text{output}(\sigma, t) = E_\varphi(\sigma, t)$, meaning that the output of the enforcement monitor \mathcal{E} with input σ at time t is exactly the output of function E_φ with the same input and the same date. \square

A.2 Proofs of Chapter 4

A.2.1 Proofs for the untimed setting (Section 4.2)

In all this section, we will use the notations from Section 4.2, meaning that φ is a property whose associated automaton is $\mathcal{A}_\varphi = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$, and the game graph $\mathcal{G} = \langle V, E \rangle$ is the one as per Definition 4.5. In some proofs, we also use notations from Definition 4.9.

Proposition 4.1. E_φ as per Definition 4.9 is an enforcement function as per Definition 4.1.

Proof. We have to prove that for $\sigma \in \Sigma^*$ and $\sigma' \in \Sigma^*$, if $(\sigma, o) \in E_\varphi$, $(\sigma', o') \in E_\varphi$ and $\sigma \preceq \sigma'$, then $o \preceq o'$. Let us consider $\sigma \in \Sigma^*$, $\sigma' \in \Sigma^*$, $(\sigma, o) \in E_\varphi$ and $(\sigma \cdot \sigma', o') \in E_\varphi$.

If $\sigma' = \epsilon$, then $o = o \preceq o'$.

Otherwise, let us consider $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, $a = \sigma'(1)$, $(\sigma \cdot a, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_\varphi$, and $(\sigma \cdot a, o_a) \in E_\varphi$. Then,

- if $a \in \Sigma_u$, $\sigma_t = \sigma_s \cdot a \cdot \sigma'_s$, where σ'_s is defined in Definition 4.9, meaning that $\sigma_s \preceq \sigma_t$.
- If $a \in \Sigma_c$, then $\sigma_t = \sigma_s \cdot \sigma''_s$, where σ''_s is defined in Definition 4.9, thus again, $\sigma_s \preceq \sigma_t$.

In both cases, $o = \sigma_s \preceq \sigma_t = o_a$. Since the order \preceq is transitive, we can iterate through all the events of σ' , thus $o \preceq o_a \preceq \dots \preceq \sigma'$.

Thus E_φ is an enforcement function. \square

Lemma A.11. $\forall q \in Q, \forall w \in \Sigma_c^n, (\langle q, w, 1 \rangle \in W_0 \wedge (\langle q, w, 1 \rangle, \langle q', w', l \rangle) \in E) \implies \langle q', w', l \rangle \in W_0$.

Proof. W_0 is the winning set of the Büchi game for P_0 . \square

Lemma A.12. $\forall q \in Q, \forall \sigma \in \Sigma_c^n, \langle q, \sigma, 0 \rangle \in W_0 \implies G(q, \sigma) \neq \emptyset$.

Proof. Let us consider $q \in Q$ and $\sigma \in \Sigma_c^n$ such that $\langle q, \sigma, 0 \rangle \in W_0$. Then, since $\langle q, \sigma, 0 \rangle$ is a node that belongs to P_0 that is winning (since it is in W_0), this means that there is a winning strategy for P_0 in the Büchi game. Thus, there is a path in \mathcal{G} that allows P_0 to reach a Büchi node, that is a node in $F \times \Sigma_c^n \times \{0, 1\}$, whatever the strategy of P_1 is. The strategy of P_0 is to follow nodes that are only in W_0 until it finally reaches a Büchi node. The construction of W_0 ensures that this is possible. Now, the only edges that leave a node belonging to P_0 are the ones corresponding to the action of emitting the first of the stored controllable events, or not emitting it and let P_1 play. Thus, if $\langle q, \sigma, 0 \rangle \in W_0$, this means that there is a path in the graph that leads to a node in $F \times \Sigma_c^n \times \{0\}$ such that all the nodes along the path belong to P_0 and are in W_0 . This holds because there is a path in W_0 to such a node, and if a node of the path belongs to P_1 , then the strategy of P_1 could be to go back to the previous node belonging to P_0 , and thus there could be an infinite loop in these two nodes, meaning that they are in $F \times \Sigma_c^n \times \{0, 1\}$ or that from the previous node belonging to P_0 , emitting the first stored controllable event is a winning strategy. Thus, there exists $w \preceq \sigma$ such that q after $w \in F$ and $\langle q \text{ after } w, w^{-1} \cdot \sigma, 0 \rangle \in W_0$. Now, since Σ_c^n is finite, it is possible to choose w such that $\langle q \text{ after } w, w^{-1} \cdot \sigma, 1 \rangle \in W_0$, because otherwise, the only possible strategy would be to emit from every node, but it is not possible from nodes whose second member is ϵ , and then the only possible strategy would lead to a node not in W_0 , meaning that the original node would not be in W_0 , which is absurd. Thus, $G(q, \sigma) \neq \emptyset$. \square

Lemma A.13. $\forall \sigma \in \Sigma^*, (\sigma \notin \text{Pre}(\varphi) \wedge (\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_c = \sigma|_{\Sigma_c})$.

Proof. For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate “ $(\sigma \notin \text{Pre}(\varphi) \wedge (\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\sigma_s = \sigma|_{\Sigma_u} \wedge \sigma_c = \sigma|_{\Sigma_c})$ ”. Let us show by induction that $P(\sigma)$ holds for any $\sigma \in \Sigma^*$.

Induction basis: $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$, and since $\epsilon|_{\Sigma_u} = \epsilon|_{\Sigma_c} = \epsilon$, $P(\epsilon)$ holds.

Induction step: let us suppose that for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us then consider $a \in \Sigma$, $(\sigma, \langle \sigma_s, \sigma_b \rangle) \in \text{store}_\varphi$, and $(\sigma \cdot a, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_\varphi(\sigma \cdot a)$.

Then, if $\sigma \cdot a \in \text{Pre}(\varphi)$, $P(\sigma \cdot a)$ holds.

Let us now consider that $\sigma \cdot a \notin \text{Pre}(\varphi)$. Then, since $\text{Pre}(\varphi)$ is extension-closed, $\sigma \notin \text{Pre}(\varphi)$, and thus, by induction hypothesis, $\sigma_s = \sigma|_{\Sigma_u}$ and $\sigma_c = \sigma|_{\Sigma_c}$. We consider two cases:

- if $a \in \Sigma_u$, then $\sigma_t = \sigma_s \cdot a \cdot \sigma'_s$, with $\sigma'_s \in G(\text{Reach}(\sigma_s \cdot a), \sigma_c) \cup \{\epsilon\}$ (according to Definition 4.9). Since $\sigma \cdot a \notin \text{Pre}(\varphi)$, following Definition 4.10, $G(\text{Reach}((\sigma \cdot a)|_{\Sigma_u}), (\sigma \cdot a)|_{\Sigma_c}) = \emptyset$. Moreover, since $a \in \Sigma_u$, $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} \cdot a = \sigma_s \cdot a$ and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} = \sigma_c$, thus $G(\text{Reach}(\sigma_s \cdot a), \sigma_c) = \emptyset$. It follows that $\sigma'_s \in \{\epsilon\}$, meaning that $\sigma_t = \sigma_s \cdot a = \sigma|_{\Sigma_u} \cdot a = (\sigma \cdot a)|_{\Sigma_u}$, and $\sigma_d = \sigma'^{-1}_s \cdot \sigma_c = \sigma_c = \sigma|_{\Sigma_c} = (\sigma \cdot a)|_{\Sigma_c}$.
- Otherwise, $a \in \Sigma_c$, and then, according to Definition 4.9, $\sigma_t = \sigma_s \cdot \sigma''_s$, with $\sigma''_s \in G(\sigma_s, \sigma_c \cdot a) \cup \{\epsilon\}$. Since $\sigma \cdot a \notin \text{Pre}(\varphi)$, following Definition 4.9, $G(\text{Reach}((\sigma \cdot a)|_{\Sigma_u}), (\sigma \cdot a)|_{\Sigma_c}) = \emptyset$. Moreover, since $a \in \Sigma_c$, $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$ and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} \cdot a = \sigma_c \cdot a$. Thus, $G(\text{Reach}(\sigma_s), \sigma_c \cdot a) = \emptyset$, meaning that $\sigma''_s = \epsilon$. Thus, $\sigma_t = \sigma_s = \sigma|_{\Sigma_u} = (\sigma \cdot a)|_{\Sigma_u}$ and $\sigma_d = \sigma''^{-1}_s \cdot (\sigma_c \cdot a) = \sigma_c \cdot a = \sigma|_{\Sigma_c} \cdot a = (\sigma \cdot a)|_{\Sigma_c}$.

In both cases, $P(\sigma \cdot a)$ holds.

Thus, $P(\sigma) \implies P(\sigma \cdot a)$.

By induction on $\sigma \in \Sigma^*$, for all $\sigma \in \Sigma^*$, if $\sigma \notin \text{Pre}(\varphi)$ and $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, then $\sigma_s = \sigma|_{\Sigma_u}$ and $\sigma_c = \sigma|_{\Sigma_c}$. \square

Proposition 4.2. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$, as per Definition 4.2.

Proof. We have to prove that for $\sigma \in \text{Pre}(\varphi)$, if $(\sigma, o) \in E_\varphi$, then $o \models \varphi$.

Let $P(\sigma)$ be the predicate: “ $(\sigma \in \text{Pre}(\varphi) \wedge (\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\sigma_s \models \varphi \wedge \langle \text{Reach}(\sigma_s), \max_{\preceq}(\{w \preceq \sigma_c \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0)$ ”. Let us prove by induction that for any $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis: if $\epsilon \in \text{Pre}(\varphi)$, then following the definition of $\text{Pre}(\varphi)$ (Definition 4.10), $G(\text{Reach}(\epsilon), \epsilon) \neq \emptyset$. Thus $\epsilon \in G(\text{Reach}(\epsilon), \epsilon)$ (since ϵ is the only word satisfying $\epsilon \preceq \epsilon$). This means that $\text{Reach}(\epsilon)$ after $\epsilon = \text{Reach}(\epsilon) \in F$, and thus that $\epsilon \models \varphi$.

Moreover, since $\epsilon \in G(\text{Reach}(\epsilon), \epsilon)$, $\langle \text{Reach}(\epsilon) \text{ after } \epsilon, \max(\{w \preceq \epsilon^{-1} \cdot \epsilon \mid w \in \Sigma_c^n\}), 1 \rangle = \langle \text{Reach}(\epsilon), \epsilon, 1 \rangle \in W_0$.

Considering that $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$, this means that $P(\epsilon)$ holds.

Induction step: Suppose now that, for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $a \in \Sigma$, $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, and $(\sigma \cdot a, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_\varphi$.

Let us prove that $P(\sigma \cdot a)$ holds.

We consider three different cases:

- $(\sigma \cdot a) \notin \text{Pre}(\varphi)$. Then $P(\sigma \cdot a)$ holds.
- $(\sigma \cdot a) \in \text{Pre}(\varphi) \wedge \sigma \notin \text{Pre}(\varphi)$. Then, since $\text{Pre}(\varphi)$ is extension-closed, it follows that $\sigma \cdot a \in \{w \in \Sigma^* \mid G(\text{Reach}(w|_{\Sigma_u}), w|_{\Sigma_c}) \neq \emptyset\}$, meaning that $G(\text{Reach}((\sigma \cdot a)|_{\Sigma_u}), (\sigma \cdot a)|_{\Sigma_c}) \neq \emptyset$. Moreover, since $\sigma \notin \text{Pre}(\varphi)$, following lemma A.13, $\sigma_s = \sigma|_{\Sigma_u}$ and $\sigma_c = \sigma|_{\Sigma_c}$. Now, we consider two cases:

- If $a \in \Sigma_u$, then $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} \cdot a = \sigma_s \cdot a$, and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} = \sigma_c$, thus $G(\text{Reach}((\sigma \cdot a)|_{\Sigma_u}), (\sigma \cdot a)|_{\Sigma_c}) = G(\text{Reach}(\sigma_s \cdot a), \sigma_c) \neq \emptyset$, meaning that $\sigma'_s = (\sigma_s \cdot a)^{-1} \cdot \sigma_t \in G(\text{Reach}(\sigma_s \cdot a), \sigma_c)$ (according to Definition 4.9). Thus, following the definition of G (Definition 4.8), $\text{Reach}(\sigma_s \cdot a)$ after $\sigma'_s = \text{Reach}(\sigma_s \cdot a \cdot \sigma'_s) = \text{Reach}(\sigma_t) \in F$, and $\langle \text{Reach}(\sigma_s \cdot a) \text{ after } \sigma'_s, \max_{\preceq}(\{w \preceq \sigma'^{-1}_s \cdot (\sigma_c) \mid w \in \Sigma_c^n\}), 1 \rangle = \langle \text{Reach}(\sigma_t), \max_{\preceq}(\{w \preceq \sigma_d \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$. Since $\text{Reach}(\sigma_t) \in F$, $\sigma_t \models \varphi$.

This means that $P(\sigma \cdot a)$ holds.

- If $a \in \Sigma_c$, then $(\sigma \cdot a)|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$, and $(\sigma \cdot a)|_{\Sigma_c} = \sigma|_{\Sigma_c} \cdot a = \sigma_c \cdot a$. Thus, $G(\text{Reach}(\sigma_s), \sigma_c \cdot a) \neq \emptyset$, meaning that $\sigma''_s = \sigma_s^{-1} \cdot \sigma_t \in G(\text{Reach}(\sigma_s), \sigma_c \cdot a)$. As in the case where $a \in \Sigma_u$, it follows that $\langle \text{Reach}(\sigma_t), \max_{\preceq}(\{w \preceq \sigma_d \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$ and thus $\sigma_t \models \varphi$.

This means that $P(\sigma \cdot a)$ holds.

Thus, if $\sigma \cdot a \in \text{Pre}(\varphi)$ but $\sigma \notin \text{Pre}(\varphi)$, $P(\sigma \cdot a)$ holds.

- $\sigma \in \text{Pre}(\varphi)$ (and then $(\sigma \cdot a) \in \text{Pre}(\varphi)$ since $\text{Pre}(\varphi)$ is extension-closed). Then, by induction hypothesis, $P(\sigma)$ holds, meaning that $\sigma_s \models \varphi$ and $\langle \text{Reach}(\sigma_s), \max_{\preceq}(\{w \preceq \sigma_c \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$. Let us note $\sigma_c^m = \max_{\preceq}(\{w \preceq \sigma_c \mid w \in \Sigma_c^n\})$. Again, we consider two cases:
- If $a \in \Sigma_u$, then, since $\langle \text{Reach}(\sigma_s), \sigma_c^m, 1 \rangle \in W_0$, following lemma A.11, since $(\langle \text{Reach}(\sigma_s), \sigma_c^m, 1 \rangle, \langle \text{Reach}(\sigma_s) \text{ after } a, \sigma_c^m, 0 \rangle) \in E_3 \subseteq E$, $\langle \text{Reach}(\sigma_s) \text{ after } a, \sigma_c^m, 0 \rangle = \langle \text{Reach}(\sigma_s \cdot a), \sigma_c^m, 0 \rangle \in W_0$. Following

lemma A.12, this means that $G(\text{Reach}(\sigma_s . a), \sigma_c) \neq \emptyset$, thus $\sigma'_s = (\sigma_s . a)^{-1} . \sigma_t \in G(\text{Reach}(\sigma_s . a), \sigma_c)$. It follows that $\text{Reach}(\sigma_s . a . \sigma'_s) = \text{Reach}(\sigma_t) \in F$, and that $\langle \text{Reach}(\sigma_s . a), \max_{\preceq}(\{w \preceq \sigma'^{-1}_s . \sigma_c \mid w \in \Sigma_c^n\}), 1 \rangle = \langle \text{Reach}(\sigma_t), \max_{\preceq}(\{w \preceq \sigma_d \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$. Thus, $\sigma_t \models \varphi$ and $\langle \text{Reach}(\sigma_t), \max_{\preceq}(\{w \preceq \sigma_d \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$.

Thus $P(\sigma . a)$ holds.

- If $a \in \Sigma_c$, then, since $\langle \text{Reach}(\sigma_s), \sigma_c^m, 1 \rangle \in W_0$ and $(\langle \text{Reach}(\sigma_s), \sigma_c^m, 1 \rangle, \langle \text{Reach}(\sigma_s), \max_{\preceq}(\{w \preceq \sigma_c . a \mid w \in \Sigma_c^n\}), 0 \rangle) \in E_4 \cup E_5 \subseteq E$, following lemmas A.11 and A.12, this means that $G(\text{Reach}(\sigma_s), \sigma_c . a) \neq \emptyset$. Thus, $\sigma''_s = \sigma_s^{-1} . \sigma_t \in G(\text{Reach}(\sigma_s), \sigma_c . a)$. As in the previous case, this means that $\sigma_t \models \varphi$ and $\langle \text{Reach}(\sigma_t), \max_{\preceq}(\{w \preceq \sigma_d \mid w \in \Sigma_c^n\}), 1 \rangle \in W_0$.

Thus $P(\sigma . a)$ holds.

Thus, if $\sigma \in \text{Pre}(\varphi)$, $P(\sigma . a)$ holds.

In all cases, $P(\sigma . a)$ holds, meaning that $P(\sigma) \implies P(\sigma . a)$.

Thus, by induction on σ , for any $\sigma \in \text{Pre}(\varphi)$, if $(\sigma, \langle \sigma_s, \sigma_b \rangle) \in \text{store}_\varphi$, then $\sigma_s \models \varphi$ and $\langle \text{Reach}(\sigma_s), \sigma_c, 1 \rangle \in W_0$. In particular, for all $\sigma \in \text{Pre}(\varphi)$, $(\sigma, o) \in E_\varphi \implies o = \sigma_s \models \varphi$.

This means that E_φ is sound with respect to φ in $\text{Pre}(\varphi)$. \square

Proposition 4.3. E_φ is compliant, as per Definition 4.3.

Proof. We have to show that for any $\sigma \in \Sigma^*$, if $(\sigma, o) \in E_\varphi$, then the following properties hold:

1. $o \preceq_{\Sigma_c} \sigma$
2. $o =_{\Sigma_u} \sigma$
3. $\forall u \in \Sigma_u, (\sigma . u, o') \in E_\varphi \implies o . u \preceq o'$.

We start by proving that items 1 and 2 hold. For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate: “ $((\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\sigma_{s|\Sigma_c} . \sigma_c = \sigma_{|\Sigma_c} \wedge \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u})$ ”. Let us prove that for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis : $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$, and $\epsilon_{|\Sigma_c} = \epsilon_{|\Sigma_c} . \epsilon$, and $\epsilon_{|\Sigma_u} = \epsilon_{|\Sigma_u}$. Thus $P(\epsilon)$ holds.

Induction step : Let us suppose that for $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, $a \in \Sigma$, and $(\sigma . a, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_\varphi$. Let us prove that $P(\sigma . a)$ holds.

- If $a \in \Sigma_u$, then, $\sigma_t = \sigma_s . a . \sigma'_s$, where σ'_s is defined in Definition 4.9, and $\sigma_t . \sigma_d = \sigma_s . a . \sigma_c$. Therefore, $\sigma_{t|\Sigma_c} . \sigma_d = (\sigma_t . \sigma_d)_{|\Sigma_c}$, since $\sigma_d \in \Sigma_c^*$. Thus, $\sigma_{t|\Sigma_c} . \sigma_d = \sigma_{s|\Sigma_c} . \sigma_c$. Since $P(\sigma)$ holds, $\sigma_{t|\Sigma_c} . \sigma_d = \sigma_{|\Sigma_c} = (\sigma . a)_{|\Sigma_c}$. Moreover, since $\sigma'_s \in \Sigma_c^*$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u} . a$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_u} = \sigma_{|\Sigma_u} . a = (\sigma . a)_{|\Sigma_u}$. Thus $P(\sigma . a)$ holds.
- Otherwise, $a \in \Sigma_c$, and then $\sigma_t = \sigma_s . \sigma''_s$, where σ''_s is defined in Definition 4.9, and $\sigma_t . \sigma_d = \sigma_s . \sigma_c . a$. Therefore, $\sigma_{t|\Sigma_c} . \sigma_d = (\sigma_t . \sigma_d)_{|\Sigma_c} = (\sigma_s . \sigma_c . a)_{|\Sigma_c} = \sigma_{s|\Sigma_c} . \sigma_c . a$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_c} . \sigma_d = \sigma_{|\Sigma_c} . a = (\sigma . a)_{|\Sigma_c}$. Moreover, since $\sigma''_s \in \Sigma_c^*$, $\sigma_{t|\Sigma_u} = \sigma_{s|\Sigma_u}$. Since $P(\sigma)$ holds, this means that $\sigma_{t|\Sigma_u} = \sigma_{|\Sigma_u} = (\sigma . a)_{|\Sigma_u}$. Thus $P(\sigma . a)$ holds.

In both cases, $P(\sigma . a)$ holds.

Thus, for any $\sigma \in \Sigma^*$, and $a \in \Sigma$, $P(\sigma) \implies P(\sigma . a)$.

Thus, by induction on σ , for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds, meaning that $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi \implies (\sigma_{s|\Sigma_c} . \sigma_c = \sigma_{|\Sigma_c} \wedge \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u})$. If $(\sigma, o) \in E_\varphi$, then $o = \sigma_s$, meaning that $o_{|\Sigma_c} = \sigma_{s|\Sigma_c} \preceq \sigma_{s|\Sigma_c} . \sigma_c = \sigma_{|\Sigma_c}$, and $o_{|\Sigma_u} = \sigma_{s|\Sigma_u} = \sigma_{|\Sigma_u}$. Thus, items 1 and 2 hold.

Now, let us prove that item 3 holds. Let us consider $\sigma \in \Sigma^*$, $u \in \Sigma_u$, $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$, and $(\sigma . u, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_\varphi$, then $\sigma_t = \sigma_s . u . \sigma'_s$, where σ'_s is defined in Definition 4.9. Thus $\sigma_s . u \preceq \sigma_t$, meaning that if $(\sigma, o) \in E_\varphi$ and $(\sigma . u, o') \in E_\varphi$, then $o = \sigma_s$, and $o' = \sigma_t$, and thus $o . u \preceq o'$. Thus item 3 holds.

Thus, E_φ is compliant with respect to Σ_u and Σ_c . □

Proposition 4.4. E_φ is optimal in $\text{Pre}(\varphi)$, as per Definition 4.4.

Proof. Let E be an enforcement function such that $\text{compliant}(E, \Sigma_u, \Sigma_c)$ holds, and let us consider $\sigma \in \text{Pre}(\varphi)$, $a \in \Sigma$ such that $(\sigma, o) \in E \cap E_\varphi$, $(\sigma . a, o') \in E_\varphi$ and $(\sigma . a, p') \in E$.

Then, we have to prove that $p' \preceq o'$.

Let us consider $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$. Let us suppose that $o' \prec p'$. We then show that there exists $\sigma_u \in \Sigma_u^*$ such that if $(\sigma . a . \sigma_u, p_u) \in E$, then $p_u \not\models \varphi$, meaning that E is not sound, and thus that if E is sound then $o' \prec p'$ does not hold. We distinguish two cases:

- if $a \in \Sigma_u$, then, since E is compliant, and $(\sigma, o) \in E \cap E_\varphi$, there exists $\sigma_{s1} \preceq \sigma_c$ such that $p' = o . a . \sigma_{s1} = \sigma_s . a . \sigma_{s1}$. Moreover, there exists $\sigma'_s \preceq \sigma_c$ such that $o' = o . a . \sigma'_s = \sigma_s . a . \sigma'_s$. Since $o' \prec p'$, $\sigma'_s \prec \sigma_{s1}$. Considering that $\sigma'_s = \max_{\preceq}(\text{G}(\text{Reach}(\sigma_s . a), \sigma_c) \cup \{\epsilon\})$, it follows that $\sigma_{s1} \notin \text{G}(\text{Reach}(\sigma_s . a), \sigma_c)$. Following the definition of G (Definition 4.8), this means that either $\sigma_{s1} \not\preceq \sigma_c$; $\text{Reach}(\sigma_s . a)$ after $\sigma_{s1} \notin F$; or that $\langle \text{Reach}(\sigma_s . a) \text{ after } \sigma_{s1}, \sigma_{s1}^{-1} . \sigma_c, 1 \rangle \notin W_0$. Since E' is compliant, $\sigma_{s1} \preceq \sigma_c$, thus at least one of the two last conditions holds.

If $\text{Reach}(\sigma_s . a) \text{ after } \sigma_{s1} = \text{Reach}(\sigma_s . a . \sigma_{s1}) = \text{Reach}(p') \notin F$, then $p' \not\models \varphi$.

Otherwise, $\langle \text{Reach}(\sigma_s . a) \text{ after } \sigma_{s1}, \sigma_{s1}^{-1} . \sigma_c, 1 \rangle \notin W_0$. Then, $\langle \text{Reach}(\sigma_s . a . \sigma_{s1}), \sigma_{s1}^{-1} . \sigma_c, 1 \rangle \in W_1$, meaning that P_1 has a winning strategy. Since receiving controllable events only helps P_0 to win, this means that there exists an uncontrollable event $u \in \Sigma_u$ such that $\langle \text{Reach}(\sigma_s . a . \sigma_{s1}) \text{ after } u, \sigma_{s1}^{-1} . \sigma_c, 0 \rangle \in W_1$. Then, since W_1 is the set of winning nodes for P_1 , if $(\sigma . a . u, p'') \in E$, then $\langle \text{Reach}(p''), p''^{-1} . (\sigma . a . u)_{|\Sigma_c}, 1 \rangle \in W_1$. Then again, there exists an uncontrollable event u' such that the output of E after receiving it reaches a node in W_1 again. In the end, it is possible to reach a node that is not a Büchi node (*i.e.* in $\bar{F} \times \Sigma_c^n \times \{0, 1\}$), and that is in W_1 . Thus, there exists $\sigma_u \in \Sigma_u^*$ such that if $(\sigma . a . \sigma_u, p_u) \in E_\varphi$, then $\text{Reach}(p_u) \notin F$, meaning that $p_u \not\models \varphi$.

- Otherwise, $a \in \Sigma_c$, and then the proof is the same as in the case where $a \in \Sigma_u$, by replacing occurrences of “ $\sigma_s . a$ ” by “ σ_s ”, and occurrences of “ σ_c ” by “ $\sigma_c . a$ ”.

In both cases, there exists σ_u such that if $(\sigma . a . u, p_u) \in E$, then $p_u \not\models \varphi$. Since $\sigma . a . u \in \text{Pre}(\varphi)$, it follows that E is not sound in $\text{Pre}(\varphi)$.

Thus, if E is sound in $\text{Pre}(\varphi)$, it follows that $p' \preceq o'$.

This means that E_φ is optimal in $\text{Pre}(\varphi)$. \square

Proposition 4.5. *The output o of the enforcement monitor \mathcal{E} as per Definition 3.12 for input σ is the output of E_φ as per Definition 4.9 with input σ , *i.e.* $(\sigma, o) \in E_\varphi$.*

Proof. Let us introduce some notation for this proof: for a word $w \in \Gamma^{\mathcal{E}*}$, we note $\text{input}(w) = \Pi_1(w(1)) . \Pi_1(w(2)) \dots \Pi_1(w(|w|))$, the word obtained by concatenating the first members (the inputs) of w . In a similar way, we note $\text{output}(w) = \Pi_3(w(1)) . \Pi_3(w(2)) \dots \Pi_3(w(|w|))$, the word obtained by concatenating all the third members (outputs) of w . Since all configurations are not reachable from $c_0^\mathcal{E}$, for $w \in \Gamma^{\mathcal{E}*}$, we note $\text{Reach}^\mathcal{E}(w) = c$ whenever $c_0^\mathcal{E} \xrightarrow{w}_\mathcal{E} c$, and $\text{Reach}^\mathcal{E}(w) = \perp$ if such a c does not exist. We also define the Rules function as follows:

$$\text{Rules} : \begin{cases} \Sigma^* & \rightarrow \Gamma^{\mathcal{E}*} \\ \sigma & \mapsto \max_{\preceq}(\{w \in \Gamma^{\mathcal{E}*} \mid \text{input}(w) = \sigma \wedge \text{Reach}(w) \neq \perp\}) \end{cases}$$

For a word $\sigma \in \Sigma^*$, $\text{Rules}(\sigma)$ is the trace of the longest valid run in \mathcal{E} , *i.e.* the sequence of all the rules that can be applied with input σ . We then extend the definition of output to words in Σ^* : for $\sigma \in \Sigma^*$, $\text{output}(\sigma) = \text{output}(\text{Rules}(\sigma))$. In the same way, we note $\text{Reach}^{\mathcal{E}}(\sigma) = \text{Reach}^{\mathcal{E}}(\text{Rules}(\sigma))$.

We have to show that, for any $\sigma \in \Sigma^*$, $\text{output}(\sigma) = o$, where $(\sigma, o) \in E_{\varphi}$.

For $\sigma \in \Sigma^*$, let $P(\sigma)$ be the predicate: “ $((\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi} \wedge \text{Reach}^{\mathcal{E}}(\sigma) = \langle q, \sigma_c^{\mathcal{E}} \rangle) \implies (q = \text{Reach}(\sigma_s) \wedge \sigma_c = \sigma_c^{\mathcal{E}} \wedge \sigma_s = \text{output}(\sigma))$ ”.

Let us prove by induction that for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds.

Induction basis: $(\epsilon, \epsilon) \in E_{\varphi}$, and $\text{output}(\epsilon) = \epsilon$. Moreover, $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_{\varphi}$, and $\text{Reach}^{\mathcal{E}}(\epsilon) = c_0^{\mathcal{E}}$. Therefore, as $c_0^{\mathcal{E}} = \langle q_0, \epsilon \rangle$, $P(\epsilon)$ holds, because $\text{Reach}(\epsilon) = q_0$.

Induction step: Let us suppose now that for some $\sigma \in \Sigma^*$, $P(\sigma)$ holds. Let us consider $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi}$, $q = \text{Reach}(\sigma_s)$, $a \in \Sigma$, and $(\sigma \cdot a, \langle \sigma_t, \sigma_d \rangle) \in \text{store}_{\varphi}$. Let us prove that $P(\sigma \cdot a)$ holds.

Since $P(\sigma)$ holds, $\text{Reach}^{\mathcal{E}}(\sigma) = \langle q, \sigma_c \rangle$ and $\sigma_s = \text{output}(\sigma)$. We consider two cases:

- if $a \in \Sigma_u$, then, considering $\sigma'_s = (\sigma_s \cdot a)^{-1} \cdot \sigma_t$, $\sigma_t = \sigma_s \cdot a \cdot \sigma'_s$. Since $a \in \Sigma_u$, rule *pass-uncont* can be applied: let us consider $q' = q$ after a . Then, $\langle q, \sigma_c \rangle \xrightarrow{a/\text{pass-uncont}(a)/a}_{\mathcal{E}} \langle q', \sigma_c \rangle$.

Then, if $\sigma'_s = \epsilon$, $G(q', \sigma_c) = \emptyset$ or $G(q', \sigma_c) = \{\epsilon\}$, meaning that no other rule can be applied, and thus $P(\sigma \cdot a)$ holds.

Otherwise, $\sigma'_s \neq \epsilon$, and thus $\sigma'_s \in G(q', \sigma_c)$, meaning that $G(q', \sigma_c) \neq \emptyset$ and $G(q', \sigma_c) \neq \{\epsilon\}$, thus rule *dump*($\sigma_c(1)$) can be applied. Since $\sigma'_s \preceq \sigma_c$, $\sigma'_s(1) = \sigma_c(1)$, thus if $q_1 = q'$ after $\sigma_c(1)$, $q_1 = q'$ after $\sigma'_s(1)$. If $\sigma'_s(1)^{-1} \cdot \sigma'_s \neq \epsilon$, then $\sigma'_s(1)^{-1} \cdot \sigma'_s \in G(q_1, \sigma_c(1)^{-1} \cdot \sigma_c)$, meaning that rule *dump* can be applied again. Rule *dump* can actually be applied $|\sigma'_s|$ times, since for all $w \preceq \sigma'_s$, if $w \neq \sigma'_s$, then $w^{-1} \cdot \sigma'_s \neq \epsilon$ and $w^{-1} \cdot \sigma'_s \in G(q' \text{ after } w, w^{-1} \cdot \sigma_c)$. Thus, after rule *dump* has been applied $|\sigma'_s|$ times, the configuration reached is $\langle q' \text{ after } \sigma'_s, \sigma'^{-1}_s \cdot \sigma_c \rangle$. Moreover, the output produced by all the rules *dump* is σ'_s . Since no rule can be applied after the $|\sigma'_s|$ applications of the rule *dump*, $\text{output}(\sigma \cdot a) = \text{output}(\sigma) \cdot a \cdot \sigma'_s = \sigma_t$, and $\text{Reach}^{\mathcal{E}}(\sigma \cdot a) = \langle q' \text{ after } \sigma'_s, \sigma'^{-1}_s \cdot \sigma_c \rangle = \langle q \text{ after } a \text{ after } \sigma'_s, \sigma_d \rangle = \langle \text{Reach}(\sigma_s) \text{ after } a \text{ after } \sigma'_s, \sigma_d \rangle = \langle \text{Reach}(\sigma_s \cdot a \cdot \sigma'_s), \sigma_d \rangle = \langle \text{Reach}(\sigma_t), \sigma_d \rangle$.

Thus, if $a \in \Sigma_u$, $P(\sigma \cdot a)$ holds.

- Otherwise, $a \in \Sigma_c$, then, considering $\sigma''_s = \sigma_s^{-1} \cdot \sigma_t$, $\sigma_t = \sigma_s \cdot \sigma''_s$. Since $a \in \Sigma_c$, it is possible to apply the *store-cont* rule, and $\langle q, \sigma_c \rangle$ after $a/\text{store-cont}(a)/\epsilon = \langle q, \sigma_c \cdot a \rangle$. Then as in the case where $a \in \Sigma_u$, rule *dump* can be applied $|\sigma''_s|$ times, meaning that the configuration

reached is then $\langle q \text{ after } (\sigma_c \cdot a)(1) \cdot (\sigma_c \cdot a)(2) \cdots (\sigma_c \cdot a)(|\sigma_s''|), (\sigma_c \cdot a)(|\sigma_s''| + 1) \cdot (\sigma_c \cdot a)(|\sigma_s''| + 2) \cdots (\sigma_c \cdot a)(|\sigma_c \cdot a|) \rangle$. Since $\sigma_s'' \preceq \sigma_c \cdot a$, $(\sigma_c \cdot a)(1) \cdot (\sigma_c \cdot a)(2) \cdots (\sigma_c \cdot a)(|\sigma_s''|) = \sigma_s''$, thus $\text{Reach}(\text{Rules}(\sigma \cdot a)) = \langle q \text{ after } \sigma_s'', \sigma_s''^{-1} \cdot (\sigma_c \cdot a) \rangle = \langle \text{Reach}(\sigma_t), \sigma_d \rangle$. Moreover, $\text{output}(\sigma \cdot a) = \text{output}(\sigma) \cdot \sigma_s'' = \sigma_s \cdot \sigma_s'' = \sigma_t$.

Thus, if $a \in \Sigma_c$, $P(\sigma \cdot a)$ holds.

This means that $P(\sigma) \implies P(\sigma \cdot a)$.

Thus, by induction on σ , for all $\sigma \in \Sigma^*$, $P(\sigma)$ holds. In particular, for all $\sigma \in \Sigma^*$, if $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi$ and $(\sigma, o) \in E_\varphi$, then $o = \sigma_s = \text{output}(\sigma)$. \square

A.2.2 Proofs for the timed setting (Section 4.3)

Proposition 4.6. E_φ as per Definition 4.19 is an enforcement function, as per Definition 4.11.

Proof. We have to prove the two following properties:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t,$
 $\langle (\sigma, t), o_1 \rangle \in E_\varphi \wedge \langle (\sigma, t'), o_2 \rangle \in E_\varphi \implies o_1 \preceq o_2$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall \delta \in \mathbb{R}_{\geq 0}, \forall a \in \Sigma,$
 $\langle \langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o_3 \rangle \in E_\varphi \wedge \langle (\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))), o_4 \rangle \in E_\varphi \implies o_3 \preceq o_4$

For $\sigma \in \text{tw}(\Sigma)$, let $P(\sigma)$ be the predicate “ $\forall t \in \mathbb{R}_{\geq 0}, \forall t' \geq t, \forall (\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma, (\langle (\sigma, t), o_1 \rangle \in E_\varphi \wedge \langle (\sigma, t'), o_2 \rangle \in E_\varphi \wedge \langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o_3 \rangle \in E_\varphi \wedge \langle (\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))), o_4 \rangle \in E_\varphi) \implies (o_1 \preceq o_2 \wedge o_3 \preceq o_4)$ ”.

Let us show by induction that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$:

Induction basis: for $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$ and $t' \geq t$. Then, $\langle (\epsilon, t), \epsilon \rangle \in E_\varphi$, and $\langle (\epsilon, t'), \epsilon \rangle \in E_\varphi$. Moreover, for $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $\langle (\epsilon, \delta), \epsilon \rangle \in E_\varphi$, thus if $\langle ((\delta, a), \delta), o_4 \rangle \in E_\varphi$, then $\epsilon \preceq o_4$. Thus, $P(\epsilon)$ holds.

Induction step: suppose that $P(\sigma)$ holds for some $\sigma \in \text{tw}(\Sigma)$. Then, let us consider $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $t \in \mathbb{R}_{\geq 0}$, and $t' \geq t$. We first prove that the first condition holds. Let us consider $\langle (\sigma, t), o_1 \rangle \in E_\varphi$, $\langle (\sigma, t'), o_2 \rangle \in E_\varphi$, $\langle (\sigma \cdot (\delta, a), t), o'_1 \rangle \in E_\varphi$, and $\langle (\sigma \cdot (\delta, a), t'), o'_2 \rangle \in E_\varphi$. We have to prove that $o'_1 \preceq o'_2$.

Three cases are possible:

1. $t \leq t' < \text{time}(\sigma \cdot (\delta, a))$. Then $\text{obs}(\sigma \cdot (\delta, a), t') = \text{obs}(\sigma, t')$, and $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma, t)$. Let us consider $(\text{obs}(\sigma, t), \langle \sigma_{s1}, \sigma_c \rangle) \in \text{store}_\varphi$ and

$(\text{obs}(\sigma, t'), \langle \sigma_{s2}, \sigma'_c \rangle) \in \text{store}_\varphi$. Then, considering the definition of E_φ (Definition 4.19), $o_1 = \text{obs}(\sigma_{s1}, t) = o'_1$, and $o_2 = \text{obs}(\sigma_{s2}, t') = o'_2$ (since $\text{obs}(\sigma, t) = \text{obs}(\sigma \cdot (\delta, a), t)$ and $\text{obs}(\sigma, t') = \text{obs}(\sigma \cdot (\delta, a), t')$). Following the induction hypothesis, $P(\sigma)$ holds, meaning that $o_1 \preceq o_2$. This means that $o'_1 \preceq o'_2$.

2. $t < \text{time}(\sigma \cdot (\delta, a)) \leq t'$. Then, $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma, t)$, meaning that (see previous case) $o'_1 = o_1$. Let us consider $\langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o_{1a} \rangle \in E_\varphi$ and $\langle (\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))), o_{1b} \rangle \in E_\varphi$. Following the induction hypothesis, since $P(\sigma)$ holds, $o_1 \preceq o_{1a} \preceq o_{1b}$. Thus, we have to show that $o_{1b} \preceq o_2$. Since $\text{time}(\sigma \cdot (\delta, a)) \leq t'$, $\text{obs}(\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a))) = \sigma \cdot (\delta, a) = \text{obs}(\sigma \cdot (\delta, a), t')$, thus if $(\sigma \cdot (\delta, a), \langle \sigma_{s2}, \sigma_c \rangle) \in \text{store}_\varphi$, then $o_{1b} = \text{obs}(\sigma_{s2}, \text{time}(\sigma \cdot (\delta, a)))$, and $o'_2 = \text{obs}(\sigma_{s2}, t')$. Since $\text{time}(\sigma \cdot (\delta, a)) \leq t'$, this means that $o_{1b} \preceq o'_2$.

Thus $o'_1 \preceq o'_2$.

3. $\text{time}(\sigma \cdot (\delta, a)) \leq t \leq t'$. Then, $\text{obs}(\sigma \cdot (\delta, a), t) = \text{obs}(\sigma \cdot (\delta, a), t') = \sigma \cdot (\delta, a)$. Thus, if $(\sigma \cdot (\delta, a), \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, then $o'_1 = \text{obs}(\sigma_{s0}, t)$ and $o'_2 = \text{obs}(\sigma_{s0}, t')$. Since $t \leq t'$, this means that $o'_1 \preceq o'_2$.

Thus, in all cases, the first required condition holds (*i.e.* $o'_1 \preceq o'_2$).

Let us now consider $(\delta', a') \in \mathbb{R}_{\geq 0} \times \Sigma$, $\langle (\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a'))), o'_3 \rangle \in E_\varphi$, and $\langle (\sigma \cdot (\delta, a) \cdot (\delta', a'), \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a'))), o'_4 \rangle \in E_\varphi$. We have to show that $o'_3 \preceq o'_4$. Since $\text{obs}(\sigma \cdot (\delta, a), \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a'))) = \sigma \cdot (\delta, a)$ and $\text{obs}(\sigma \cdot (\delta, a) \cdot (\delta', a'), \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a'))) = \sigma \cdot (\delta, a) \cdot (\delta', a')$, if $(\sigma \cdot (\delta, a), \langle \sigma_{s3}, \sigma_c \rangle) \in \text{store}_\varphi$ and $(\sigma \cdot (\delta, a) \cdot (\delta', a'), \langle \sigma_{s4}, \sigma'_c \rangle) \in \text{store}_\varphi$, then $o'_3 = \text{obs}(\sigma_{s3}, \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a')))$ and $o'_4 = \text{obs}(\sigma_{s4}, \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a')))$. Following the definition of store_φ (Definition 4.19), it is clear that $o'_3 \preceq \sigma_{s4}$. Thus, since $\text{time}(o'_3) \leq \text{time}(\sigma \cdot (\delta, a) \cdot (\delta', a'))$, $o'_3 \preceq o'_4$.

This means that $P(\sigma \cdot (\delta, a))$ holds.

Thus, for any $\sigma \in \text{tw}(\Sigma)$ and $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $P(\sigma) \implies P(\sigma \cdot (\delta, a))$.

Thus, by induction on σ , $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$. Thus E_φ is an enforcement function. \square

Lemma A.14. $\forall \sigma \in \text{tw}(\Sigma), \forall t \geq \text{time}(\sigma), (\sigma \notin \text{Pre}(\varphi, t) \wedge (\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\text{obs}(\sigma_{s0}, t) = \sigma|_{\Sigma_u} \wedge \Pi_\Sigma(\text{nobs}(\sigma_{s0}, t)) \cdot \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c}))$.

Proof. For $\sigma \in \text{tw}(\Sigma)$ and $t \geq \text{time}(\sigma)$, let $P(\sigma, t)$ be the predicate “ $(\sigma \notin \text{Pre}(\varphi, t) \wedge (\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\text{obs}(\sigma_{s0}, t) = \sigma|_{\Sigma_u} \wedge \Pi_\Sigma(\text{nobs}(\sigma_{s0}, t)) \cdot \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c}))$ ”, and $P(\sigma)$ be the predicate “ $\forall t \geq \text{time}(\sigma), P(\sigma, t)$ ”. Let us then prove by induction on σ that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$.

Induction basis: for $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$. Then, $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$. Since $\text{obs}(\epsilon, t) = \epsilon_{|\Sigma_u}$, and $\Pi_\Sigma(\text{nobs}(\epsilon_{|\Sigma_c}, t)) \cdot \epsilon = \Pi_\Sigma(\epsilon_{|\Sigma_c})$, it follows that $P(\epsilon, t)$ holds, and thus $P(\epsilon)$ holds.

Induction step: let us suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, $(\sigma \cdot (\delta, a), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_\varphi$, and $\sigma_s = \text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))$. Let us also consider $t \geq \text{time}(\sigma \cdot (\delta, a))$.

If $\sigma \cdot (\delta, a) \in \text{Pre}(\varphi, t)$, then $P(\sigma \cdot (\delta, a), t)$ trivially holds.

Let us consider that $\sigma \cdot (\delta, a) \notin \text{Pre}(\varphi, t)$.

- If $a \in \Sigma_u$, then $\sigma_{t0} = \sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) \cdot \sigma'_s$ for some $\sigma'_s \in \text{tw}(\Sigma)$. Since $\sigma \cdot (\delta, a) \notin \text{Pre}(\varphi, t)$, this means that for any $t' \leq t$, $G(\text{Reach}((\sigma \cdot (\delta, a))_{|\Sigma_u}, t'), \Pi_\Sigma(\text{obs}(\sigma \cdot (\delta, a), t')_{|\Sigma_c})) = \emptyset$. This means that for any $t' \leq t - \text{time}(\sigma \cdot (\delta, a))$, $t' \notin T(\text{Reach}((\sigma \cdot (\delta, a))_{|\Sigma_u}), \Pi_\Sigma(\sigma \cdot (\delta, a))_{|\Sigma_c})$. Now, by induction hypothesis, since $\sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$ (otherwise $\sigma \cdot (\delta, a)$ would be in $\text{Pre}(\varphi, t)$), $\sigma_{|\Sigma_u} = \sigma_s$, and $\Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))))_{|\Sigma_c} \cdot \sigma_c = \Pi_\Sigma(\sigma)_{|\Sigma_c}$. Thus, for any $t' \leq t - \text{time}(\sigma \cdot (\delta, a))$, $t' \notin T(\text{Reach}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c)$. Thus, $\text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) = \epsilon$. It follows that $\text{obs}(\sigma_{t0}, t) = \sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) \cdot \text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) = \sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) = (\sigma \cdot (\delta, a))_{|\Sigma_u}$ and $\Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d = \sigma'_s \cdot \sigma_d = \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c = \Pi_\Sigma(\sigma)_{|\Sigma_c} = \Pi_\Sigma((\sigma \cdot (\delta, a))_{|\Sigma_c})$. Thus $P(\sigma \cdot (\delta, a), t)$ holds.

- Otherwise, $a \in \Sigma_c$, and there exists σ''_s such that $\sigma_{t0} = \sigma_s \cdot \sigma''_s$. Since $\sigma \cdot (\delta, a) \notin \text{Pre}(\varphi, t)$, for any $t' \leq t$, $G(\text{Reach}((\sigma \cdot (\delta, a))_{|\Sigma_u}, t'), \Pi_\Sigma(\text{obs}(\sigma \cdot (\delta, a), t')_{|\Sigma_c})) = \emptyset$. Thus, for any $t' \leq t - \text{time}((\sigma \cdot (\delta, a))_{|\Sigma_u})$, $t' \notin T(\text{Reach}((\sigma \cdot (\delta, a))_{|\Sigma_u}), \Pi_\Sigma((\sigma \cdot (\delta, a))_{|\Sigma_c}))$. Now, by induction hypothesis, considering that $(\sigma \cdot (\delta, a))_{|\Sigma_u} = \sigma_{|\Sigma_u}$ and $(\sigma \cdot (\delta, a))_{|\Sigma_c} = \sigma_{|\Sigma_c} \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_{|\Sigma_c}), a)$, and since $\sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, for any $t' \leq t - \text{time}(\sigma \cdot (\delta, a))$, $t' \notin T(\text{Reach}(\sigma_s, \text{time}(\sigma \cdot (\delta, a))), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \cdot a)$. Thus, $\text{obs}(\sigma''_s -_t (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s)), t - \text{time}(\sigma \cdot (\delta, a))) = \epsilon$. Thus, $\text{obs}(\sigma''_s, t - \text{time}(\sigma \cdot (\delta, a)) + \text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s)) -_t (t - \text{time}(\sigma \cdot (\delta, a))) = \epsilon$, meaning that $\text{obs}(\sigma''_s, t - \text{time}(\sigma_s)) = \epsilon$.

Thus, $\text{obs}(\sigma_{t0}, t) = \sigma_s \cdot \text{obs}(\sigma''_s, t - \text{time}(\sigma_s)) = \sigma_s = \sigma_{|\Sigma_u}$, and $\Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d = \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \cdot a = \Pi_\Sigma(\sigma_{|\Sigma_c}) \cdot a = \Pi_\Sigma((\sigma \cdot (\delta, a))_{|\Sigma_c})$.

Thus $P(\sigma \cdot (\delta, a), t)$ holds.

In both cases, $P(\sigma \cdot (\delta, a), t)$ holds. Thus, it holds for any $t \geq \text{time}(\sigma \cdot (\delta, a))$, meaning that $P(\sigma \cdot (\delta, a))$ holds.

This means that for any $\sigma \in \text{tw}(\Sigma)$ and $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $P(\sigma) \implies P(\sigma \cdot (\delta, a))$.

Thus, we have shown by induction on σ that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$. \square

Lemma A.15. $\forall q \in Q, \forall w \in \Sigma_c^*, \forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \sigma \in G(q, w) \implies \text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t))) \in G(q \text{ after } (\sigma, t), \Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w).$

Proof. Let us consider q, w and σ such that $\sigma \in G(q, w)$, and $t \in \mathbb{R}_{\geq 0}$. Following the definition of G (Definition 4.18), this means that the three following properties hold:

1. $\Pi_\Sigma(\sigma) \preceq w$,
2. $q \text{ after } \sigma \in F_G$,
3. $\forall t \in \mathbb{R}_{\geq 0}, \forall v \in V_s,$
 $q \text{ after } (\sigma, t) \in v \implies \langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w), 1 \rangle \in W_0.$

Now, considering $\sigma' = \text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t)))$, σ' satisfies the following properties:

1. $\Pi_\Sigma(\sigma') = \Pi_\Sigma(\text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t))))$
 $= \Pi_\Sigma(\text{nobs}(\sigma, t)),$
thus,
 $\Pi_\Sigma(\text{obs}(\sigma, t)) \cdot \Pi_\Sigma(\sigma') = \Pi_\Sigma(\text{obs}(\sigma, t)) \cdot \Pi_\Sigma(\text{nobs}(\sigma, t))$
 $= \Pi_\Sigma(\sigma).$

Since $\Pi_\Sigma(\sigma) \preceq w$, this means that:

$$\Pi_\Sigma(\sigma') \preceq \Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w.$$

2. $(q \text{ after } (\sigma, t)) \text{ after } \sigma' = (q \text{ after } (\sigma, t)) \text{ after } (\text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t))))$
 $= q \text{ after } \sigma.$

Thus, $(q \text{ after } (\sigma, t)) \text{ after } \sigma' \in F_G$.

3. For $t' \in \mathbb{R}_{\geq 0}$,
 $(q \text{ after } (\sigma, t)) \text{ after } (\sigma', t') = (q \text{ after } (\sigma, t)) \text{ after } (\text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t))), t')$
 $= q \text{ after } (\sigma, t + t').$

Since $t + t' \in \mathbb{R}_{\geq 0}$, then if $v \in V_s$ is such that $(q \text{ after } (\sigma, t)) \text{ after } (\sigma', t') \in v$, then $\langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma, t + t'))^{-1} \cdot w), 1 \rangle \in W_0$. Moreover, since $t \geq \text{time}(\text{obs}(\sigma, t))$, $\Pi_\Sigma(\text{obs}(\sigma, t + t'))^{-1} \cdot w = \Pi_\Sigma(\text{obs}(\sigma', t'))^{-1} \cdot (\Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w)$. Thus, $\langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma', t'))^{-1} \cdot (\Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w)), 1 \rangle \in W_0$.

This means that $\sigma' = \text{nobs}(\sigma, t) -_t (t - \text{time}(\text{obs}(\sigma, t))) \in G(q \text{ after } (\sigma, t), \Pi_\Sigma(\text{obs}(\sigma, t))^{-1} \cdot w)$. \square

Proposition 4.7. E_φ is sound with respect to φ in $\text{Pre}(\varphi)$ as per Definition 4.12.

Proof. Notation from Definition 4.19 is to be used in this proof: for $\sigma \in \text{tw}(\Sigma)$, if $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $t = \text{time}(\sigma \cdot (\delta, a))$, and $\sigma_s = \text{obs}(\sigma_{s0}, t)$, then, for $q \in Q$, and $w \in \Sigma_c^*$,

$$\begin{aligned} T(q, w) &= \{t' \in \mathbb{R}_{\geq 0} \mid \forall t'' < t', G(q \text{ after } (\epsilon, t''), w) = \emptyset\}, \\ \kappa_\varphi(q, w) &= \min_{\text{lex}}(\max_{\preceq}(\{\epsilon\} \cup \bigcup_{t' \in T(q, w)} \{w' +_t t' \mid w' \in G(q \text{ after } (\epsilon, t'), w)\})) \\ \text{buf}_c &= \Pi_\Sigma(\text{nobs}(\sigma_{s0}, t)) \cdot \sigma_c, \end{aligned}$$

and

$$\begin{aligned} \sigma'_s &= \kappa_\varphi(\text{Reach}(\sigma_s \cdot (t - \text{time}(\sigma_s), a)), \text{buf}_c) & \sigma'_c &= \Pi_\Sigma(\sigma'_s)^{-1} \cdot \text{buf}_c, \\ \sigma''_s &= \kappa_\varphi(\text{Reach}(\sigma_s, t), \text{buf}_c \cdot a) +_t (t - \text{time}(\sigma_s)) & \sigma''_c &= \Pi_\Sigma(\sigma''_s)^{-1} \cdot (\text{buf}_c \cdot a). \end{aligned}$$

We have to prove that for any $(\sigma, t) \in \text{Pre}(\varphi)$, there exists $t' \geq t$ such that for any $t'' \geq t'$, if $\langle (\sigma, t''), o \rangle \in E_\varphi$, then $o \models \varphi$.

For $\sigma \in \text{tw}(\Sigma)$, and $t \geq \text{time}(\sigma)$, let $P(\sigma, t)$ be the predicate “ $((\sigma, t) \in \text{Pre}(\varphi) \wedge (\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_\varphi) \implies (\sigma_s \models \varphi \wedge \text{nobs}(\sigma_s, t) -_t (t - \text{time}(\text{obs}(\sigma_s, t))) \in G(\text{Reach}(\sigma_s, t), \Pi_\Sigma(\text{nobs}(\sigma_s, t)) \cdot \sigma_c))$ ”. Let also $P(\sigma)$ be the predicate: “ $\forall t \geq \text{time}(\sigma), P(\sigma, t)$ ”. Let us show by induction that for any $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds.

Induction basis: for $\sigma = \epsilon$, let us consider $t \in \mathbb{R}_{\geq 0}$.

- If $\epsilon \notin \text{Pre}(\varphi, t)$, then, $P(\epsilon)$ trivially holds.
- Otherwise, $\epsilon \in \text{Pre}(\varphi, t)$. Then, following the definition of $\text{Pre}(\epsilon, t)$ (Definition 4.20), there exists $t' \leq t$ such that $G(\text{Reach}(\epsilon_{|\Sigma_u}, t'), \epsilon) \neq \emptyset$, meaning that $G(\text{Reach}(\epsilon, t'), \epsilon) \neq \emptyset$. Thus, following the definition of $G(\text{Reach}(\epsilon, t'), \epsilon)$ (Definition 4.18), $\epsilon \in G(\text{Reach}(\epsilon, t'), \epsilon)$, and $\text{Reach}(\epsilon) \in F_G$, thus $\epsilon \models \varphi$. Since $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$ and $\epsilon \models \varphi$, $P(\epsilon, t)$ holds.

Thus, in both cases, $P(\epsilon, t)$ holds, meaning that $P(\epsilon)$ holds.

Induction step: suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $t \geq \text{time}(\sigma \cdot (\delta, a))$, $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, $\sigma_s = \text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))$, $(\sigma \cdot (\delta, a), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_\varphi$, and $\sigma_t = \text{obs}(\sigma_{t0}, t)$. We have to prove that $(\sigma \cdot (\delta, a), t) \in \text{Pre}(\varphi) \implies \sigma_t \models \varphi \wedge \text{nobs}(\sigma_t, t) -_t (t - \text{time}(\text{obs}(\sigma_t, t))) \in G(\text{Reach}(\sigma_t, t), \Pi_\Sigma(\text{nobs}(\sigma_t, t)) \cdot \sigma_d)$.

- If $\sigma \cdot (\delta, a) \notin \text{Pre}(\varphi, t)$, then $P(\sigma \cdot (\delta, a), t)$ trivially holds.

- If $\sigma \cdot (\delta, a) \in \text{Pre}(\varphi, t) \wedge \sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, then, since $\sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, following lemma A.14, since $\text{obs}(\sigma, \text{time}(\sigma \cdot (\delta, a))) = \sigma$, $\text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) = \sigma_s = \text{obs}(\sigma|_{\Sigma_u}, \text{time}(\sigma \cdot (\delta, a))) = \sigma|_{\Sigma_u}$ and $\Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c})$. Since $\sigma \cdot (\delta, a) \in \text{Pre}(\varphi, t)$, and $\sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, following the definition of $\text{Pre}(\varphi, t)$ and $\text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$ (Definition 4.20), there exists $t' \in \mathbb{R}_{\geq 0}$ such that $\text{time}(\sigma \cdot (\delta, a)) \leq t' \leq t$, and $G(\text{Reach}((\sigma \cdot (\delta, a))|_{\Sigma_u}, t'), \Pi_\Sigma(\text{obs}(\sigma \cdot (\delta, a), t')|_{\Sigma_c})) \neq \emptyset$. Let us consider the minimum such t' . Since $t' \geq \text{time}(\sigma \cdot (\delta, a))$, then $\text{obs}(\sigma \cdot (\delta, a), t') = \sigma \cdot (\delta, a)$. This means that:

$$G(\text{Reach}((\sigma \cdot (\delta, a))|_{\Sigma_u}, t'), \Pi_\Sigma((\sigma \cdot (\delta, a))|_{\Sigma_c})) \neq \emptyset. \quad (\text{A.2})$$

- If $a \in \Sigma_u$, then considering that $(\sigma \cdot (\delta, a))|_{\Sigma_u} = \sigma|_{\Sigma_u} \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) = \sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)$, and $\Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c = \Pi_\Sigma(\sigma|_{\Sigma_c}) = \Pi_\Sigma((\sigma \cdot (\delta, a))|_{\Sigma_c})$, (A.2) becomes:

$$G(\text{Reach}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a), t'), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c) \neq \emptyset.$$

Let us consider $\delta' = \text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s)$, such that $\text{time}(\sigma \cdot (\delta, a)) = \text{time}(\sigma_s \cdot (\delta', a))$, and $t'' = t' - \text{time}(\sigma \cdot (\delta, a))$.

Then, t' is the minimum number such that $G(\text{Reach}(\sigma_s \cdot (\delta', a), t'), \text{buf}_c) \neq \emptyset$. Therefore, $t'' \in T(\text{Reach}(\sigma_s \cdot (\delta', a), \text{buf}_c))$.

Thus, there exists $w' \in G(\text{Reach}(\sigma_s \cdot (\delta', a))$ after (ϵ, t'') , buf_c such that $\sigma'_s = w' \cdot_t t''$. Thus, $\sigma'_s \cdot_t t'' \in G(\text{Reach}(\sigma_s \cdot (\delta', a), t'), \text{buf}_c)$.

Now, note that:

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) &= \text{obs}(\sigma_{t0}, t)^{-1} \cdot \sigma_{t0} \\ &= \text{obs}(\sigma_s \cdot (\delta', a) \cdot \sigma'_s, t)^{-1} \cdot \sigma_{t0} \\ \text{Since } t &\geq \text{time}(\sigma \cdot (\delta, a)) = \text{time}(\sigma_s \cdot (\delta', a)), \\ \text{nobs}(\sigma_{t0}, t) &= (\sigma_s \cdot (\delta', a) \cdot \text{obs}(\sigma'_s, t - \text{time}(\sigma_s \cdot (\delta', a))))^{-1} \cdot \sigma_{t0} \\ &= \text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a)))^{-1} \cdot \\ &\quad ((\sigma_s \cdot (\delta', a))^{-1} \cdot (\sigma_s \cdot (\delta', a) \cdot \sigma'_s)) \\ &= \text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a)))^{-1} \cdot \sigma'_s \\ &= \text{nobs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) \end{aligned}$$

We know that $\sigma'_s \cdot_t t'' \in G(\text{Reach}(\sigma_s \cdot (\delta', a), t'), \text{buf}_c)$, thus following lemma A.15, since $t \geq t', t - t' \geq 0$, and

$$\begin{aligned} \text{nobs}(\sigma'_s \cdot_t t'', t - t') \cdot_t (t - t' - \text{time}(\text{obs}(\sigma'_s \cdot_t t'', t - t'))) &\in \\ G(\text{Reach}(\sigma_s \cdot (\delta', a), t') \text{ after } (\sigma'_s \cdot_t t'', t - t'), & \\ \Pi_\Sigma(\text{obs}(\sigma'_s \cdot_t t'', t - t'))^{-1} \cdot \text{buf}_c) & \end{aligned} \quad (\text{A.3})$$

Now, note that for any $\sigma \in \text{tw}(\Sigma)$, $t \in \mathbb{R}_{\geq 0}$ and $t' \in \mathbb{R}_{\geq 0}$,

$$\text{nobs}(\sigma \dashv_t t, t') = \begin{cases} \text{nobs}(\sigma, t + t') \dashv_t t' & \text{if } \text{delay}(\sigma(1)) > t + t' \\ \text{nobs}(\sigma, t + t') & \text{otherwise} \end{cases}$$

The reason is that the operator \dashv_t affects only the first delay of the word, thus if this delay is in $\text{obs}(\sigma \dashv_t t, t')$, *i.e.* $\text{delay}(\sigma(1)) \geq t + t'$, the remaining events are not changed by the \dashv_t operator.

Thus, if $\text{delay}(\sigma'_s(1)) > t - t' + t'' = t - t' + t' - \text{time}(\sigma \cdot (\delta, a)) = t - \text{time}(\sigma \cdot (\delta, a))$, then

$$\begin{aligned} \text{nobs}(\sigma'_s \dashv_t t'', t - t') &= \text{nobs}(\sigma'_s, t - t' + t'') \dashv_t t'' \\ &= \text{nobs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) \dashv_t t'' \end{aligned}$$

Moreover, since $\text{delay}(\sigma'_s(1)) > t - \text{time}(\sigma \cdot (\delta, a))$, $\text{obs}(\sigma'_s \dashv_t t'', t - t') = \epsilon$, and $\text{obs}(\sigma_{t0}, t) = \sigma_s \cdot (\delta, a)$, thus:

$$\begin{aligned} \text{nobs}(\sigma'_s \dashv_t t'', t - t') \dashv_t (t - t' - \text{time}(\text{obs}(\sigma'_s \dashv_t t'', t - t'))) \\ &= (\text{nobs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) \dashv_t t'') \dashv_t (t - t') \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - t' + t'') \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\sigma \cdot (\delta, a))) \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \end{aligned}$$

On the other hand, if $\text{delay}(\sigma'_s(1)) \leq t - \text{time}(\sigma \cdot (\delta, a))$, then

$$\text{nobs}(\sigma'_s \dashv_t t'', t - t') = \text{nobs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a)))$$

Moreover, since $\text{delay}(\sigma'_s(1)) \leq t - \text{time}(\sigma \cdot (\delta, a))$, $\text{obs}(\sigma'_s \dashv_t t'', t - t') = \text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) \dashv_t t''$, thus

$$\begin{aligned} \text{nobs}(\sigma'_s \dashv_t t'', t - t') \dashv_t (t - t' - \text{time}(\text{obs}(\sigma'_s \dashv_t t'', t - t'))) \\ &= \text{nobs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))) \\ &\quad \dashv_t (t - t' - \text{time}(\text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a)))) \dashv_t t'') \\ &= \text{nobs}(\sigma_{t0}, t) \\ &\quad \dashv_t (t - t' + t'' - \text{time}(\text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))))) \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\sigma \cdot (\delta, a)) - \\ &\quad \text{time}(\text{obs}(\sigma'_s, t - \text{time}(\sigma \cdot (\delta, a))))) \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\sigma \cdot (\delta, a)) - \\ &\quad (\text{time}(\text{obs}(\sigma_{t0}, t)) - \text{time}(\sigma \cdot (\delta, a)))) \\ &= \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \end{aligned}$$

Thus, in both cases, (A.3) becomes:

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) \dashv_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) &\in \\ \text{G}(\text{Reach}(\sigma_s \cdot (\delta', a), t') \text{ after } (\sigma'_s \dashv_t t'', t - t'), \\ \Pi_\Sigma(\text{obs}(\sigma'_s \dashv_t t'', t - t'))^{-1} \cdot \text{buf}_c) \end{aligned}$$

Now, since $t' \geq \text{time}(\sigma \cdot (\delta, a))$,

$$\begin{aligned}
& \text{Reach}(\sigma_s.(\delta', a), t') \text{ after } (\sigma'_s -_t t'', t - t') \\
&= \text{Reach}(\sigma_s.(\delta', a)) \text{ after} \\
&\quad (\epsilon, t' - \text{time}(\sigma_s.(\delta', a))) \text{ after } (\sigma'_s -_t t'', t - t') \\
&= \text{Reach}(\sigma_s.(\delta', a)) \text{ after} \\
&\quad ((\sigma'_s -_t t'') +_t t'', t - t' + t'') \\
&= \text{Reach}(\sigma_s.(\delta', a)) \text{ after } (\sigma'_s, t - \text{time}(\sigma.(\delta, a))) \\
&= \text{Reach}(\sigma_s.(\delta', a). \sigma'_s, t) \\
&= \text{Reach}(\sigma_{t0}, t) \\
&\text{and} \\
&\Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)). \sigma_d \\
&= \Pi_\Sigma(\text{nobs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a)))) . (\Pi_\Sigma(\sigma'_s)^{-1} . \text{buf}_c) \\
&= \Pi_\Sigma(\text{obs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a))))^{-1} . \sigma'_s. \\
&\quad (\Pi_\Sigma(\sigma'_s)^{-1} . \text{buf}_c) \\
&= \Pi_\Sigma(\text{obs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a))))^{-1} . \Pi_\Sigma(\sigma'_s). \\
&\quad (\Pi_\Sigma(\sigma'_s)^{-1} . \text{buf}_c) \\
&= \Pi_\Sigma(\text{obs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a))))^{-1} . \text{buf}_c \\
&\text{On the other hand,} \\
&\Pi_\Sigma(\text{obs}(\sigma'_s -_t t'', t - t'))^{-1} . \text{buf}_c \\
&\quad = \Pi_\Sigma(\text{obs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a))) -_t t'')^{-1} . \text{buf}_c \\
&\quad = \Pi_\Sigma(\text{obs}(\sigma'_s, t - \text{time}(\sigma.(\delta, a))))^{-1} . \text{buf}_c \\
&\text{Thus, } \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)). \sigma_d = \Pi_\Sigma(\text{obs}(\sigma'_s -_t t'', t - t'))^{-1} . \text{buf}_c. \\
&\text{Considering all this, (A.3) becomes:}
\end{aligned}$$

$$\begin{aligned}
& \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\
& \quad G(\text{Reach}(\sigma_{t0}, t), \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)). \sigma_d)
\end{aligned}$$

– Otherwise, $a \in \Sigma_c$. Then, $(\sigma.(\delta, a))|_{\Sigma_u} = \sigma|_{\Sigma_u} = \sigma_s$, and $\Pi_\Sigma((\sigma.(\delta, a))|_{\Sigma_c}) = \Pi_\Sigma(\sigma|_{\Sigma_c}). a = \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma.(\delta, a)))) . \sigma_c . a$. Thus, (A.2) becomes:

$$G(\text{Reach}(\sigma_s, t'), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma.(\delta, a)))) . \sigma_c . a) \neq \emptyset$$

Since t' is the minimum date that satisfies this equation, $t' - \text{time}(\sigma.(\delta, a)) \in T(\text{Reach}(\sigma_s, \text{time}(\sigma.(\delta, a))), \text{buf}_c . a)$.

Thus, there exists $w' \in G(\text{Reach}(\sigma_s, \text{time}(\sigma.(\delta, a))) \text{ after } (\epsilon, t' - \text{time}(\sigma.(\delta, a))), \text{buf}_c . a)$ such that $\sigma''_s = (w' +_t t' - \text{time}(\sigma.(\delta, a))) +_t (\text{time}(\sigma.(\delta, a)) - \text{time}(\sigma_s))$. Thus, $\sigma''_s -_t (t' - \text{time}(\sigma.(\delta, a)) + \text{time}(\sigma.(\delta, a)) - \text{time}(\sigma_s)) = \sigma''_s -_t (t' - \text{time}(\sigma_s)) \in G(\text{Reach}(\sigma_s, t'), \text{buf}_c . a)$. Let us consider $t'' = t' - \text{time}(\sigma_s)$, such that $\sigma''_s -_t t'' \in G(\text{Reach}(\sigma_s, t'), \text{buf}_c . a)$.

Now,

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) &= \text{obs}(\sigma_{t0}, t)^{-1} \cdot \sigma_{t0} \\ &= \text{obs}(\sigma_s \cdot \sigma_s'', t)^{-1} \cdot \sigma_{t0} \end{aligned}$$

Since $\text{time}(\sigma_s) \leq t$, it follows that

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) &= (\sigma_s \cdot \text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\sigma_s \cdot \sigma_s'') \\ &= \text{obs}(\sigma_s'', t - \text{time}(\sigma_s))^{-1} \cdot (\sigma_s^{-1} \cdot (\sigma_s \cdot \sigma_s'')) \\ &= \text{obs}(\sigma_s'', t - \text{time}(\sigma_s))^{-1} \cdot \sigma_s'' \\ &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) \end{aligned}$$

We know that $\sigma_s'' -_t t'' \in G(\text{Reach}(\sigma_s, t'), \text{buf}_c \cdot a)$ and that $t \geq t'$ meaning that $t - t' \geq 0$. Thus, following lemma A.15:

$$\begin{aligned} &\text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'' -_t t'', t - t'))) \in \\ &G(\text{Reach}(\sigma_s, t') \text{ after } (\sigma_s'' -_t t'', t - t'), \\ &\Pi_\Sigma(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a)) \end{aligned}$$

If $\text{delay}(\sigma_s''(1)) > t - \text{time}(\sigma_s)$ (i.e. $\text{delay}((\sigma_s'' -_t t'')(1)) > t - t'$), then:

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') &= \text{nobs}(\sigma_s'', t - t' + t'') -_t t'' \\ &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t t'' \end{aligned}$$

and $\text{obs}(\sigma_s'' -_t t'', t - t') = \epsilon$ and $\text{obs}(\sigma_{t0}, t) = \sigma_s$. Thus,

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'' -_t t'', t - t'))) &= (\text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t t'') -_t (t - t') \\ &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t (t - t' + t'') \\ &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\sigma_s)) \\ &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \end{aligned}$$

Otherwise, $\text{delay}(\sigma_s''(1)) \leq t - \text{time}(\sigma_s)$, and then $\text{nobs}(\sigma_s'' -_t t'', t - t') = \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s))$, thus:

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'' -_t t'', t - t'))) &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t \\ &\quad (t - t' - \text{time}(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t t'')) \\ &= \text{nobs}(\sigma_{t0}, t) -_t \\ &\quad (t - t' - (\text{time}(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s))) - t'')) \\ &= \text{nobs}(\sigma_{t0}, t) -_t \\ &\quad (t - t' + t'' - (\text{time}(\text{obs}(\sigma_{t0}, t)) - \text{time}(\sigma_s))) \\ &= \text{nobs}(\sigma_{t0}, t) -_t \\ &\quad (t - \text{time}(\sigma_s) - \text{time}(\text{obs}(\sigma_{t0}, t)) + \text{time}(\sigma_s)) \\ &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \end{aligned}$$

Thus, in both cases, this means that

$$\begin{aligned} & \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\ & \quad G(\text{Reach}(\sigma_s, t') \text{ after } (\sigma_s'' -_t t'', t - t'), \\ & \quad \Pi_\Sigma(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a)) \end{aligned}$$

$$\begin{aligned} & \text{Now, since } t' \geq \text{time}(\sigma_s), \\ & \text{Reach}(\sigma_s, t') \text{ after } (\sigma_s'' -_t t'', t - t') \\ & \quad = \text{Reach}(\sigma_s) \text{ after } (\epsilon, t' - \text{time}(\sigma_s)) \text{ after} \\ & \quad \quad (\sigma_s'' -_t t'', t - t') \\ & \quad = \text{Reach}(\sigma_s) \text{ after } (\epsilon, t'') \text{ after } (\sigma_s'' -_t t'', t - t') \\ & \quad = \text{Reach}(\sigma_s) \text{ after } ((\sigma_s'' -_t t'') +_t t'', t - t' + t'') \\ & \quad = \text{Reach}(\sigma_s) \text{ after } (\sigma_s'', t - \text{time}(\sigma_s)) \\ & \quad = \text{Reach}(\sigma_s \cdot \sigma_s'', t) \end{aligned}$$

$$\begin{aligned} & \text{and} \\ & \Pi_\Sigma(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t t'')^{-1} \cdot (\text{buf}_c \cdot a) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\text{buf}_c \cdot a) \end{aligned}$$

Moreover, since

$$\begin{aligned} & \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_{t0}, t))^{-1} \cdot \sigma_{t0} \cdot \sigma_d \\ & \quad = (\Pi_\Sigma(\text{obs}(\sigma_{t0}, t))^{-1} \cdot \Pi_\Sigma(\sigma_{t0})) \cdot \sigma_d \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s \cdot \sigma_s'', t))^{-1} \cdot (\Pi_\Sigma(\sigma_{t0}) \cdot \sigma_d) \\ & \quad = \Pi_\Sigma(\sigma_s \cdot \text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\Pi_\Sigma(\sigma_{t0}) \cdot \sigma_d) \\ & \quad = (\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s))))^{-1} \cdot \\ & \quad \quad (\Pi_\Sigma(\sigma_s \cdot \sigma_s'') \cdot \sigma_d) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot \\ & \quad \quad (\Pi_\Sigma(\sigma_s)^{-1} \cdot (\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_s'') \cdot \sigma_d)) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\Pi_\Sigma(\sigma_s'') \cdot \sigma_d) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot \\ & \quad \quad (\Pi_\Sigma(\sigma_s'') \cdot (\Pi_\Sigma(\sigma_s'')^{-1} \cdot (\text{buf}_c \cdot a))) \\ & \quad = \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\text{buf}_c \cdot a) \end{aligned}$$

it follows that $\Pi_\Sigma(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a) = \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d$, and thus (A.3) becomes:

$$\begin{aligned} & \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\ & \quad G(\text{Reach}(\sigma_{t0}, t), \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d) \end{aligned}$$

Thus, in both cases,

$$\begin{aligned} & \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\ & \quad G(\text{Reach}(\sigma_{t0}, t), \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d). \end{aligned}$$

Since this holds for any $t \in \mathbb{R}_{\geq 0}$, in particular, if $t = \text{time}(\sigma_{t0})$, this means that $\epsilon \in G(\text{Reach}(\sigma_{t0}), \sigma_d)$, meaning that $\text{Reach}(\sigma_{t0})$ after $\epsilon = \text{Reach}(\sigma_{t0}) \in F_G$. This means that $\sigma_{t0} \models \varphi$.

Thus, if $\sigma \cdot (\delta, a) \in \text{Pre}(\varphi, t) \wedge \sigma \notin \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, $P(\sigma) \implies P(\sigma \cdot (\delta, a), t)$.

- Otherwise, $\sigma \cdot (\delta, a) \in \text{Pre}(\varphi, t)$ and $\sigma \in \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$. Then, by induction hypothesis:

$$\begin{aligned} & \text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) -_t \\ & (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \in \\ & G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c) \end{aligned}$$

- If $a \in \Sigma_u$, since $G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \neq \emptyset$, following the definition of G (Definition 4.18), it means that there exists $\sigma' \in \text{tw}(\Sigma_c)$ such that the three following properties hold:

1. $\Pi_\Sigma(\sigma') \preceq \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c$,
2. $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))$ after $\sigma' \in F \times \mathbb{R}_{\geq 0}$,
3. for any $t' \in \mathbb{R}_{\geq 0}$, if $v \in V_s$ is such that $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \in v$, then $\langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma', t'))^{-1} \cdot \text{buf}_c), 1 \rangle \in W_0$.

In particular, for item 3, with $t' = 0$, we get $\langle v, \text{maxbuffer}(\text{buf}_c), 1 \rangle \in W_0$, with $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \in v$. Thus, following the edge $(\langle v, \text{maxbuffer}(\text{buf}_c), 1 \rangle, \langle v \text{ after } a, \text{maxbuffer}(\text{buf}_c), 0 \rangle) \in E_3$, since W_0 is the winning region for player 0, it follows that $\langle v \text{ after } a, \text{maxbuffer}(\text{buf}_c), 0 \rangle \in W_0$.

Thus, there exists a winning strategy for player 0 from node $\langle v, \text{maxbuffer}(\text{buf}_c), 0 \rangle$, meaning that there exists a play π such that the set of nodes visited infinitely often by π , noted $\text{inf}(\pi)$, is such that $\text{inf}(\pi) \cap F_G \times \Sigma_c^n \times \{0, 1\} \neq \emptyset$, and $\pi(1) = \langle v, \text{maxbuffer}(\text{buf}_c), 0 \rangle$. Moreover, we can choose π such that no edge from E_3 or E_4 (corresponding to receiving uncontrollable or controllable events, respectively) is taken when playing π . This is possible since W_0 is the winning region for player 0, thus it is winning for all the strategies of player 1, and the edges of E_3 and E_4 leave a node belonging to player 1. Now, since the only cycles in the graph without the edges of E_3 and E_4 are cycles of the form $\langle v, w, 0 \rangle \langle v, w, 1 \rangle \langle v, w, 0 \rangle$, with $(\langle v, w, 0 \rangle, \langle v, w, 1 \rangle) \in E_1$ and $(\langle v, w, 1 \rangle, \langle v, w, 0 \rangle) \in E_6$, it follows that π ends with such a cycle repeated indefinitely, *i.e.* $\pi = \pi_0 \cdot (\langle v_e, w_e, 0 \rangle \cdot \langle v_e, w_e, 1 \rangle)^\omega$ for some finite π_0 . Thus, $\text{inf}(\pi) = \{\langle v_e, w_e, 0 \rangle, \langle v_e, w_e, 1 \rangle\}$, meaning that $v_e \subseteq F_G$.

This allows us to associate a word σ' to π . To build it, we first build a sequence in $Q \times \mathbb{R}_{\geq 0} \times \text{tw}(\Sigma_c)$ by induction as follows:

$$(q_0, \delta_0, w_0) = (\text{Reach}(\sigma_s, \text{time}(\sigma \cdot (\delta, a))) \text{ after } (0, a), 0, \epsilon)$$

and, for $i \in \mathbb{N}$,

$$(q_{i+1}, \delta_{i+1}, w_{i+1}) = \begin{cases} (q_i, \delta_i, w_i) & \text{if } (\pi(i), \pi(i+1)) \in E_1 \cup E_6 \\ (q_i \text{ after } (\delta_i, c), 0, w_i \cdot (\delta_i, c)) & \text{if } (\pi(i), \pi(i+1)) \in E_2, \text{ with } \pi(i) = \langle v, c \cdot w, 0 \rangle \text{ for some } (c, w) \in \Sigma_c \times \Sigma_c^* \\ (q_i, \delta_i + \delta, w_i) & \text{if } (\pi(i), \pi(i+1)) \in E_5, \text{ with } \delta = \min(\{\delta' \in \mathbb{R}_{\geq 0} \mid q_i \text{ after } (\epsilon, \delta_i + \delta') \in \Pi_1(\pi(i+1))\}) \end{cases}$$

Now since $\pi = \pi_0 \cdot (\langle v_e, w_e, 0 \rangle \cdot \langle v_e, w_e, 1 \rangle)^\omega$, there exists $n \in \mathbb{N}$ such that for any $n' \geq n$, $(\pi(n'), \pi(n'+1)) \in E_1 \cup E_6$, meaning that $(q_{n'}, \delta_{n'}, w_{n'}) = (q_n, \delta_n, w_n)$. Thus, the sequence stabilises. Let us consider $\sigma' = w_n$, where w_n is the third component of the previous sequence when it is stabilised. Then, σ' satisfies:

1. $\Pi_\Sigma(\sigma') \preceq \text{maxbuffer}(\text{buf}_c)$, because there is no edge $(\pi(i), \pi(i+1))$ belonging to E_3 or E_4 , and $\Pi_2(\pi(1)) = \text{maxbuffer}(\text{buf}_c)$.
2. $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \text{ after } (0, a) \text{ after } \sigma' \in F_G$, because it belongs to $v_e \subseteq F_G$ (v_e is such that $\pi = \pi_0 \cdot (\langle v_e, w_e, 0 \rangle \cdot \langle v_e, w_e, 1 \rangle)^\omega$).
3. For any $t' \in \mathbb{R}_{\geq 0}$, if $v \in V_s$ is such that $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \text{ after } (0, a) \text{ after } (\sigma', t') \in v$, then $\langle v, \Pi_\Sigma(\text{nobs}(\sigma', t'))^{-1} \cdot \text{buf}_c, 1 \rangle \in W_0$, because π is winning for player 0. By construction of σ' , and because of the different constraints required on \mathcal{G}_s , this implies that all states $v \in V_s$ such that $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \text{ after } (0, a) \text{ after } (\sigma', t') \in v$ are in W_0 , for any $t' \in \mathbb{R}_{\geq 0}$. We know by construction of σ' that this holds for some t' , when an edge belonging to E_5 can be followed. The constraint item (6) required on V_s (see Definition 4.16) ensures that this is thus true for all t' .

Thus, $\sigma' \in G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \text{ after } (0, a), \text{buf}_c)$, so $G(\text{Reach}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)), \text{buf}_c) \neq \emptyset$. Thus, $0 \in T(\text{Reach}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)), \text{buf}_c)$, meaning that $\sigma'_s \in G(\text{Reach}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)), \text{buf}_c)$. Let us consider $t' = \text{time}(\sigma \cdot (\delta, a))$.

Now, following lemma A.15, since $t \geq t'$, $t - t' \geq 0$, then:

$$\begin{aligned} & \text{nobs}(\sigma'_s, t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma'_s, t - t'))) \in \\ & \quad \text{G}(\text{Reach}(\sigma_s . (t' - \text{time}(\sigma_s), a)) \text{ after } (\sigma'_s, t - t'), \\ & \quad \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . \text{buf}_c) \end{aligned}$$

Since $t \geq t' = \text{time}(\sigma . (\delta, a))$,

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) &= \text{nobs}(\sigma_s . (t' - \text{time}(\sigma_s), a) . \sigma'_s, t) \\ &= \text{nobs}(\sigma'_s, t - \text{time}(\sigma_s . (t' - \text{time}(\sigma_s), a))) \\ &= \text{nobs}(\sigma'_s, t - \text{time}(\sigma . (\delta, a))) \\ &= \text{nobs}(\sigma'_s, t - t') \end{aligned}$$

and $\text{obs}(\sigma_{t0}, t) = \sigma_s . (t' - \text{time}(\sigma_s), a) . (\text{obs}(\sigma'_s, t - t'))$. Thus,
 $\text{time}(\text{obs}(\sigma_{t0}, t)) = \text{time}(\sigma_s . (t' - \text{time}(\sigma_s), a)) + \text{time}(\text{obs}(\sigma'_s, t - t')) = t' + \text{time}(\text{obs}(\sigma'_s, t - t'))$.

This means that:

$$\begin{aligned} & \text{nobs}(\sigma'_s, t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma'_s, t - t'))) \\ &= \text{nobs}(\sigma_{t0}, t) -_t (t - t' - (\text{time}(\text{obs}(\sigma_{t0}, t) - t')) \\ &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \end{aligned}$$

Thus,

$$\begin{aligned} & \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\ & \quad \text{G}(\text{Reach}(\sigma_s . (t' - \text{time}(\sigma_s), a)) \text{ after } (\sigma'_s, t - t'), \\ & \quad \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . \text{buf}_c) \end{aligned}$$

Since

$$\begin{aligned} & \text{Reach}(\sigma_s . (t' - \text{time}(\sigma_s), a)) \text{ after } (\sigma'_s, t - t') \\ &= \text{Reach}(\sigma_s . (t' - \text{time}(\sigma_s), a)) . \sigma'_s, t - t' + \\ & \quad \text{time}(\sigma_s . (t' - \text{time}(\sigma_s), a))) \\ &= \text{Reach}(\sigma_{t0}, t - t' + \text{time}(\sigma . (\delta, a))) \\ &= \text{Reach}(\sigma_{t0}, t) \end{aligned}$$

and

$$\begin{aligned} & \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) . \sigma_d \\ &= \Pi_\Sigma(\text{obs}(\sigma_{t0}, t))^{-1} . \sigma_{t0} . \sigma_d \\ &= (\Pi_\Sigma(\text{obs}(\sigma_{t0}, t))^{-1} . \Pi_\Sigma(\sigma_{t0})) . \sigma_d \\ &= \Pi_\Sigma(\text{obs}(\sigma_s . (t' - \text{time}(\sigma_s), a) . \sigma'_s, t))^{-1} . \\ & \quad (\Pi_\Sigma(\sigma_s . (t' - \text{time}(\sigma_s), a)) . \sigma'_s . \sigma_d) \\ &= (\Pi_\Sigma(\sigma_s . (t' - \text{time}(\sigma_s), a)) . \Pi_\Sigma(\text{obs}(\sigma'_s, t - t')))^{-1} . \\ & \quad (\Pi_\Sigma(\sigma_s . (t' - \text{time}(\sigma_s), a)) . \Pi_\Sigma(\sigma'_s) . \sigma_d) \\ &= \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . (\Pi_\Sigma(\sigma_s . (t' - \text{time}(\sigma_s), a)))^{-1} . \\ & \quad (\Pi_\Sigma(\sigma_s . (t' - \text{time}(\sigma_s), a)) . \Pi_\Sigma(\sigma'_s) . \sigma_d) \\ &= \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . (\Pi_\Sigma(\sigma'_s) . \sigma_d) \\ &= \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . (\Pi_\Sigma(\sigma'_s) . (\Pi_\Sigma(\sigma'_s)^{-1} . \text{buf}_c)) \\ &= \Pi_\Sigma(\text{obs}(\sigma'_s, t - t'))^{-1} . \text{buf}_c \end{aligned}$$

it follows that:

$$\begin{aligned} \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) &\in \\ G(\text{Reach}(\sigma_{t0}, t), \Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d) \end{aligned}$$

- Otherwise, $a \in \Sigma_c$, and then, since $G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \neq \emptyset$, there exists $\sigma' \in \text{tw}(\Sigma)$ that satisfies the three following constraints:

1. $\Pi_\Sigma(\sigma') \preceq \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c$,
2. $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))$ after $\sigma' \in F \times \mathbb{R}_{\geq 0}$,
3. for any $t' \in \mathbb{R}_{\geq 0}$, if $v \in V_s$ is such that $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \in v$, then $\langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma', t'))^{-1} \cdot \text{buf}_c), 1 \rangle \in W_0$.

Thus, item 1 can be written as $\Pi_\Sigma(\sigma') \preceq \text{buf}_c \cdot a$, and from item 3 we can deduce that for any $t' \in \mathbb{R}_{\geq 0}$, if $v \in V_s$ is such that $\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \in v$, then $\langle v, \text{maxbuffer}(\Pi_\Sigma(\text{obs}(\sigma', t'))^{-1} \cdot (\text{buf}_c \cdot a)), 1 \rangle \in W_0$. This last property holds because adding a controllable event to the buffer only gives more possibilities to the enforcement mechanism (in the game graph, if $\langle v, w, p \rangle$ is winning, then $\langle v, w \cdot c, p \rangle$ is also winning). This means that $\sigma' \in G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \text{buf}_c \cdot a)$, and thus, $G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \text{buf}_c \cdot a) \neq \emptyset$.

Thus, $0 \in T(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \text{buf}_c \cdot a)$, meaning that $\sigma_s'' -_t (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s)) \in G(\text{Reach}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))), \text{buf}_c \cdot a)$. Let us consider $t' = \text{time}(\sigma \cdot (\delta, a))$, and $t'' = t' - \text{time}(\sigma_s)$.

Then, following lemma A.15,

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'' -_t t'', t - t'))) &\in \\ G(\text{Reach}(\sigma_{s0}, t') \text{ after } (\sigma_s'' -_t t'', t - t'), & \\ \Pi_\Sigma(\text{nobs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a)) & \end{aligned}$$

Now, if $\text{delay}(\sigma_s''(1)) > t - \text{time}(\sigma_s)$ (i.e. $\text{delay}((\sigma_s'' -_t t'')(1)) > t - t'$), then

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') &= \text{nobs}(\sigma_s'', t - t' + t'') -_t t'' \\ &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t t'' \end{aligned}$$

Since $\text{nobs}(\sigma_{t0}, t) = \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s))$, it follows that $\text{nobs}(\sigma_s'' -_t t'', t - t') = \text{nobs}(\sigma_{t0}, t) -_t t''$.

Moreover, $\text{obs}(\sigma_s'' -_t t'', t - t') = \epsilon$ since $\text{delay}(\sigma_s''(1)) > t - \text{time}(\sigma_s)$, thus, considering that $\text{obs}(\sigma_{t0}, t) = \sigma_s$,

$$\begin{aligned} \text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'', t - t'))) & \\ = \text{nobs}(\sigma_{t0}, t) -_t (t - t' + t'') & \\ = \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\sigma_s)) & \\ = \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) & \end{aligned}$$

On the other hand, if $\text{delay}(\sigma_s''(1)) \leq t - \text{time}(\sigma_s)$, then $\text{nobs}(\sigma_s'' -_t t'', t - t') = \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) = \text{nobs}(\sigma_{t0}, t)$, and since $\text{time}(\text{obs}(\sigma_{t0}, t)) = \text{time}(\text{obs}(\sigma_s \cdot \sigma_s'', t))$

$$\begin{aligned}
 &= \text{time}(\sigma_s \cdot (\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)), t)) \\
 &= \text{time}(\sigma_s) + \text{time}(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))
 \end{aligned}$$

it follows that

$$\begin{aligned}
 &\text{nobs}(\sigma_s'' -_t t'', t - t') -_t (t - t' - \text{time}(\text{obs}(\sigma_s'' -_t t'', t - t'))) \\
 &= \text{nobs}(\sigma_s'', t - \text{time}(\sigma_s)) -_t \\
 &\quad (t - t' - \text{time}(\text{obs}(\sigma_s'', t - t' + t'') -_t t'')) \\
 &= \text{nobs}(\sigma_{t0}, t) -_t (t - t' - (\text{time}(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s))) - t'')) \\
 &= \text{nobs}(\sigma_{t0}, t) -_t (t - t' + t'' - (\text{time}(\text{obs}(\sigma_{t0}, t)) - \text{time}(\sigma_s))) \\
 &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\sigma_s) + \text{time}(\sigma_s) - \text{time}(\text{obs}(\sigma_{t0}, t))) \\
 &= \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t)))
 \end{aligned}$$

Thus, in both cases,

$$\begin{aligned}
 &\text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\
 &\quad \text{G}(\text{Reach}(\sigma_{s0}, t') \text{ after } (\sigma_s'' -_t t'', t - t'), \\
 &\quad \Pi_\Sigma(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a))
 \end{aligned}$$

Since

$$\begin{aligned}
 &\text{Reach}(\sigma_{s0}, t') \text{ after } (\sigma_s'' -_t t'', t - t') \\
 &= \text{Reach}(\sigma_s) \text{ after } (\epsilon, t' - \text{time}(\sigma_s)) \text{ after} \\
 &\quad (\sigma_s'' -_t t'', t - t') \\
 &= \text{Reach}(\sigma_s) \text{ after } ((\sigma_s'' -_t t'') +_t t'', t - t' + t'') \\
 &= \text{Reach}(\sigma_s) \text{ after } (\sigma_s'', t - \text{time}(\sigma_s)) \\
 &= \text{Reach}(\sigma_s \cdot \sigma_s'', t) \\
 &= \text{Reach}(\sigma_{t0}, t)
 \end{aligned}$$

and

$$\begin{aligned}
 &\Pi_\Sigma(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d \\
 &= \Pi_\Sigma(\text{obs}(\sigma_{t0}, t))^{-1} \cdot \sigma_{t0} \cdot \sigma_d \\
 &= (\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s))))^{-1} \cdot \\
 &\quad (\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_s'') \cdot \sigma_d) \\
 &= \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot \\
 &\quad (\Pi_\Sigma(\sigma_s)^{-1} \cdot (\Pi_\Sigma(\sigma_s) \cdot \Pi_\Sigma(\sigma_s'') \cdot \sigma_d)) \\
 &= \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot \\
 &\quad (\Pi_\Sigma(\sigma_s'') \cdot \Pi_\Sigma(\sigma_s'')^{-1} \cdot (\text{buf}_c \cdot a)) \\
 &= \Pi_\Sigma(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\text{buf}_c \cdot a)
 \end{aligned}$$

considering that

$$\begin{aligned}
& \Pi_{\Sigma}(\text{obs}(\sigma_s'' -_t t'', t - t'))^{-1} \cdot (\text{buf}_c \cdot a) \\
&= \Pi_{\Sigma}(\text{obs}(\sigma_s'', t - t' + t'') -_t t'')^{-1} \cdot (\text{buf}_c \cdot a) \\
&= \Pi_{\Sigma}(\text{obs}(\sigma_s'', t - \text{time}(\sigma_s)))^{-1} \cdot (\text{buf}_c \cdot a)
\end{aligned}$$

we finally obtain

$$\begin{aligned}
& \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\
& G(\text{Reach}(\sigma_{t0}, t), \Pi_{\Sigma}(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d)
\end{aligned}$$

Thus, in both cases,

$$\begin{aligned}
& \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in \\
& G(\text{Reach}(\sigma_{t0}, t), \Pi_{\Sigma}(\text{nobs}(\sigma_{t0}, t)) \cdot \sigma_d).
\end{aligned}$$

In particular, this means that $\text{Reach}(\sigma_{t0}, t)$ after $\text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \in F_G$. Since

$$\begin{aligned}
& \text{Reach}(\sigma_{t0}, t) \text{ after } \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \\
&= \text{Reach}(\text{obs}(\sigma_{t0}, t)) \text{ after } (\epsilon, t - \text{time}(\text{obs}(\sigma_{t0}, t))) \text{ after} \\
& \quad \text{nobs}(\sigma_{t0}, t) -_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \\
&= \text{Reach}(\text{obs}(\sigma_{t0}, t)) \text{ after } (\text{nobs}(\sigma_{t0}, t) -_t \\
& \quad (t - \text{time}(\text{obs}(\sigma_{t0}, t)))) +_t (t - \text{time}(\text{obs}(\sigma_{t0}, t))) \\
&= \text{Reach}(\text{obs}(\sigma_{t0}, t)) \text{ after } \text{nobs}(\sigma_{t0}, t) \\
&= \text{Reach}(\text{obs}(\sigma_{t0}, t) \cdot \text{nobs}(\sigma_{t0}, t)) \\
&= \text{Reach}(\sigma_{t0})
\end{aligned}$$

this means that $\text{Reach}(\sigma_{t0}) \in F_G$, meaning that $\sigma_{t0} \models \varphi$.

Thus, if $\sigma \in \text{Pre}(\varphi, \text{time}(\sigma \cdot (\delta, a)))$, $P(\sigma) \implies P(\sigma \cdot (\delta, a), t)$.

Thus, in all cases, for any $t \in \mathbb{R}_{\geq 0}$, $P(\sigma) \implies P(\sigma \cdot (\delta, a), t)$. This means that $P(\sigma) \implies P(\sigma \cdot (\delta, a))$.

We then have shown by induction that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$. In particular, we have shown that for any $(\sigma, t) \in \text{Pre}(\varphi)$, $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \text{store}_{\varphi} \implies \sigma_s \models \varphi$. Thus there exists t' that we can consider such that $t' \geq t$, that is such that for any $t'' \geq t'$, $\langle (\sigma, t''), \sigma_s \rangle \in E_{\varphi}$.

Thus, E_{φ} is sound in $\text{Pre}(\varphi)$. □

Proposition 4.8. E_{φ} is compliant, as per Definition 4.13.

Proof. We have to prove that the three following properties hold:

1. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \langle (\sigma, t), o_1 \rangle \in E_{\varphi} \implies o_1 \preceq_{\text{d}_{\Sigma_c}} \text{obs}(\sigma, t)$
2. $\forall \sigma \in \text{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0}, \langle (\sigma, t), o_2 \rangle \in E_{\varphi} \implies o_2 =_{\Sigma_u} \text{obs}(\sigma, t)$

3. $\forall \sigma \in \text{tw}(\Sigma), \forall (\delta, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u,$
 $\langle (\sigma, \text{time}(\sigma \cdot (\delta, u))), o_3 \rangle \in E_\varphi \wedge \langle (\sigma \cdot (\delta, u), \text{time}(\sigma \cdot (\delta, u))), o_4 \rangle \in E_\varphi$
 $\implies o_3 \cdot (\text{time}(\sigma \cdot (\delta, u)) - \text{time}(o_3), u) \preceq o_4.$

We start by proving items 1 and 2.

For $\sigma \in \text{tw}(\Sigma)$, let $P(\sigma)$ be the predicate “ $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi \implies (\sigma_{s0} \preceq_{d\Sigma_c} \sigma \wedge \sigma_{s0} =_{\Sigma_u} \sigma \wedge \Pi_\Sigma(\sigma_{s0})|_{\Sigma_c} \cdot \sigma_c = \Pi_\Sigma(\sigma)|_{\Sigma_c})$ ”. Let us prove by induction that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$.

Induction basis: for $\sigma = \epsilon$, $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_\varphi$, and since $\epsilon \preceq_{d\Sigma_c} \epsilon$, $\epsilon =_{\Sigma_u} \epsilon$, and $\Pi_\Sigma(\epsilon)|_{\Sigma_c} \cdot \epsilon = \Pi_\Sigma(\epsilon)|_{\Sigma_c}$, it follows that $P(\epsilon)$ holds.

Induction step: Suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, $(\sigma \cdot (\delta, a), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_\varphi$, and $\sigma_s = \text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))$.

- If $a \in \Sigma_u$, then there exists σ'_s such that $\sigma_{t0} = \sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) \cdot \sigma'_s$, and $\Pi_\Sigma(\sigma'_s) \cdot \sigma_d = \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c$. Thus, since $a \in \Sigma_u$

$$\begin{aligned} & \Pi_\Sigma(\sigma_{t0})|_{\Sigma_c} \cdot \sigma_d \\ &= \Pi_\Sigma(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) \cdot \sigma'_s)|_{\Sigma_c} \cdot \sigma_d \\ &= \Pi_\Sigma(\sigma_s)|_{\Sigma_c} \cdot \Pi_\Sigma(\sigma'_s)|_{\Sigma_c} \cdot \sigma_d \end{aligned}$$

Now, following the induction hypothesis, $\sigma_{s0} =_{\Sigma_u} \sigma$, and since $\text{nobs}(\sigma, \text{time}(\sigma \cdot (\delta, a))) = \epsilon$, it follows that $\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a))) \in \text{tw}(\Sigma_c)$, and thus $\sigma'_s \in \text{tw}(\Sigma_c)$ too. Also following the induction hypothesis, we know that $\Pi_\Sigma(\sigma_{s0})|_{\Sigma_c} \cdot \sigma_c = \Pi_\Sigma(\sigma)|_{\Sigma_c}$. It follows that

$$\begin{aligned} & \Pi_\Sigma(\sigma_{t0})|_{\Sigma_c} \cdot \sigma_d = \Pi_\Sigma(\sigma_s)|_{\Sigma_c} \cdot \Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \\ &= \Pi_\Sigma(\text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \\ & \quad \text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))|_{\Sigma_c} \cdot \sigma_c \\ &= \Pi_\Sigma(\sigma_{s0})|_{\Sigma_c} \cdot \sigma_c \\ &= \Pi_\Sigma(\sigma)|_{\Sigma_c} \\ &= \Pi_\Sigma(\sigma \cdot (\delta, a))|_{\Sigma_c} \end{aligned}$$

Moreover, following the induction hypothesis, $\sigma_{s0} \preceq_{d\Sigma_c} \sigma$, thus in particular, $\sigma_s \preceq_{d\Sigma_c} \sigma$, meaning that if $i \in [1; |\sigma_s|]$, then $\text{time}(\sigma_s|_{\Sigma_c[..i]}) \leq \text{time}(\sigma|_{\Sigma_c[..i]})$, and since $i \leq |\sigma|_{\Sigma_c}$, that means that $\text{time}(\sigma_s|_{\Sigma_c[..i]}) \leq \text{time}((\sigma \cdot (\delta, a))|_{\Sigma_c[..i]})$. Since $\text{time}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a)) = \text{time}(\sigma \cdot (\delta, a))$, it follows that for any $i \in [|\sigma_s|+1; |\sigma_{t0}|_{\Sigma_c}]$, $\text{time}(\sigma_{t0}|_{\Sigma_c[..i]}) \geq \text{time}(\sigma \cdot (\delta, a))$ (remember that the restriction to an alphabet conserves dates, not delays). Thus, for any $i \in [1; |\sigma_{t0}|_{\Sigma_c}]$, $\text{time}(\sigma_{t0}|_{\Sigma_c[..i]}) \geq \text{time}((\sigma \cdot (\delta, a))|_{\Sigma_c[..i]})$. Since we have already shown that $\Pi_\Sigma(\sigma_{t0})|_{\Sigma_c} \cdot \sigma_d = \Pi_\Sigma(\sigma)|_{\Sigma_c}$, we know that $\Pi_\Sigma(\sigma_{t0})|_{\Sigma_c} \preceq \Pi_\Sigma(\sigma)|_{\Sigma_c}$. This means that $\sigma_{t0} \preceq_{d\Sigma_c} \sigma \cdot (\delta, a)$.

Finally, by induction hypothesis, $\sigma_{s0} =_{\Sigma_u} \sigma$, thus, since

$$\begin{aligned}
\sigma_{t0|_{\Sigma_u}} &= (\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a))|_{\Sigma_u} \\
&= \sigma_{s|_{\Sigma_u}} \cdot ((\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s), a) +_t \\
&\quad (\text{time}(\sigma_s) - \text{time}(\sigma_{s|_{\Sigma_u}}))) \\
&= \sigma_{s|_{\Sigma_u}} \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s) + \\
&\quad \text{time}(\sigma_s) - \text{time}(\sigma_{s|_{\Sigma_u}}), a) \\
&= \sigma_{s|_{\Sigma_u}} \cdot (\text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_{s|_{\Sigma_u}}), a) \\
&= \sigma|_{\Sigma_u} \cdot ((\delta, a) +_t (\text{time}(\sigma) - \text{time}(\sigma|_{\Sigma_u}))) \\
&= (\sigma \cdot (\delta, a))|_{\Sigma_u}
\end{aligned}$$

Thus, $P(\sigma \cdot (\delta, a))$ holds.

- Otherwise, $a \in \Sigma_c$, and then, there exists $\sigma_s'' \in \text{tw}(\Sigma)$ such that $\sigma_{t0} = \sigma_s \cdot \sigma_s''$ and $\Pi_{\Sigma}(\sigma_s'') \cdot \sigma_d = \Pi_{\Sigma}(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \cdot a$.

Thus,

$$\begin{aligned}
\Pi_{\Sigma}(\sigma_{t0})|_{\Sigma_c} \cdot \sigma_d &= \Pi_{\Sigma}(\sigma_s)|_{\Sigma_c} \cdot \Pi_{\Sigma}(\sigma_s'') \cdot \sigma_d \\
&= \Pi_{\Sigma}(\sigma_s)|_{\Sigma_c} \cdot \Pi_{\Sigma}(\text{nobs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, a)))) \cdot \sigma_c \cdot a \\
&= \Pi_{\Sigma}(\sigma_{s0})|_{\Sigma_c} \cdot \sigma_c \cdot a \\
&= \Pi_{\Sigma}(\sigma)|_{\Sigma_c} \cdot a \\
&= \Pi_{\Sigma}(\sigma \cdot (\delta, a))|_{\Sigma_c}
\end{aligned}$$

As in the case where $a \in \Sigma_u$, for any $i \in [1; |\sigma_{s|_{\Sigma_c}}|]$, $\text{time}(\sigma_{t0|_{\Sigma_c[..i]}}) \leq \text{time}(\sigma_{s|_{\Sigma_c[..i]}})$. Moreover, by construction, $\text{delay}(\sigma_s''(1)) \geq \text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s)$, thus for $i \in [|\sigma_{s|_{\Sigma_c}}| + 1; \text{time}(\sigma_{t0|_{\Sigma_c}})]$, if $i' = i - |\sigma_{s|_{\Sigma_c}}|$, then $\text{time}(\sigma_{t0|_{\Sigma_c[..i]}}) = \text{time}(\sigma_s) + \text{time}(\sigma_s''[..i']) \geq \text{time}(\sigma_s) + \text{time}(\sigma \cdot (\delta, a)) - \text{time}(\sigma_s) = \text{time}(\sigma \cdot (\delta, a))$. Since $\text{time}(\sigma \cdot (\delta, a)) \leq \text{time}((\sigma \cdot (\delta, a))|_{\Sigma_c[..i]})$, and $\Pi_{\Sigma}(\sigma_{t0})|_{\Sigma_c} \preceq \Pi_{\Sigma}(\sigma \cdot (\delta, a))|_{\Sigma_c}$, this means that $\sigma_{t0} \preceq_{d\Sigma_c} \sigma \cdot (\delta, a)$.

Finally, $\sigma_{t0|_{\Sigma_u}} = \sigma_{s|_{\Sigma_u}} = \sigma|_{\Sigma_u} = (\sigma \cdot (\delta, a))|_{\Sigma_u}$.

Thus $P(\sigma \cdot (\delta, a))$ holds.

In both cases, $P(\sigma) \implies P(\sigma \cdot (\delta, a))$.

Thus, we have shown by induction that for all $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Consequently, for any $\sigma \in \text{tw}(\Sigma)$, if $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_{\varphi}$, then $\sigma_{s0} \preceq_{d\Sigma_c} \sigma$ and $\sigma_{s0} =_{\Sigma_u} \sigma$. Thus, for any $t \in \mathbb{R}_{\geq 0}$, if $\langle (\sigma, t), o \rangle \in E_{\varphi}$, then $o = \text{obs}(\sigma_{s0}, t) \preceq_{d\Sigma_c} \text{obs}(\sigma, t)$, and $o = \text{obs}(\sigma_{s0}, t) =_{\Sigma_u} \text{obs}(\sigma, t)$.

Thus, items 1 and 2 hold.

Now, let us prove item 3. Let us consider $\sigma \in \text{tw}(\Sigma)$, $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_{\varphi}$, $(\delta, u) \in \mathbb{R}_{\geq 0} \times \Sigma_u$, $(\sigma \cdot (\delta, u), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_{\varphi}$, and $\sigma_s = \text{obs}(\sigma_{s0}, \text{time}(\sigma \cdot (\delta, u)))$. Then, $\langle (\sigma, \text{time}(\sigma \cdot (\delta, u))), \sigma_s \rangle \in E_{\varphi}$, and following the definition

of store_φ (Definition 4.19), $\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, u)) - \text{time}(\sigma_s), u) \preceq \sigma_{t0}$. Thus, if $\langle (\sigma \cdot (\delta, u), \text{time}(\sigma \cdot (\delta, u))), o_4 \rangle \in E_\varphi$, then $o_4 = \text{obs}(\sigma_{t0}, \text{time}(\sigma \cdot (\delta, u)))$. Since $\text{time}(\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, u)) - \text{time}(\sigma_s), u)) = \text{time}(\sigma \cdot (\delta, u))$, it follows that $\sigma_s \cdot (\text{time}(\sigma \cdot (\delta, u)) - \text{time}(\sigma_s), u) \preceq o_4$.

We have then shown that E_φ is compliant with respect to Σ_u and Σ_c . \square

Proposition 4.9. E_φ is optimal in $\text{Pre}(\varphi)$ as per Definition 4.14.

Proof. Let us consider $\sigma \in \text{tw}(\Sigma)$, $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$ such that $(\sigma, \text{time}(\sigma \cdot (\delta, a))) \in \text{Pre}(\varphi)$, E an enforcement function that is compliant with respect to Σ_u and Σ_c , $\langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o \rangle \in E \cap E_\varphi$, $(\sigma \cdot (\delta, a), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_\varphi$, and $\langle (\sigma \cdot (\delta, a), \infty), o'_1 \rangle \in E$. Let us suppose that $\sigma_{s0} \prec_d o'_1$. We then have to prove that there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that if $\langle (\sigma \cdot (\delta, a) \cdot \sigma_u, \infty), o_u \rangle \in E$, then $o_u \not\models \varphi$.

Let us consider $\sigma_s = \text{obs}(o, \text{time}(\sigma \cdot (\delta, a)))$. Then, since E_φ and E are compliant, there exists $\sigma'_s \in \text{tw}(\Sigma)$ such that $\sigma_{t0} = o \cdot \sigma'_s$ and $\sigma_s^E \in \text{tw}(\Sigma)$ such that $o'_1 = \sigma_s \cdot \sigma_s^E$. Now, since $E_\varphi(\sigma \cdot (\delta, a)) \prec_d E(\sigma \cdot (\delta, a))$, this means that $\sigma'_s \prec_d \sigma_s^E$.

- If $a \in \Sigma_u$, since $(\sigma, \text{time}(\sigma \cdot (\delta, a))) \in \text{Pre}(\varphi)$, we know that (see proof of Proposition 4.7) $\sigma'_s \in G(\text{Reach}(\sigma_s), \Pi_\Sigma(\sigma_s)_{|\Sigma_c}^{-1} \cdot \Pi_\Sigma(\sigma)_{|\Sigma_c})$. Now, since $\sigma'_s \prec_d \sigma_s^E$, and since σ'_s is the maximal word for \prec_d that is in G , this means that $\sigma_s^E \notin G(\text{Reach}(\sigma_s), \Pi_\Sigma(\sigma_s)_{|\Sigma_c}^{-1} \cdot \Pi_\Sigma(\sigma)_{|\Sigma_c})$. This means that one of the following does not hold :

1. $\Pi_\Sigma(\sigma_s^E) \preceq \Pi_\Sigma(\sigma_s)_{|\Sigma_c}^{-1} \cdot \Pi_\Sigma(\sigma)_{|\Sigma_c}$, but if this did not hold, then E would not be compliant.
2. $\text{Reach}(\sigma_s)$ after $\sigma_s^E \notin F_G$. If this does not hold, then $\text{Reach}(\sigma_s \cdot \sigma_s^E) \notin F_G$, meaning that $o'_1 \not\models \varphi$.
3. $\forall t \in \mathbb{R}_{\geq 0}, \forall v \in V_s, \text{Reach}(\sigma_s \cdot \sigma_s^E, t) \in v \implies \langle v, \text{maxbuffer}(\Pi_\Sigma(\sigma_s \cdot \text{obs}(\sigma_s^E, t))^{-1} \cdot \Pi_\Sigma(\sigma)_{|\Sigma_c}), 1 \rangle \in W_0$. If this does not hold, then there exists $t \in \mathbb{R}_{\geq 0}$ and $v \in V_s$ such that $\text{Reach}(\sigma_s \cdot \sigma_s^E, t) \in v$ and $\langle v, \text{maxbuffer}(\Pi_\Sigma(\sigma \cdot \text{obs}(\sigma_s^E, t))^{-1} \cdot \Pi_\Sigma(\sigma)_{|\Sigma_c}), 1 \rangle \notin W_0$. Then, there exists a winning strategy for player 1 from this node. This means that we can construct a word by following the winning strategy of player 1, like it is done in the proof of Proposition 4.7: depending on the edge followed in the game graph, player 1 can add an uncontrollable event to the input word (the delays are given by the edges corresponding to letting time elapse) that allows to stay in a node not belonging to W_0 . This can be done until the strategy of player 0 goes back to the previous node, making a loop if it has no time successor. This must ultimately happen since adding controllable events to the input only gives player 0 more possibilities,

thus player 1 can choose only edges corresponding to adding uncontrollable events or letting time elapse. By privileging the elapse of time, it can ensure that the word will be finite. Thus, player 1 can build a word $\sigma_u \in \text{tw}(\Sigma_u)$ such that if $\langle (\sigma \cdot (\delta, a) \cdot \sigma_u, \infty), o_u \rangle \in E$, then $o_u \not\models \varphi$.

In any possible case, there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that if $\langle (\sigma \cdot (\delta, a) \cdot \sigma_u, \infty), o_u \rangle \in E$, then $o_u \not\models \varphi$ (in the second case, $\sigma_u = \epsilon$).

- Otherwise, $a \in \Sigma_c$, and we can prove as in the previous case that there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that if $\langle (\sigma \cdot (\delta, a) \cdot \sigma_u, \infty), o_u \rangle \in E_\varphi$, then $o_u \not\models \varphi$. All that is needed is to adapt the parameters of G : $\sigma'_s \in G(\text{Reach}(\sigma_s, \text{time}(\sigma \cdot (\delta, a))), \Pi_\Sigma(\sigma_s)_{|\Sigma_c}^{-1} \cdot \Pi_\Sigma(\sigma \cdot (\delta, a))_{|\Sigma_c})$, but the arguments are the same.

Thus, if E is compliant, and $\sigma \in \text{tw}(\Sigma)$ and $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$ are such that $(\sigma, \text{time}(\sigma \cdot (\delta, a))) \in \text{Pre}(\varphi)$, $\langle (\sigma, \text{time}(\sigma \cdot (\delta, a))), o \rangle \in E \cap E_\varphi$, $\langle (\sigma \cdot (\delta, a), \infty), o_1 \rangle \in E_\varphi$, $\langle (\sigma \cdot (\delta, a), \infty), o'_1 \rangle \in E$, and $o_1 \prec_d o'_1$, then there exists $\sigma_u \in \text{tw}(\Sigma_u)$ such that if $\langle (\sigma \cdot (\delta, a) \cdot \sigma_u, o_u) \rangle \in E$, then $o_u \not\models \varphi$.

This means that E_φ is optimal in $\text{Pre}(\varphi)$. \square

Proposition 4.10. *The output o of \mathcal{E} as per Definition 4.21 for input σ at date t is such that $((\sigma, t), o) \in E_\varphi$.*

Proof. In this proof, we use some notation from Section 4.3.3:

- $C^\mathcal{E} = \text{tw}(\Sigma) \times \Sigma_c^* \times Q \times \mathbb{R}_{\geq 0}$ is the set of configurations.
- $c_0^\mathcal{E} = \langle \epsilon, \epsilon, q_0, 0 \rangle \in C^\mathcal{E}$ 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, composed of an optional input, an operation and an optional output.

The set of operations is $\{\text{dump}(\cdot), \text{pass-uncont}(\cdot), \text{store-cont}(\cdot), \text{delay}(\cdot)\}$.

For a sequence of rules $w \in (\Gamma^\mathcal{E})^*$, we note the concatenation of all the inputs of w , $\text{input}(w) = \Pi_1(w(1)) \cdot \Pi_1(w(2)) \dots \Pi_1(w(|w|))$, and $\text{output}(w) = \Pi_3(w(1)) \cdot \Pi_3(w(2)) \dots \Pi_3(w(|w|))$ the concatenation of all the outputs of w . Since all configurations are not reachable from $c_0^\mathcal{E}$, for $w \in (\Gamma^\mathcal{E})^*$, we note $\text{Reach}^\mathcal{E}(w) = c$ if $c_0^\mathcal{E} \xrightarrow{w} c$ for some configuration $c \in C^\mathcal{E}$, or $\text{Reach}^\mathcal{E}(w) = \perp$ if such a configuration does not exist. For a word $\sigma \in \text{tw}(\Sigma)$, and a date $t \in \mathbb{R}_{\geq 0}$, we note $\text{Rules}(\sigma, t) = \max_{\prec}(\{w \in (\Gamma^\mathcal{E})^* \mid \text{input}(w) = \text{obs}(\sigma, t) \wedge \text{Reach}^\mathcal{E}(w) \neq \perp \wedge \Pi_4(\text{Reach}^\mathcal{E}(w)) = t - \text{time}(\text{output}(w))\})$. We also note $\text{Reach}^\mathcal{E}(\sigma, t) = \text{Reach}^\mathcal{E}(\text{Rules}(\sigma, t))$. $\text{Rules}(\sigma, t)$ represent the sequence that the EM applies with input word σ until date t . Since rule $\text{delay}()$ can be applied an infinite number of times by slicing time, we only consider words in

$(\Gamma^{\mathcal{E}})^*$ that are minimal in the number of rules $\text{delay}()$, *i.e.* the word obtained by merging two consecutive rules $\text{delay}()$ into one with the sum of delays of the two rules, until stabilisation. This allows to define $\text{Rules}(\sigma, t)$ correctly, without “cheating” by slicing time to increase the length of the word. Note that the words obtained by merging or adding $\text{delay}()$ rules this way reach exactly the same configurations in the end. We will also allow ourselves to extend the use of output to timed words, such that $\text{output}(\sigma, t) = \text{output}(\text{Rules}(\sigma, t))$.

We have to show that for any $\sigma \in \text{tw}(\Sigma)$, and $t \in \mathbb{R}_{\geq 0}$, if $\langle (\sigma, t), o \rangle \in E_{\varphi}$, then $o = \text{output}(\sigma, t)$.

Now, for $\sigma \in \text{tw}(\Sigma)$ and $t \in \mathbb{R}_{\geq 0}$, let $P(\sigma, t)$ be the predicate “ $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_{\varphi} \implies (\text{output}(\sigma, t) = \text{obs}(\sigma_{s0}, t) \wedge \text{Reach}^{\mathcal{E}}(\sigma, t) = \langle \text{nobs}(\sigma_{s0}, t), \sigma_c, \text{Reach}(\sigma_{s0}, t), t - \text{time}(\text{obs}(\sigma_{s0}, t)) \rangle)$ ”, and $P(\sigma)$ be the predicate “ $\forall t \in \mathbb{R}_{\geq 0}, P(\sigma, t)$ ”. Let us then show by induction that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$.

Induction basis: if $\sigma = \epsilon$, then let us consider $t \in \mathbb{R}_{\geq 0}$. Then, $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \text{store}_{\varphi}$. On the other hand, the only rule that can be applied is $\text{delay}(t)$, thus $\text{Reach}^{\mathcal{E}}(\epsilon, t) = \langle \epsilon, \epsilon, q_0 \text{ after } (\epsilon, t), t \rangle$.

Thus, $\text{output}(\epsilon, t) = \text{obs}(\epsilon, t)$, and $\text{Reach}^{\mathcal{E}}(\epsilon, t) = \langle \text{nobs}(\epsilon, t), \epsilon, \text{Reach}(\epsilon, t), t - \text{time}(\epsilon, t) \rangle$. Thus, $P(\epsilon, t)$ holds. Thus, for any $t \in \mathbb{R}_{\geq 0}$, $P(\epsilon, t)$ holds, meaning that $P(\epsilon)$ holds.

Induction step: let us suppose that for $\sigma \in \text{tw}(\Sigma)$, $P(\sigma)$ holds. Let us consider $(\delta, a) \in \mathbb{R}_{\geq 0} \times \Sigma$, $t \in \mathbb{R}_{\geq 0}$, $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_{\varphi}$, $(\sigma . (\delta, a), \langle \sigma_{t0}, \sigma_d \rangle) \in \text{store}_{\varphi}$, $\sigma_s = \text{obs}(\sigma_{s0}, \text{time}(\sigma . (\delta, a)))$, and $c = \text{Reach}^{\mathcal{E}}(\sigma, \text{time}(\sigma . (\delta, a)))$. Then, by induction hypothesis, $c = \langle \text{nobs}(\sigma_{s0}, \text{time}(\sigma . (\delta, a))), \sigma_c, \text{Reach}(\sigma_{s0}, \text{time}(\sigma . (\delta, a))), \text{time}(\sigma . (\delta, a)) - \text{time}(\sigma_s) \rangle$.

If $t < \text{time}(\sigma . (\delta, a))$, then $\text{Reach}^{\mathcal{E}}(\sigma . (\delta, a), t) = \text{Reach}^{\mathcal{E}}(\sigma, t)$, and $\text{obs}(\sigma_{t0}, t) = \text{obs}(\sigma_{s0}, t)$, meaning that $P(\sigma . (\delta, a), t)$ holds.

Then, let us consider that $t \geq \text{time}(\sigma . (\delta, a))$.

- If $a \in \Sigma_u$, then rule $\text{pass-uncont}(a)$ can be applied, meaning that c after $(a / \text{pass-uncont}(a) / a) = \langle \sigma'_b, \sigma'_c, q, 0 \rangle$, with $q = \text{Reach}(\sigma_{s0}, \text{time}(\sigma . (\delta, a)))$ after $(0, a) = \text{Reach}(\sigma_s . (\text{time}(\sigma . (\delta, a)) - \text{time}(\sigma_s), a))$, $\sigma'_b = \kappa_{\varphi}(q, \Pi_{\Sigma}(\text{nobs}(\sigma_{s0}, \text{time}(\sigma . (\delta, a)))) . \sigma_c)$, and $\sigma'_c = \Pi_{\Sigma}(\sigma'_b)^{-1} . (\Pi_{\Sigma}(\sigma_b) . \sigma_c)$. Thus, σ'_b is such that $\sigma_{t0} = \sigma_s . \sigma'_b$, thus c after $(a / \text{pass-uncont}(a) / a) = \langle \sigma_s^{-1} . \sigma_{t0}, \sigma_d, \text{Reach}(\sigma_s . (\text{time}(\sigma . (\delta, a)) - \text{time}(\sigma_s, a))), 0 \rangle$. Then, rules $\text{delay}()$ and $\text{dump}()$ can be applied, until date t is reached, leading to the configuration $\langle \text{nobs}(\sigma_{t0}, t), \sigma_d, \text{Reach}(\sigma_{t0}, t), t - \text{time}(\text{obs}(\sigma_{t0}, t)) \rangle$.

Moreover, considering the transitions taken,

$$\begin{aligned}
\text{output}(\sigma . (\delta, a), t) &= \text{output}(\sigma, \text{time}(\sigma . (\delta, a))) . (\text{time}(\sigma . (\delta, a)) - \\
&\quad \text{time}(\sigma_s), a) . \text{obs}(\sigma'_b, t - \text{time}(\sigma . (\delta, a))) \\
&= \sigma_s . (\text{time}(\sigma . (\delta, a)) - \text{time}(\sigma_s), a) . \\
&\quad \text{obs}(\sigma'_b, t - \text{time}(\sigma . (\delta, a))) \\
&= \text{obs}(\sigma_{t0}, t)
\end{aligned}$$

Thus, $P(\sigma . (\delta, a), t)$ holds.

- Otherwise, $a \in \Sigma_c$, and then rule $\text{store-cont}(a)$ can be applied from configuration c , leading to c after $(a / \text{store-cont}(a) / \epsilon) = \langle \sigma'_b, \sigma'_c, \text{Reach}(\sigma_{s0}, \text{time}(\sigma . (\delta, a))), t - \text{time}(\sigma_s) \rangle$, with $\sigma'_b = \kappa_\varphi(\text{Reach}(\sigma_{s0}, \text{time}(\sigma . (\delta, a))))$, $\Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma . (\delta, a)))) . \sigma_c . a +_t (t - \text{time}(\sigma_s))$ and $\sigma'_c = \Pi_\Sigma(\sigma'_b)^{-1} . (\Pi_\Sigma(\text{nobs}(\sigma_{s0}, \text{time}(\sigma . (\delta, a)))) . \sigma_c . a$. Thus, σ'_b is such that $\sigma_{t0} = \sigma_s . \sigma'_b$, and $\sigma'_c = \sigma_d$. Then, rules $\text{delay}()$ and $\text{dump}()$ can be applied until date t is reached, leading to $\text{Reach}^\mathcal{E}(\sigma . (\delta, a), t) = \langle \text{nobs}(\sigma_{t0}, t), \sigma_d, \text{Reach}(\sigma_{t0}, t), t - \text{time}(\text{obs}(\sigma_{t0}, t)) \rangle$.

Moreover, considering the transitions taken,

$$\begin{aligned}
\text{output}(\sigma . (\delta, a), t) &= \text{output}(\sigma, \text{time}(\sigma . (\delta, a))) . \text{obs}(\sigma'_b, t - \text{time}(\sigma_s)) \\
&= \sigma_s . \text{obs}(\sigma'_b, t - \text{time}(\sigma_s)) \\
&= \text{obs}(\sigma_s . \sigma'_b, t) \\
&= \text{obs}(\sigma_{t0}, t)
\end{aligned}$$

Thus, $P(\sigma . (\delta, a), t)$ holds.

Thus, in both cases, $P(\sigma . (\delta, a), t)$ holds.

This means that for any $t \in \mathbb{R}_{\geq 0}$, $P(\sigma . (\delta, a), t)$ holds.

Thus $P(\sigma) \implies P(\sigma . (\delta, a))$.

We have then shown by induction that $P(\sigma)$ holds for any $\sigma \in \text{tw}(\Sigma)$. In particular, this means that for any $\sigma \in \text{tw}(\Sigma)$, if $(\sigma, \langle \sigma_{s0}, \sigma_c \rangle) \in \text{store}_\varphi$, then for any $t \in \mathbb{R}_{\geq 0}$, $\text{obs}(\sigma_{s0}, t) = \text{output}(\sigma, t)$. Thus, if $\langle (\sigma, t), o \rangle \in E_\varphi$, then $o = \text{obs}(\sigma_{s0}, t) = \text{output}(\sigma, t)$. \square

Bibliography

- ALCALDE, Baptiste, CAVALLI, Ana, CHEN, Dongluo, KHUU, Davy and LEE, David, 2004. Network protocol system passive testing for fault management: A backward checking approach. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 150–166. Springer.
- ALUR, Rajeev, COURCOUBETIS, Costas, HALBWACHS, Nicolas, DILL, David and WONG-TOI, Howard, 1992. Minimization of timed transition systems. In *CONCUR'92*, pages 340–354. Springer.
- ALUR, Rajeev and DILL, David, 1992. The theory of timed automata. In DE BAKKER, J.W., HUIZING, C., DE ROEVER, W.P. and ROZENBERG, G., editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer Berlin Heidelberg. ISBN 978-3-540-55564-3. doi:10.1007/BFb0031987.
URL <http://dx.doi.org/10.1007/BFb0031987>
- ALUR, Rajeev, FIX, Limor and HENZINGER, Thomas A, 1999. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273.
- BASIN, David, JUGÉ, Vincent, KLAEDTKE, Felix and ZĂLINESCU, Eugen, 2013. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1):3:1–3:26. doi:10.1145/2487222.2487225.
URL <http://doi.acm.org/10.1145/2487222.2487225>
- BAUER, Andreas and FALCONE, Ylies, 2012. Decentralised ltl monitoring. *FM 2012: Formal Methods*, pages 85–100.
- BAUER, Andreas, LEUCKER, Martin and SCHALLHART, Christian, 2006. Monitoring of real-time properties. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer.
- BAUER, Andreas, LEUCKER, Martin and SCHALLHART, Christian, 2007. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138. Springer.

- BAUER, Andreas, LEUCKER, Martin and SCHALLHART, Christian, 2011. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14.
- BAUER, Lujo, LIGATTI, Jay and WALKER, David, 2009. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(3):9.
- BENGTSSON, Johan and YI, Wang, 2004. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science*, 3098:87–124.
- BLOEM, Roderick, KÖNIGHOFFER, Bettina, KÖNIGHOFFER, Robert and WANG, Chao, 2015. Shield synthesis: Runtime enforcement for reactive systems. *CoRR*, abs/1501.02573.
URL <http://arxiv.org/abs/1501.02573>
- CAVALLI, Ana, GERVY, Caroline and PROKOPENKO, Svetlana, 2003. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837–852.
- CHANG, Edward, MANNA, Zohar and PNUELI, Amir, 1992. Characterization of temporal property classes. *Automata, languages and programming*, pages 474–486.
- CHEN, Feng and ROSU, Grigore, 2005. Java-mop: A monitoring oriented programming environment for java. In *TACAS*, volume 3440, pages 546–550. Springer.
- CUPPENS, Frederic, CUPPENS-BOULAHIA, Nora and RAMARD, Tony, 2006. Availability enforcement by obligations and aspects identification. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 10–pp. IEEE.
- DILL, David L, 1989. Timing assumptions and verification of finite-state concurrent systems. In *International Conference on Computer Aided Verification*, pages 197–212. Springer.
- DOLZHENKO, Egor, LIGATTI, Jay and REDDY, Srikar, 2015. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60.
- EL-HOKAYEM, Antoine and FALCONE, Ylies, 2017a. Monitoring decentralized specifications. In *26th International Symposium on Software Testing and Analysis, ISSTA*.
- EL-HOKAYEM, Antoine and FALCONE, Yliès, 2017b. Themis: A tool for decentralized monitoring algorithms.

- FALCONE, Yliès, FERNANDEZ, Jean-Claude and MOUNIER, Laurent, 2012. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382.
- FALCONE, Ylies, HAVELUND, Klaus and REGER, Giles, 2013. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34:141–175.
- FALCONE, Yliès, MOUNIER, Laurent, FERNANDEZ, Jean-Claude and RICHIER, Jean-Luc, 2011a. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262. doi:10.1007/s10703-011-0114-4.
- FALCONE, Yliès, MOUNIER, Laurent, FERNANDEZ, Jean-Claude and RICHIER, Jean-Luc, 2011b. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262. doi:10.1007/s10703-011-0114-4.
URL <http://dx.doi.org/10.1007/s10703-011-0114-4>
- FONG, Philip WL, 2004. Access control by tracking shallow execution history. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 43–55. IEEE.
- GIRAULT, Johan, LOISEAU, Jean Jacques and ROUX, Olivier H, 2013. Synthèse en ligne de superviseur compositionnel pour flotte de robots mobiles. *European Journal of Automation, MSR*, 13:1–3.
- HALLÉ, Sylvain, KHOURY, Raphaël, EL-HOKAYEM, Antoine and FALCONE, Yliès, 2016. Decentralized enforcement of artifact lifecycles. In *Enterprise Distributed Object Computing Conference (EDOC), 2016 IEEE 20th International*, pages 1–10. IEEE.
- HAMLEN, Kevin W, MORRISETT, Greg and SCHNEIDER, Fred B, 2006. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205.
- LARSEN, Kim G, PETTERSSON, Paul and YI, Wang, 1997. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152.
- LEROY, Xavier, 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM.
- LIGATTI, Jay, BAUER, Lujo and WALKER, David, 2005. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1):2–16.

- LIGATTI, Jay, BAUER, Lujo and WALKER, David, 2009. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41. doi: 10.1145/1455526.1455532.
URL <http://doi.acm.org/10.1145/1455526.1455532>
- MANNA, Zohar and PNUELI, Amir, 1990. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410. ACM.
- MARTINELL, Fabio and MATTEUCCI, Ilaria, 2007. Through modeling to synthesis of security automata. *Electronic Notes in Theoretical Computer Science*, 179:31–46.
- MCMILLAN, Kenneth L, 1993. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer.
- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry and MARCHAND, Hervé, 2015a. Tipex: a tool chain for timed property enforcement during execution. In *Runtime Verification*, pages 306–320. Springer.
- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry and MARCHAND, Hervé, 2015b. TiPEX: A Tool Chain for Timed Property Enforcement During eXecution. In BARTOCCI, Ezio and MAJUMDAR, Rupak, editors, *RV’2015, 6th International Conference on Runtime Verification*, volume 9333 of *Lecture Notes in Computer Science*, page 12. Springer, Vienne, Austria. doi:10.1007/978-3-319-23820-3_22.
- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry, MARCHAND, Hervé, ROLLET, Antoine and NGUENA-TIMO, Omer, 2014a. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422. doi:10.1007/s10703-014-0215-y.
- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry, MARCHAND, Hervé, ROLLET, Antoine and NGUENA-TIMO, Omer Landry, 2012. Runtime enforcement of timed properties. In QADEER, Shaz and TASIRAN, Serdar, editors, *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers*, volume 7687 of *Lecture Notes in Computer Science*, pages 229–244. Springer. ISBN 978-3-642-35631-5. doi:10.1007/978-3-642-35632-2_23.
- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry, MARCHAND, Hervé, ROLLET, Antoine and NGUENA-TIMO, Omer Landry, 2014b. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422.

- PINISSETTY, Srinivas, FALCONE, Yliès, JÉRON, Thierry, MARCHAND, Hervé, ROLLET, Antoine and NGUENA TIMO, OmerLandry, 2013. Runtime enforcement of timed properties. In QADEER, Shaz and TASIRAN, Serdar, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 229–244. Springer Berlin Heidelberg. ISBN 978-3-642-35631-5. doi:10.1007/978-3-642-35632-2_23.
URL http://dx.doi.org/10.1007/978-3-642-35632-2_23
- RAMADGE, P. J. G. and WONHAM, W. M., 1989. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98. doi:10.1109/5.21072.
- RENARD, Matthieu, FALCONE, Yliès, ROLLET, Antoine, JÉRON, Thierry and MARCHAND, Hervé, 2017a. Optimal enforcement of (timed) properties with uncontrollable events. *Mathematical Structures in Computer Science*, page 1–46. doi:10.1017/S0960129517000123.
- RENARD, Matthieu, FALCONE, Yliès, ROLLET, Antoine, PINISSETTY, Srinivas, JÉRON, Thierry and MARCHAND, Hervé, 2015. Enforcement of (timed) properties with uncontrollable events. In LEUCKER, Martin, RUEDA, Camilo and VALENCIA, Frank D., editors, *Theoretical Aspects of Computing - ICTAC 2015*, volume 9399 of *Lecture Notes in Computer Science*, pages 542–560. Springer International Publishing. ISBN 978-3-319-25149-3. doi:10.1007/978-3-319-25150-9_31.
- RENARD, Matthieu, ROLLET, Antoine and FALCONE, Yliès, 2017b. Grep: Games for the runtime enforcement of properties. In *Testing Software and Systems: 29th IFIP WG 6.1 International Conference, ICTSS 2017, St. Petersburg, Russia, October 9-11, 2017, Proceedings*, pages 259–275. Springer International Publishing, Cham. ISBN 978-3-319-67549-7. doi:10.1007/978-3-319-67549-7_16.
URL https://doi.org/10.1007/978-3-319-67549-7_16
- RENARD, Matthieu, ROLLET, Antoine and FALCONE, Yliès, 2017c. Runtime enforcement using Büchi games. In *Proceedings of Model Checking Software - 24th International Symposium, SPIN 2017, Co-located with ISSSTA 2017, Santa Barbara, USA*, pages 70–79. ACM Press.
- RINARD, Martin, 2003. Acceptability-oriented computing. *Acm sigplan notices*, 38(12):57–75.
- SCHNEIDER, Fred B., 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50. doi:10.1145/353323.353382.
URL <http://doi.acm.org/10.1145/353323.353382>
- UDBM, 2011. Uppaal DBM Library. <http://people.cs.aau.dk/~adavid/UDBM/>. Accessed: 2017-04-27.

- VISWANATHAN, Mahesh, 2000. Foundations for the run-time analysis of software systems.
- WU, Meng, ZENG, Haibo and WANG, Chao, 2016. Synthesizing runtime enforcer of safety properties under burst error. In *NASA Formal Methods Symposium*, pages 65–81. Springer.