Runtime Enforcement Using Büchi Games

Matthieu Renard
LaBRI
Bordeaux INP
University of Bordeaux
Bordeaux, France
matthieu.renard@labri.fr

Antoine Rollet
LaBRI
Bordeaux INP
University of Bordeaux
Bordeaux, France
antoine.rollet@labri.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria
Laboratoire d'Informatique de Grenoble
F-38000, Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr

ABSTRACT

We leverage Büchi games for the runtime enforcement of regular properties with uncontrollable events. Runtime enforcement consists in modifying the execution of a running system to have it satisfy a given regular property, modelled by an automaton. We revisit runtime enforcement with uncontrollable events and propose a framework where we model the runtime enforcement problem as a Büchi game and synthesise sound, compliant, and optimal enforcement mechanisms as strategies. We present algorithms and a tool implementing enforcement mechanisms. We reduce the complexity of the computations performed by enforcement mechanisms at runtime by pre-computing the decisions of enforcement mechanisms ahead of time.

ACM Reference format:
Matthieu Renard, Antoine Rollet, and Yliès Falcone. 2016. Runtime Enforcement Using Büchi Games. In *Proceedings of 24th International Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 12–14, 2017 (SPIN 2017),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Runtime Verification (RV) consists in checking if the execution of a running system satisfies some given specification. Unlike static verification, RV studies a real execution of a system, possibly after deployment. This paper deals with *runtime enforcement*, an extension of runtime verification where executions are corrected when they violate the property [1, 10, 11, 15]; see [7] for a tutorial on runtime enforcement. The considered properties are regular properties, that are represented by deterministic and complete automata. All such properties are monitorable since they are defined with a semantics over finite words [8]. An enforcement mechanism modifies an execution: it takes an execution as input and outputs a possibly-different execution. Enforcement mechanisms may operate *online*, meaning that they modify the executions of a system while it is running, or *offline*, by reading a log of system events. While working online, enforcement mechanisms can add a time overhead due to their need to compute a correct output. We distinguish two categories of events: *controllable* events that can be modified by an enforcement mechanism, and *uncontrollable* events that can only be observed by the enforcement mechanism. Enforcement mechanisms should be *sound* and *compliant*, meaning that the output should satisfy the specification when it is possible, and that the output should be as close to the input as possible, respectively. The general scheme is given in Fig. 1.

Motivations. In this paper, we improve the modelling of the enforcement mechanisms proposed in [13, 14], as well as the computation of their output. Such mechanisms should impact the system as little as possible, thus reducing the time spent by enforcement mechanisms to compute their behaviour allows us to use them in a more realistic way. For example, in interactive systems, where the system interacts with a human user, if an event takes too long to be output, the user may think that the system failed. It could also be useful for embedded systems, where computing power may be reduced. Computing the behaviour of the enforcement mechanism ahead of the execution and storing it ensures that the computation does not depend on the size of the automaton, thus allowing the time spent on online computations by enforcement mechanisms to be reduced and more predictable. Indeed, not exploring the whole execution tree for all possible outputs at runtime, as a naive approach would do, allows us to have computation times that vary less (with a naive approach, the computation time can become very important with an increasing number of stored controllable events).

Challenges. Storing the behaviour of enforcement mechanisms to improve their online computation time induces some changes compared to previous work (as [13, 14]). The main difficulty resides in the fact that the number of states of the enforcement mechanism is infinite. Indeed, the mechanism has the possibility to store controllable events that it may choose to release or not. The number of events that can be stored at the same time is not bounded, thus the number of states of the enforcement mechanism is infinite. Therefore, computing and storing its behaviour for all possible input traces entails defining appropriate abstractions.

Contributions. A first approach of enforcement with uncontrollable events has been presented in [14] providing sound and compliant enforcement mechanisms. An optimal version of this approach

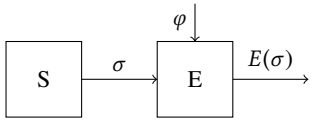

Figure 1: Schematic description of an enforcement mechanism E, modifying the execution σ of the system S to $E(\sigma)$, so that it satisfies the property φ.

SPIN 2017, Santa Barbara, CA, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

may be found in [13]. In this paper, we propose to extend this work by computing the behaviour of the enforcement mechanism using Büchi games. Using Büchi games allows us to compute the behaviour of the enforcement mechanism before the execution, thus allowing to trade time complexity with space complexity. When using an *online* enforcement mechanism (i.e. an enforcement mechanism on a running system), it allows us to produce the output faster in the worst case than in [14] and [13] where the behaviour of the enforcement mechanism was computed every time an event was received. Leveraging games, when an event is received, the enforcement mechanism only follows a path in a graph, and the destination vertex is sufficient to indicate if a stored event should be output or not. Moreover, the generated graph can be visualised to understand the behaviour of the enforcement mechanism, and it also shows clearly when an enforcement mechanism can effectively ensure soundness. We redefine soundness, compliance and optimality using a set-theoretic view of the system, thus providing a global vision of the system based on inputs and outputs at any instant. We give the algorithms (and their complexity analysis) implementing the behaviour of enforcement mechanisms. We finally present a tool implementing the proposed approach.

2 PRELIMINARIES AND NOTATION

General 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^*. The *length* of a finite word w is noted $|w|$, and the *empty word* is noted ϵ. Σ^+ stands for $\Sigma^* \setminus \{\epsilon\}$. A *language* over Σ is any subset $L \subseteq \Sigma^*$. The concatenation of two words w and w' is noted $w.w'$ (or ww' when clear from the context). A word w' is a *prefix* of a word w, noted $w' \preccurlyeq w$, if there exists a word w'' such that $w = w'.w''$. Word w'' is called the *residual* of w after reading the prefix w', noted $w'' = {w'}^{-1}.w$. Note that $w'.w'' = w'.{w'}^{-1}.w = w$. 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 $w' \in \Sigma^*$, $w.w' \in L$. Given a word w and an integer i such that $1 \leq i \leq |w|$, we note $w(i)$ the i-th element of w. Given a tuple $e = (e_1, e_2, \ldots, e_n)$ 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 $(e_1, e_2, \ldots, e_n)$ is sometimes noted $\langle e_1, e_2, \ldots, e_n \rangle$ 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 $\forall \sigma \in \Sigma^*, \forall a \in \Sigma$, $(w.a)_{|\Sigma'} = w_{|\Sigma'}.a$ if $a \in \Sigma'$, and $(w.a)_{|\Sigma'} = w_{|\Sigma'}$ 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 $\preccurlyeq_{\Sigma'}$: $\sigma \preccurlyeq_{\Sigma'} \sigma'$ if $\sigma_{|\Sigma'} \preccurlyeq \sigma'_{|\Sigma'}$.

Automata. An *automaton* is a tuple $\langle Q, q_0, \Sigma, \rightarrow, F \rangle$, where Q is the 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 its reflexive and transitive closure in the usual way. 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 $\mathcal{A}_\varphi$. Given a complete and 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 σ the only state such that $q \xrightarrow{\sigma} (q \text{ after } \sigma)$. The completeness of $\mathcal{A}$ ensures that q after σ exists, and its determinism ensures that it is unique. We also note $\text{Reach}(\sigma) = q_0$ after σ. We extend these definitions to languages: if L is a language, $q \text{ after } L = \bigcup_{\sigma \in L} q \text{ after } \sigma$ and $\text{Reach}(L) = q_0 \text{ after } L$.

Graphs and Büchi games. 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 $\pi = v_0, v_1, \ldots$ is *consistent* with the strategy σ if for any $v_i \in V_0$, $v_{i+1} = \sigma(v_0 \,.\, v_1 \,.\, \cdots \,.\, v_i)$, 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, $\pi \in (V^*F_G)^\omega$). 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.

3 ENFORCEMENT MONITORING OF PROPERTIES USING BÜCHI GAMES

This paper revisits and extends the approach described in [14] by writing the definitions in a set-theoretic view instead of a functional way and proposing a new synthesis technique of *Enforcement Mechanisms* (EM) using Büchi games.

In this section, φ is a regular property defined by a complete and deterministic automaton $\mathcal{A}_\varphi = \langle Q, q_0, \Sigma, \rightarrow, F \rangle$. Recall that the general scheme of an EM is given in Fig. 1, where S represents the running system, σ its execution, E the enforcement mechanism, φ the property to enforce, and $E(\sigma)$ the output of the enforcement mechanism, which should satisfy φ.

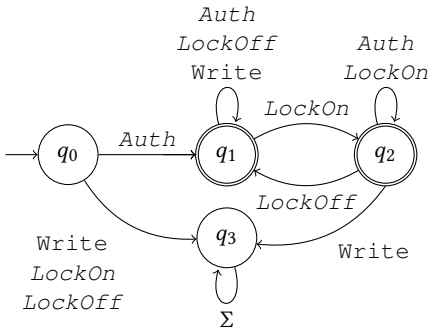

Figure 2: Property φ_{ex} modelling writes on a shared storage device

We consider uncontrollable events[1] in the set $\Sigma_u \subseteq \Sigma$. These events cannot be modified by an EM, so they must be output by the EM whenever they are received. Let us note $\Sigma_c = \Sigma \setminus \Sigma_u$ 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 delay them endlessly, keeping their order unchanged).[2] Thus, an EM may interleave controllable and uncontrollable events.

3.1 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 $\preccurlyeq$):

Definition 3.1 (Enforcement function). A function $f \in \mathcal{F}(\Sigma^*, \Sigma^*)$ is an *enforcement function* if $\forall i_1 \in \Sigma^*, \forall i_2 \in \Sigma^*, (i_1 \preccurlyeq i_2 \wedge (i_1, o_1) \in f \wedge (i_2, o_2) \in f) \implies o_1 \preccurlyeq o_2$. We note $\mathcal{F}_{\text{enf}}$ the set of all enforcement functions.

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

In the sequel, we define the requirements on an EM and express them on enforcement functions. As stated previously, the usual purpose of an EM is to ensure that the executions of a running system satisfy a property, thus its enforcement function has to be *sound*, meaning that its output always satisfies the property:

Definition 3.2 (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 $\mathcal{F}_{\text{snd}}(S)$ the set of all enforcement functions that are sound in S.

The reception of 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. Thus, the enforcement mechanism would not be able to make the property satisfied. Consequently, in Definition 3.2, soundness is not defined for all words in Σ^*, but in a subset S, since the property could not be enforceable from the initial state. In practice, S needs to be extension-closed to ensure that once the enforcement mechanism becomes sound, its output will always satisfy the property afterwards.

The usual notion of *transparency* in enforcement monitoring (cf. [11, 15]) states that the output of an enforcement function is the longest prefix of the input satisfying the property, implying that correct executions are left unchanged. 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). $E \in \mathcal{F}_{\text{enf}}$ is *compliant* with respect to Σ_u and Σ_c, noted compliant(E, Σ_u, Σ_c), if $\forall i \in \Sigma^*, (i, o) \in E \implies (o \preccurlyeq_{\Sigma_c} i \wedge o =_{\Sigma_u} i \wedge \forall u \in \Sigma_u, ((i.u, o') \in E \implies o.u \preccurlyeq o'))$. We note $\mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ the set of all enforcement functions that are compliant with respect to Σ_u and Σ_c.

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. When clear from the context, the partition is not mentioned: E is said to be compliant, we note it compliant(E), and the set of all compliant functions is then denoted $\mathcal{F}_{\text{cpl}}$.

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

In addition, an enforcement mechanism should be optimal in the sense that its output sequences should be maximal while preserving soundness and compliance. We define the optimality of sound and compliant enforcement functions as follows:

Definition 3.4 (Optimality). An enforcement function $E \in \mathcal{F}_{\text{snd}}(S) \cap \mathcal{F}_{\text{cpl}}(\Sigma_u, \Sigma_c)$ is *optimal* in S if:

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

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 restrained to an extension-closed subset of Σ^*, as in the definition of soundness (see Definition 3.2).

Example 3.5. We consider a simple 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. 2) formalises the above requirement. φ_{ex} is not enforceable if the uncontrollable alphabet is $\{\texttt{LockOn}, \texttt{LockOff}, \texttt{Auth}\}$ [3] 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 prevent φ_{ex} from being enforced 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` only when in state q_1.

[1] This notion of uncontrollable event should not be confused with the notion of uncontrollable transition used in some game theory (e.g. [4])

[2] This choice appeared to us as the most realistic one. Extending the notions presented in this section in order to handle enforcement mechanisms with suppression is rather simple.

[3] Uncontrollable events are emphasised in italics.

3.2 Synthesising Enforcement Functions

Example 3.5 shows that some input words cannot be corrected by the EM because of uncontrollable events. Nevertheless, since the received events may lead to a state from which it is possible to ensure that φ will be satisfied (meaning that for any events received as input, the enforcement mechanism can output a sequence that satisfies φ), it is then possible to define a subset of Σ^* in which an enforcement function is sound.

A compliant enforcement mechanism may store the received controllable events to emit them after having received another event, possibly uncontrollable. To ensure soundness, the enforcement mechanism must know if it is possible to emit some of its stored controllable events in order to reach an accepting state, from which it will be able to reach an accepting state even if some uncontrollable events are received later on. Thus, it should compute the set of words it can emit to reach such an accepting state. This set will be called G, and to define it, we solve a Büchi game over a graph representing the possible actions of an enforcement monitor. Solving a Büchi game over a graph with a set of goal nodes (referred to as Büchi nodes) consists in finding the nodes of the graph from where it is possible to reach a Büchi node infinitely often, no matter what the opponent's strategy is. Note that a winning node of a Büchi game is also a winning node for the reachability game on the same graph, with the set of Büchi nodes used as the set of goal nodes, since the Büchi criterion requires that a node is reachable infinitely often (thus, in particular, once). Besides, Büchi games are more permissive than safety games, where one wants to remain in the given set of goal nodes. Since the definition of soundness requires that the output of the enforcement mechanism always eventually satisfies the property, the natural choice of game is a Büchi game. Solving a Büchi game is made by computing a set of nodes of the graph from which there exists a winning strategy. Then, from any of these winning nodes, the player can always come back to a Büchi state, whatever the strategy of the adversary is. Here, we construct a graph such that the enforcement mechanism is a player, and we compute its winning nodes, with the Büchi nodes representing a valid execution. The vertices 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 3.6 (Game graph). The game graph $\mathcal{G}$ is defined as $\mathcal{G} = \langle V, E\rangle$, where

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

A vertex $\langle q, w, l\rangle \in V$ represents the state of the enforcement mechanism: $q \in Q$ is the state of $\mathcal{A}_\varphi$ that has been reached so far by the output of the enforcement mechanism, $w \in \Sigma_c^*$ is the stored controllable events of the enforcement mechanism, and $l \in \{0, 1\}$ indicates that the vertex belongs to the player P_l. In the definition of E, each set of edges represents an action of the enforcement mechanism or the environment. 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.

Unfortunately, this graph has an infinite number of vertices, it is thus impossible to compute a winning strategy in a Büchi game played over this graph. 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. Thus, the enforcement mechanism can emit all the events until it reaches this state for the second time, and then its decision will only depend on the remaining controllable events. Thus, the number of controllable events can be reduced to at most $|Q|$. 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:

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

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 3.7 (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.

$\mathcal{G}$ then has 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 (accepting) vertices being $F \times \Sigma_c^n \times \{0, 1\}$.

Example 3.8. The graph in Fig. 3 is computed from property φ_{ex}, with `Write` abbreviated `w` in the second member of the nodes. The Büchi nodes are double circled, and the winning nodes for player 0 (i.e. nodes in W_0) are in blue and rounded rectangles (in our example, all the Büchi nodes are winning). Each edge has a different colour and a different head depending on the set it belongs to. Blue edges (empty triangular head) belong to E_1, green edges (filled triangular head) belong to E_2, orange edges (empty diamond head) belong to E_4, and red edges (filled diamond head) 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

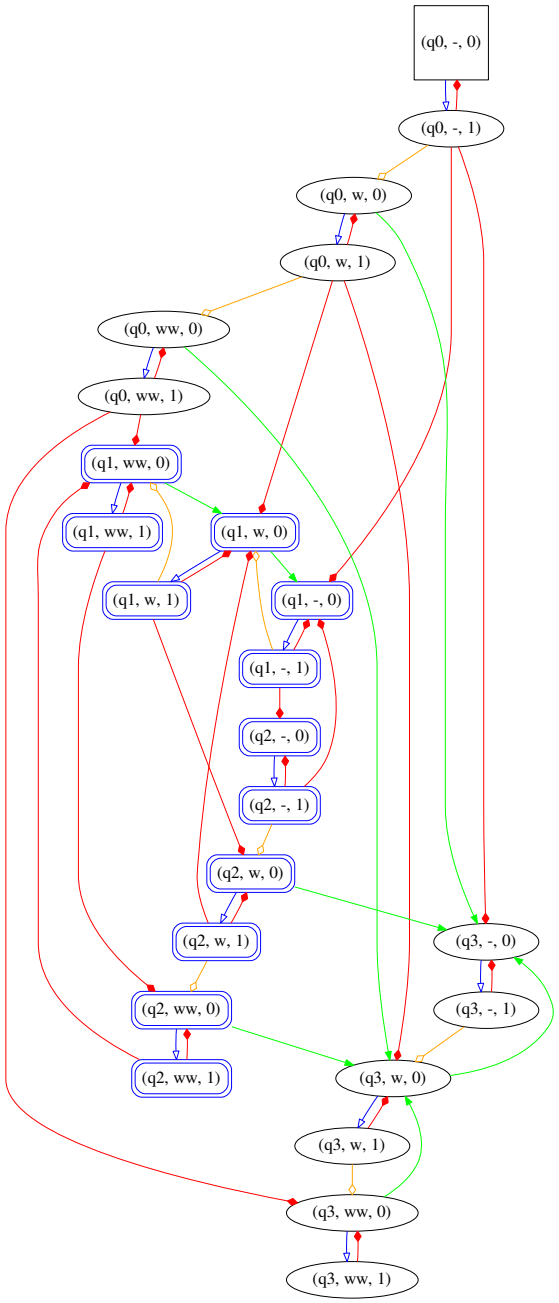

Figure 3: Graph of the game associated to φ_{ex}

edge from a vertex in $\{q_0\} \times \{\epsilon, \mathtt{w}, \mathtt{w.w}\} \times \{1\}$, that corresponds to receiving the uncontrollable event `Auth` (since it leads to a state in $\{q_2\} \times \{\epsilon, \mathtt{w}, \mathtt{w.w}\} \times \{0\}$). 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`, reaching q_1. Then, from q_1, it is possible to emit `Write` events, and if some uncontrollable events are received that lead to q_2, a `LockOff` event must occur to get 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 3.9 (G). For a state $q \in Q$ and a word of controllable events $\sigma \in \Sigma_c^*$, we define the set $\mathrm{G}(q, \sigma)$ as follows:

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

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 3.10 (Functions store_φ, E_φ*).* [4] Function $\mathrm{store}_\varphi \in \Sigma^* \times (\Sigma^* \times \Sigma_c^*)$ is defined by induction on its first member as follows:

- $(\epsilon, \langle \epsilon, \epsilon \rangle) \in \mathrm{store}_\varphi$;
- for $\sigma \in \Sigma^*$ and $a \in \Sigma$, let $(\sigma, \langle \sigma_s, \sigma_c \rangle) \in \mathrm{store}_\varphi$, then:

$$\begin{cases} (\sigma.a, \langle \sigma_s.a.\sigma_s', \sigma_c' \rangle) \in \mathrm{store}_\varphi & \text{if } a \in \Sigma_{\mathrm{u}} \\ (\sigma.a, \langle \sigma_s.\sigma_s'', \sigma_c'' \rangle) \in \mathrm{store}_\varphi & \text{if } a \in \Sigma_{\mathrm{c}} \end{cases}, \text{ where:}$$

$$\kappa_\varphi(q,w) = \max\nolimits_{\preccurlyeq}(\mathrm{G}(q,w) \cup \{\epsilon\}), \text{ for } q \in Q \text{ and } w \in \Sigma_c^*,$$
$$\sigma_s' = \kappa_\varphi(\mathrm{Reach}(\sigma_s.a), \sigma_c), \qquad \sigma_c' = \sigma_s'^{-1}.\sigma_c,$$
$$\sigma_s'' = \kappa_\varphi(\mathrm{Reach}(\sigma_s), \sigma_c.a), \qquad \sigma_c'' = \sigma_s''^{-1}.(\sigma_c.a).$$

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

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

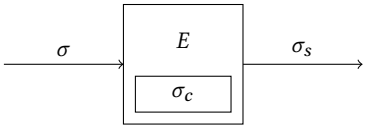

Figure 4: Enforcement function

Figure 4 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 (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 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

[4] E_φ and store_φ depend on Σ_{u} and Σ_{c}, but we did not write it in order to lighten the notations.

G, they correspond to the state reached so far by the output of the enforcement mechanism, and its current buffer, respectively.

Some properties are not enforceable (see Example 3.5), but receiving some events may lead to a state from which it is possible to enforce. Therefore, it is possible to define a set of words, called $\mathrm{Pre}(\varphi)$, such that E_φ is sound in $\mathrm{Pre}(\varphi)$, as stated in Proposition 3.14:

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

$$\mathrm{Pre}(\varphi) = \{\sigma \in \Sigma^* \mid \mathrm{G}(\mathrm{Reach}(\sigma_{|\Sigma_\mathrm{u}}), \sigma_{|\Sigma_\mathrm{c}}) \neq \emptyset\}.\Sigma_\mathrm{c}^*$$

Intuitively, $\mathrm{Pre}(\varphi)$ 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 W_0 ensures that once the set G is not empty, it will never be afterwards, no matter what events are received. Thus, $\mathrm{Pre}(\varphi)$ 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 $\mathrm{Pre}(\varphi)$ 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.

Example 3.12. Considering the property φ_{ex} as shown in Fig. 2, with the uncontrollable alphabet $\Sigma_\mathrm{u} = \{\mathtt{Auth}, \mathtt{LockOff}, \mathtt{LockOn}\}$, $\mathrm{Pre}(\varphi_{\mathrm{ex}}) = \mathtt{Write}^*.\mathtt{Auth}.\Sigma^*$. 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 $\{q_3\} \times \Sigma_\mathrm{c}^n \times \{0,1\}$ will not be in W_0. In order to reach a vertex in W_0 (i.e. a vertex in $\{q_1, q_2\} \times \Sigma_\mathrm{c}^n \times \{0,1\}$), 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, $\{q_1, q_2\} \times \Sigma_\mathrm{c}^n \times \{0,1\} \subseteq W_0$, and thus $\mathrm{Pre}(\varphi_{\mathrm{ex}}) = \mathtt{Write}^*.\mathtt{Auth}.\Sigma^*$, since `Write` events can be buffered while in state q_0 until event `Auth` is received, leading to a vertex in $\{q_1\} \times (\mathtt{Write}^* \cap \Sigma_\mathrm{c}^n) \times \{0,1\} \subseteq W_0$.

Considering the property φ_{ex} defined in Fig. 2, we illustrate in Table 1 the enforcement function by showing the evolution of σ_s and σ_c with input $\sigma = \mathtt{Auth}\,.\,\mathtt{LockOn}\,.\,\mathtt{Write}\,.\,\mathtt{LockOff}$.

Table 1: Evolution of $(\sigma, \langle\sigma_s, \sigma_c\rangle) \in \mathrm{store}_{\varphi_{\mathrm{ex}}}$

σ	σ_s	σ_c
ϵ	ϵ	ϵ
`Auth`	`Auth`	ϵ
`Auth.LockOn`	`Auth.LockOn`	ϵ
`Auth.LockOn.Write`	`Auth.LockOn`	`Write`
`Auth.LockOn.Write.LockOff`	`Auth.LockOn.LockOff.Write`	ϵ

E_φ (as per Definition 3.10) is an enforcement function that is sound with respect to φ in $\mathrm{Pre}(\varphi)$, compliant with respect to Σ_u and Σ_c, and optimal in $\mathrm{Pre}(\varphi)$.

PROPOSITION 3.13. *E_φ is an enforcement function as per Definition 3.1.*

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

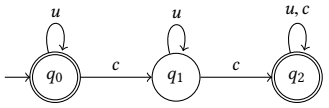

Figure 5: Property that can be enforced by blocking all controllable events c, thus outputting only the uncontrollable ones u.

PROPOSITION 3.14. *E_φ is sound with respect to φ in $\mathrm{Pre}(\varphi)$, as per Definition 3.2.*

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

(1) $\sigma.a \notin \mathrm{Pre}(\varphi)$. Then the proposition holds.
(2) $\sigma.a \in \mathrm{Pre}(\varphi)$, but $\sigma \notin \mathrm{Pre}(\varphi)$. Then the input reaches $\mathrm{Pre}(\varphi)$, and since it is extension-closed, all extensions of σ also are in $\mathrm{Pre}(\varphi)$, and we prove that the proposition holds considering the definition of $\mathrm{Pre}(\varphi)$.
(3) $\sigma \in \mathrm{Pre}(\varphi)$ (and thus, $\sigma.a \in \mathrm{Pre}(\varphi)$ 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 3.15. *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 $(\sigma.a, \sigma_o) \in \mathrm{store}_\varphi$, $(\sigma.a)_{|\Sigma_\mathrm{u}}$, and $(\sigma.a)_{|\Sigma_\mathrm{c}}$, when $a \in \Sigma_\mathrm{c}$ and $a \in \Sigma_\mathrm{u}$.

REMARK 1. *Notice that for some properties, an enforcement function that would block all controllable events may still be sound and compliant. Consider for instance the property represented in Fig. 5, where c is a controllable event, and u an uncontrollable event. Then, outputting only the event u and buffering all the c events allows us to stay in state q_0, which is sound since $\{q_0\} \times (c^* \cap \Sigma_\mathrm{c}^n) \times \{0,1\} \subseteq W_0$. This means that an enforcement mechanism that blocks all controllable events would be sound and compliant. Nevertheless, if c.c is received, it can be output to reach state q_2, which is also accepting and $\{q_2\} \times \Sigma_\mathrm{c}^n \times \{0,1\} \subseteq W_0$. Then it is possible to release more events. Therefore, an enforcement mechanism that would output two c events when they are received would be "better" than the first one blocking all of them, in the sense that its output would be longer (and thus closer to the input).*

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

PROPOSITION 3.16. *E_φ is optimal in $\mathrm{Pre}(\varphi)$, as per Definition 3.4.*

Sketch of proof. The proof is made by induction on the input $\sigma \in \Sigma^*$. Once $\sigma \in \mathrm{Pre}(\varphi)$, we know that $(\sigma, \sigma_o) \in \mathrm{E}_\varphi \implies \sigma_o \models \varphi$ since E_φ is sound in $\mathrm{Pre}(\varphi)$. 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 $\mathrm{Pre}(\varphi)$, since it outputs the longest word that allows us to be both sound and compliant.

3.3 Enforcement Monitors

Enforcement monitors are operational descriptions of enforcement mechanisms. We give a representation of an enforcement mechanism 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 defined in Section 3.2. Enforcement monitors are purposed to ease the implementation of enforcement mechanisms, since they give an operational representation of the enforcement mechanism.

Definition 3.17 (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:

- $C^{\mathcal{E}} = Q \times \Sigma^*$ is the set of configurations.
- $c_0^{\mathcal{E}} = \langle q_0, \epsilon \rangle$ is the initial configuration.
- $\Gamma^{\mathcal{E}} = \Sigma^* \times \{\mathrm{dump}(.), \text{pass-uncont}(.), \text{store-cont}(.)\} \times \Sigma^*$ 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:** $\langle q, a.\sigma_c \rangle \xhookrightarrow{\epsilon / \mathrm{dump}(a) / a}_{\mathcal{E}} \langle q', \sigma_c \rangle$, if $a \in \Sigma_c$, $\mathrm{G}(q, a.\sigma_c) \neq \emptyset$ and $\mathrm{G}(q, a.\sigma_c) \neq \{\epsilon\}$, with $q' = q$ after a,
  - **Pass-uncont:** $\langle q, \sigma_c \rangle \xhookrightarrow{a / \text{pass-uncont}(a) / a}_{\mathcal{E}} \langle q', \sigma_c \rangle$, with $a \in \Sigma_u$ and $q' = q$ after a,
  - **Store-cont:** $\langle q, \sigma_c \rangle \xhookrightarrow{a / \text{store-cont}(a) / \epsilon}_{\mathcal{E}} \langle q, \sigma_c.a \rangle$, 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 $\mathcal{A}_\varphi$ with the output of the monitor. The word of controllable events σ 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 Section 3.2, 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 $q = \mathrm{Reach}(\sigma_s)$.

PROPOSITION 3.18. *The output of the enforcement monitor $\mathcal{E}$ for input σ is $\mathrm{E}_\varphi(\sigma)$.*

In Proposition 3.18, the output of the enforcement monitor is the concatenation of all the outputs of the word labelling the path followed when reading σ.

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 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... 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.

4 Algorithms and Implementation

We describe some of the algorithms that allow us to use a game graph (as per Definition 3.6) to define an enforcement mechanism, and discuss their time complexity. We suppose that the set of winning nodes of the graph is known, as there exist well-known algorithms to compute it (see for example [5]).

4.1 Algorithms

Algorithm 1 computes the set Σ_c^n (see Section 3.2), the set of words that allow to reach a state that is unreachable with all its prefixes from at least one state. Algorithm 1 uses a recursive function described in Algorithm 2. The algorithm builds the words incrementally, adding each possible event to a word in the set, until adding an event does not allow to reach a new state from any state. We make use of arrays of one and two dimensions. Function `arrayInit`(m, n) returns an array of m rows and n columns, when n is not specified, it returns a 1-dimensional array of size m. The returned array is filled with 0.

input : An automaton $\mathcal{A} = \langle Q = \{q_i \mid i \in [1; n]\}, q_0, \Sigma = \Sigma_u \cup \Sigma_c, \delta, F \rangle$
output : The set Σ_c^n as defined in Section 3.2

```
1  reachable ← arrayInit(n, n);
2  lasts ← arrayInit(n);
3  Σ_c^n ← {ϵ};
4  for i ← 1 to n do
5  |  reachable[i, i] ← 1;
6  |  lasts[i] ← q_i;
7  end
8  foreach c ∈ Σ_c do
9  |  Σ_c^n ← Σ_c^n ∪ computeΣ_c^nRec(A, c, reachable, lasts);
10 end
```

Algorithm 1: Algorithm for computing Σ_c^n

Algorithm 1 first initialises Σ_c^n to $\{\epsilon\}$. Then, the two arrays `reachable` and `lasts` are initialised accordingly, i.e. `reachable` is filled with 0, with 1 on the diagonal, and `lasts`$[i]$ is q_i. Words are then added to Σ_c^n by calling the recursive function `computeΣ`$_c^n$`Rec` described in Algorithm 2. The array `reachable` is an array of size $n \times n$, where $n = |Q|$, such that `reachable`$[i, j]$ is equal to 1 if there is a prefix w' of w such that q_i after $w' = q_j$, and 0 otherwise. The array `lasts` is an array of size n, such that

```
input : An automaton 𝒜 = ⟨Q = {q_i | i ∈ [1;n]}, q_0, Σ = Σ_u ∪ Σ_c, δ, F⟩,
        reachable, lasts as defined in Algorithm 1, w ∈ Σ_c^*
output : The set of all the extensions of w that belong to Σ_c^n
1  Function computeΣ_c^n Rec (𝒜, w, reachable, lasts):
2      Σ_c^n ← ∅;
3      foreach c ∈ Σ_c do
4          reachableAfter ← reachable;
5          for i ← 1 to n do
6              let j ∈ [1;n] be such that lasts[i] after c = q_j;
7              reachableAfter[i,j] ← 1;
8              lastsAfter[i] ← q_j;
9          end
10         if reachableAfter ≠ reachable then
11             Σ_c^n ← Σ_c^n ∪{w}∪
                 computeΣ_c^n Rec(𝒜, w . c, reachableAfter, lastsAfter);
12         end
13     end
14     return Σ_c^n;
15 end
```

Algorithm 2: Function computeΣ_c^nRec

for all $i \in [1;n]$, `lasts`$[i] = q_i$ after `w`. The function considers recursively all the extensions of `w`, and add them to Σ_c^n until the array `reachable` stabilises, and then returns the computed Σ_c^n.

The worst-case complexity of Algorithm 1 in terms of assignments is at most $2n + |\Sigma_c|^{n^2-n}$, where $n = |Q|$.

The following algorithms define the primitives of an enforcement mechanism. A state of the enforcement mechanism is represented by a tuple in $V \times V \times \Sigma_c^* \times \Sigma^* \times \Sigma^*$, where V is the set of nodes of the graph $\mathcal{G}$ defined in Definition 3.7. The first node is the node reached by the output of the enforcement mechanism (real node). The second node is the strategy node, i.e. the first winning node that can be reached by outputting the first events of the buffer, or the node reached by outputting all the buffer if such a node does not exists. The third member is the buffer composed of the controllable events that have not been output yet. The fourth member is the input, and the fifth is the output.

The first function is `enforcerInit` (not provided here, due to the lack of space), that returns the initial state of the enforcer, i.e. $(\langle q_0, \epsilon\rangle, \langle q_0, \epsilon\rangle, \epsilon, \epsilon, \epsilon)$.

```
input : A state (⟨q_r, b_r, 0⟩, ⟨q_s, b_s, 0⟩, b, i, o) of the enforcer, an event e ∈ Σ
output : The state of the enforcer after having received e
1  Function enforcerEventReceived((⟨q_r, b_r, 0⟩, ⟨q_s, b_s, 0⟩, b, i, o), e):
2      if e ∈ Σ_c then
3          if b_r . e ∈ Σ_c^n then
4              r ← ⟨q_r, b_r . e, 0⟩;
5          else
6              r ← ⟨q_r, b_r, 0⟩;
7          end
8          w ← min_≼({w' ≼ b_s | w'^{-1} . (b_s . e) ∈ Σ_c^n});
9          s ← ⟨q_s after w, w^{-1} . (b_s . e), 0⟩;
10         return (r, s, b . e, i . e, o);
11     else / * e ∈ Σ_u                                              * /
12         r ← ⟨q_r after e, b_r, 0⟩;
13         w ← min_≼({w' ≼ b | ⟨q_r after e after w', max_≼({w'' ≼
             w'^{-1} . b | w'' ∈ Σ_c^n}), 0⟩ ∈ W_0} ∪ {b});
14         s ← ⟨q_r after e after w, max_≼({w' ≼ w^{-1} . b | w' ∈ Σ_c^n}), 0⟩;
15         return (r, s, b, i . e, o . e);
16     end
17 end
```

Function enforcerEventReceived(state, event)

Function `enforcerEventReceived` computes the next state of the enforcer after the reception of an event. If the event is controllable, then only the strategy node is updated; if the event is uncontrollable, then it is immediately emitted and the real node is changed accordingly, then the strategy node is computed from this new node. The time complexity of this function is linear in the size of the buffer.

```
input : A state (r, s, b, i, o) of the enforcer
output : EMIT if it is possible to emit some controllable events, DONTEMIT
         otherwise
1  Function enforcerGetStrat ((r, s, b, i, o)):
2      if r ≠ s and s ∈ W_0 then
3          strat ← EMIT;
4      else
5          strat ← DONTEMIT;
6      end
7      return strat;
8  end
```

Function enforcerGetStrat(state)

```
input : A state (⟨q_r, b_r, 0⟩, ⟨q_s, b_s, 0⟩, e . b, i, o) of the enforcer, where
        e ∈ Σ_c and b ∈ Σ_c^*
output : The state of the enforcer after having emitted the first controllable event
         of its buffer
1  Function enforcerEmit ((⟨q_r, b_r, 0⟩, ⟨q_s, b_s, 0⟩, e . b, i, o)):
2      s ← ⟨q_s, b_s, 0⟩;
3      r ← ⟨q_r after e, b, 0⟩;
4      w ← min_≼({w' ≼ b | ⟨q_r after e after w', max_≼({w'' ≼ w'^{-1} . b |
         w'' ∈ Σ_c^n}), 0⟩ ∈ W_0} ∪ {b});
5      s ← ⟨q_r after e after w, max_≼({w' ≼ w^{-1} . b | w' ∈ Σ_c^n}), 0⟩;
6      return (r, s, b, i, o . e)
7  end
```

Function enforcerEmit(state)

Function `enforcerGetStrat` returns the strategy to follow. If the strategy node is ahead of the real node, and it is a winning node, then the strategy is to emit the first event of the buffer. Otherwise, the strategy is not to emit. This function has a constant time complexity.

Function `enforcerEmit` emits the first event of the buffer. The real node is updated accordingly, and the strategy node is unchanged except if the real node caught up with it (i.e. they are equal), in which case the strategy node is updated accordingly. The time complexity of this function is linear in the size of the buffer.

```
input : A property φ, described by an automaton 𝒜, the game graph 𝒢
        associated to φ, the input sequence of events, through the function
        read()
output : The output of the enforcer mechanism
1  EM ← enforcerInit_¿ 𝒜, 𝒢_¿;
2  while The input sequence has not been read entirely do
3      e ← read();
4      EM ← enforcerEventReceived(EM, e);
5      while enforcerGetStrat(EM) = EMIT do
6          EM ← enforcerEmit(EM);
7      end
8  end
```

Algorithm 3: Main algorithm to enforce a property

Algorithm 3 describes the main algorithm that uses all these functions to actually enforce a property. It needs one more function: `read()` which returns the next input event. The algorithm

first creates an enforcement monitor for the given property with `enforcerInit`, then all the events from the input are read with function `read`, and fed to the enforcer with function `enforcerEventReceived`. Then, the enforcer emits a maximal number of events from its buffer with `enforcerEmit`, i.e. until `enforcerGetStrat` indicates that is should not emit. Note that emitting events is done by adding the events to the output of the enforcer. The time complexity of this algorithm between two calls to `read` is linear in the size of the buffer, and thus does not depend on the size of the automaton.

4.2 Implementation

We implemented the algorithms in the C programming language. Our tool takes as input a file describing an automaton, and reads the events from its standard input. It outputs information on the evolution of the state of the enforcement mechanism as well as a summary of the execution when it has ended on its standard output. This approach allows us to adapt easily the tool in order to use it with off-the-shelf applications. The tool first creates the game graph from the automaton, then solves the Büchi game on it. Then, the enforcement mechanism is initialised, and then used to compute the controllable events that can be emitted. When the end of the input is reached, the tool displays first the input of the enforcement mechanism, then its output and the remaining events of its buffer at the end of the the execution. Lastly, it displays a verdict indicating whether the execution ended in an accepting state of the automaton.

Performance Analysis. We provide some execution times of our tool, running on the example given in the paper (φ_{ex}). We evaluated it on 20 randomly selected inputs of 20 events each. Table 2 shows the inputs and the corresponding mean times taken by the enforcement mechanism to compute its output after each event. The times are given in nanoseconds. The means have been computed over 100 iterations. The inputs are abbreviated: w stands for `Write`, f for `LockOff`, n for `LockOn`, and a for `Auth`. The results have been obtained using a computer running Ubuntu 16.04 with a 3.40 GHz Intel Core i7 CPU with 16 GB RAM. In Table 2, all the values have the same order of magnitude, showing that the execution time depends little on the input sequence. Thus, the execution time of our enforcement mechanism is quite stable, which makes it reliable.

5 RELATED WORK AND DISCUSSION

Runtime enforcement was pioneered by the work of Schneider with security automata [15], a runtime mechanism for enforcing safety properties. In [15], monitors are able to stop the execution of the system once a deviation of the property has been detected. Later, Ligatti et al. proposed edit-automata [11], a more powerful model of enforcement monitors able to insert and suppress events from the execution, thus permitting to enforce non-safety properties. Later, Falcone et al. proposed more general models where the monitors can be synthesised from regular properties [10]. Another recent approach by Dolzehnko et al. [6] introduces Mandatory Result Automata (MRAs). MRAs extend edit-automata by refining the input-output relationship of an enforcement mechanism and thus allowing a more precise description of the enforcement abilities of an enforcement mechanism in concrete application scenarios. All these approaches do not consider uncontrollable events.

Basin et al. [2] introduced uncontrollable events for security automata [15]. The approach in [2] allows to enforce safety properties where some of the events in the specification are uncontrollable. More recently, they proposed a more general approach [1] related to enforcement of security policies with controllable and uncontrollable events. They presented several complexity results and how to synthesise enforcement mechanisms, but they did not provide a tool implementation.

To our knowledge, two runtime enforcement methods using games have been proposed. In [3], Bloem et al. focus on enforcement of safety properties for reactive hardware systems, i.e. systems with boolean signals as inputs and outputs. They propose a method based on a 2-players safety games in order to synthesise a variant of an enforcement monitor called a *safety shield* and present a tool implementing this approach. This shield ensures correctness (i.e. soundness), and minimum interference according to a notion of distance permitting to measure the deviation between the output and the input of the shield. More recently, Wu et al. propose in [16] to improve the algorithm of [3] in the way that it takes the best recovery strategy among all possible ones, permitting to minimise the deviation, especially in case of burst errors. These two approaches are limited to safety properties and do not support uncontrollable events.

6 CONCLUSION AND FUTURE WORK

This paper revisits the work done in [13, 14] and introduces another way to compute the behaviour of the defined enforcement mechanisms in case of uncontrollable events using a Büchi game. Thus, we define enforcement monitors at two levels of abstraction, one is functional and the second operational. As in [13, 14], we consider that some events are uncontrollable, meaning that they are only observable by the enforcement monitor, that must output them when they are received. We introduce a different way to compute the behaviour of the enforcement mechanism using a Büchi game, that is equivalent to the behaviour of the enforcement mechanism described in [13, 14]. Given a property, we build a graph over which we solve a Büchi game representing the behaviour of the enforcement mechanism. Even though this graph should have an infinite number of vertices, it is possible to reduce it to a finite number, which allows us to store it. The behaviour of the enforcement mechanism only depends on the "current" vertex in the graph. When receiving an event, the enforcement mechanism only updates the current vertex by following some path in the graph, instead of computing again the set of winning configurations. This reduces the time spent in computing the behaviour of the enforcement mechanism, which is one of the main inconveniences when using enforcement mechanisms on running systems. Computing and storing the graph allows us to compute offline the behaviour of the enforcement mechanism, providing better performance at runtime.

We also provide a new way to describe soundness, compliance and optimality with a global view of the system based on inputs and outputs, and we present an implementation showing the effectiveness of the approach.

As a future work, we intend to investigate the runtime enforcement of timed properties [12] using timed games. We consider using the method introduced in this paper to enforce timed properties, as

Table 2: Table of the mean execution times of our tool for different inputs

Input	Times																			
f n a w w a a a w a w a w f n w w n n	724	515	355	275	318	287	677	471	321	253	592	233	453	230	500	437	231	241	555	607
f f n n w f a w n w f a f f n n a w n f	707	534	364	423	251	345	435	222	314	285	794	389	337	320	328	321	319	244	453	969
n n w n f n a n n n a w w f a a n w n f	726	507	257	470	336	282	424	278	268	428	305	218	290	617	372	312	316	241	436	495
a a f a a a w n f f w n f w f a a a n a	700	498	394	414	281	320	545	292	320	427	322	288	279	282	273	275	274	301	383	314
n w f a n f n w a f n a a a w f a a w w	728	307	505	460	310	294	412	216	299	317	425	285	277	276	279	610	368	310	238	379
a w f w a w a w a a w w f a a n n w n a	689	617	464	459	318	432	364	287	307	426	285	276	317	274	267	293	287	217	437	318
n w a a n a n f n f f n w w f n a n w f	692	304	472	452	313	275	417	308	277	442	300	281	216	293	610	370	317	312	243	562
a a f f w a a a n a a n a a n w f w a n	677	502	392	429	468	346	354	273	314	413	271	278	264	263	278	221	450	324	377	318
w a n n a f n n w f f a a f w w f w f n	74	1309	432	421	293	444	353	272	222	621	288	274	275	275	326	289	272	309	380	315
n w f w a f n w n w n w f w n a a a f a	710	302	492	221	452	317	351	347	591	256	586	226	453	224	453	428	402	396	458	516
a w a w n a f a n n w w f w w f n n w n	680	620	405	461	344	342	405	281	316	432	223	254	571	300	283	303	283	383	286	365
w n f w w a n w n w n n f f f f w w n f	77	1010	501	246	426	629	454	336	420	240	617	414	401	386	380	379	230	244	531	663
a a n a n n f f n a a f w n a a w f f n	690	506	379	513	326	333	388	293	306	421	292	284	470	287	280	278	224	474	394	320
f a f w a a a w a n n n f a a a a f n n	735	503	374	363	422	283	405	221	288	323	428	282	285	279	276	272	268	291	315	379
n n n w w w w w f w n f w n w n w n f f	712	503	335	283	314	286	262	329	788	255	668	459	230	478	228	492	229	515	565	635
a a a a w w f f a f f f w w n f n f n a	679	512	345	409	483	369	430	315	309	434	294	279	295	287	290	287	287	311	387	320
n f w n f n a a n w a n w n w w f a w w	711	529	258	478	330	290	422	278	280	217	462	278	276	599	254	240	501	420	232	368
f f n a w w a f w f a a n a a n w w n f	704	519	352	418	240	230	365	394	278	630	513	325	328	305	297	318	237	236	497	528
w a n a w w w w n a w n f f f w f n n w	106	1320	447	438	226	247	324	352	714	458	388	475	1077	312	282	346	374	432	324	204
a n n f f f n n f w a f w w f n f n n n	678	527	342	439	299	331	359	276	306	598	348	278	355	283	283	282	272	306	380	308

done in [9] without uncontrollable events and with suppression; and in [13] with uncontrollable events and without suppression. The graph would be bigger due to the presence of clocks in the automaton, but performance should be improved. The gain in computation would be higher, and this would even be more interesting in this setting since it might allow timed properties to be enforced on devices with little computational power.

Acknowledgement

The work reported in this article has been done in the context of the COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology). It was also partially supported by French Region Nouvelle Aquitaine, and Bordeaux INP.

REFERENCES

[1] Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable security policies revisited. ACM Trans. Inf. Syst. Secur. 16(1), 3:1–3:26 (Jun 2013), http://doi.acm.org/10.1145/2487222.2487225

[2] Basin, D., Klaedtke, F., Zalinescu, E.: Algorithms for monitoring real-time properties. In: Khurshid, S., Sen, K. (eds.) Proceedings of the 2nd International Conference on Runtime Verification (RV 2011). Lecture Notes in Computer Science, vol. 7186, pp. 260–275. Springer-Verlag (2011)

[3] Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: - runtime enforcement for reactive systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 533–548 (2015)

[4] Bulychev, P., Chatain, T., David, A., Larsen, K.G.: Efficient on-the-fly Algorithm for Checking Alternating Timed Simulation, pp. 73–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04368-0_8

[5] Chatterjee, K., Henzinger, M.: An o(n2) time algorithm for alternating bÜchi games. In: Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1386–1399. SODA '12, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2012), http://dl.acm.org/citation.cfm?id=2095116.2095225

[6] Dolzhenko, E., Ligatti, J., Reddy, S.: Modeling runtime enforcement with mandatory results automata. International Journal of Information Security 14(1), 47–60 (Feb 2015), http://dx.doi.org/10.1007/s10207-014-0239-8

[7] Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)

[8] Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer 14(3), 349–382 (2012), http://dx.doi.org/10.1007/s10009-011-0196-8

[9] Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Systems & Control Letters 123, 2–41 (2016)

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

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

[12] Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods in System Design 45(3), 381–422 (2014)

[13] Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal Enforcement of (Timed) Properties with Uncontrollable Events. Mathematical Structures in Computer Science (MSCS) (To appear)

[14] Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H.: Enforcement of (timed) properties with uncontrollable events. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015. Lecture Notes in Computer Science, vol. 9399, pp. 542–560. Springer International Publishing (2015)

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

[16] Wu, M., Zeng, H., Wang, C.: Synthesizing runtime enforcer of safety properties under burst error. In: 8th NASA Formal Methods Symposium NFM16. Minneapolis, USA (June 2016)