Bounded-Memory Runtime Enforcement

Saumya Shankar, Antoine Rollet, Srinivas Pinisetty, Yliès Falcone

Bounded-Memory Runtime Enforcement*

Saumya Shankar[1], Antoine Rollet[2], Srinivas Pinisetty[3], and Yliès Falcone[4]

[1] Indian Institute of Technology Bhubaneswar India ss117@iitbbs.ac.in
[2] Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400, Talence, France antoine.rollet@labri.fr
[3] Indian Institute of Technology Bhubaneswar India spinisetty@iitbbs.ac.in
[4] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France ylies.falcone@univ-grenoble-alpes.fr

Abstract. Runtime Enforcement (RE) is a monitoring technique to ensure that a system obeys a set of formal requirements (properties). RE employs an enforcer (a safety wrapper for the system) which modifies the (untrustworthy) output by performing actions such as delaying (by storing/buffering) and suppressing events, when needed. In this paper, to handle practical applications with memory constraints, we propose a new RE paradigm where the memory of the enforcer is bounded/finite. Besides the property to be enforced, the user specifies a bound on the enforcer memory. Bounding the memory poses various challenges such as how to handle the situation when the memory is full, how to optimally discard events from the buffer to accommodate new events and let the enforcer continue operating. We define the bounded-memory RE problem and develop a framework for any regular property. The proposed framework is implemented and its performance evaluated via some examples from application scenarios indicates that the enforcer has reasonable execution time overhead.

Keywords: Formal methods · Runtime enforcement · Automata.

1 Introduction

Runtime Enforcement (RE) [9,12,3,4,6,20,8] is a monitoring technique to ensure that a system complies with a set of formal requirements (properties) at runtime. An enforcer can be considered as a safety wrapper for the system, which modifies an (untrustworthy) input (in the form of a sequence of events) into an output sequence that complies with the specified property. RE usually aims at ensuring the so-called *soundness* (the output must satisfy the property) and *transparency* (a correct input should remain unchanged).

We focus on the online enforcement of regular properties meaning that enforcement of property φ is done on the fly. The general schema is depicted in Fig. 1, where an enforcer is placed between an event emitter and an event receiver

*This work has been partially supported by IIT Bhubaneswar Seed Grant (SP093). Y. Falcone acknowledges the support from the H2020-ECSEL-2018-IA call –Grant Agreement number 826276 (CPS4EU), from the French ANR project ANR-20- CE39-0009 (SEVERITAS), the Auvergne-Rhône-Alpes research project MOAP, and LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

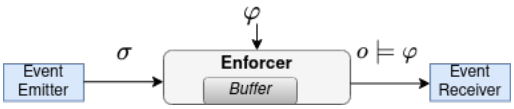

Fig. 1: Enforcement mechanism

that executes asynchronously. The enforcer takes a sequence of events σ as input and transforms it into a sequence of events o that is correct with respect to φ. The enforcer is equipped with an internal memory (*buffer*) to store some events that are received as input which would be released as output only when the satisfaction of the property is ensured. We will use the word *buffer* for referring to the internal memory of the enforcer throughout the paper.

In usual RE mechanisms such as [9,12], the buffer of the enforcer is considered to be infinite. But this assumption is obviously not realistic in the case of a real implementation of the enforcer: an internal buffer is necessarily bounded [10,22]. Then an important question arises: what should be done when the bound of the buffer is reached? To illustrate the problem, consider for instance an enforcement mechanism protecting a critical system by filtering dangerous requests in a network, i.e., inputs that are ill-formatted or suspicious. The situation where the buffer of this protecting mechanism is full is also a critical situation which has to be considered carefully. Naive reactions may lead to dangerous inputs being transmitted to the system and then make the enforcement mechanism useless.

In this work, we study RE with a bounded buffer, i.e., we will see how our enforcer tackles the situation when the buffer is finite. We allow an enforcer to continue operating even when its buffer is full. To handle the situation where the buffer of the enforcer is full, a simple possibility would be to discard the received event, or to remove the oldest one. However, these approaches do not guarantee compliance with the specified property or minimal deviation from an "ideal" enforcer (i.e., an enforcer without memory limitation). Thus, cleaning the buffer when it is full in an optimal way (minimal dropping of events, minimal deviation) in order to continue enforcement becomes the major challenge of a bounded-memory enforcement problem.

Formally speaking, given a regular property φ, and maximum size of buffer k, we aim to synthesize an enforcer that takes as input a word σ and outputs a word o that (i) satisfies φ (soundness), and (ii) is a prefix, or subword of input σ (transparency). In addition, (iii) the output should be as long as possible (optimality) and equivalent to one produced by an enforcer with unbounded memory (∞-compatible), as explained in Sect. 4. We refer to this problem as *Bounded-Memory Runtime Enforcement*.

Context and Objectives. We tackle the problem of obtaining an enforcer given a regular property specified as a Deterministic Finite-state Automaton (DFA). The enforcer intervenes when an execution is about to violate the property being enforced by catching events. The synthesized enforcers have the following abilities: storing events in a buffer, releasing them when the property is finally satisfied, and suppressing events, but only if there is no other way to avoid a buffer overflow. At the point of filtering them out, the filtered/suppressed events should be invariant with respect to the property (DFA's) language. The number of events suppressed in the buffer must be minimal. As illustrated in the

abstract architecture in Fig. 1, we consider that the emitter and receiver run in an asynchronous manner, thus delaying an event from the emitter does not have any impact on its successive events. These are reasonable assumptions for many practical applications such as networks and components in systems like autonomous vehicles.

Contributions. We introduce the first formal framework for bounded-memory runtime enforcement. The notions of soundness and monotonicity are similar to the ones used in the standard RE frameworks [11]. However, transparency is modified to take into account the possibility of suppressing/dropping events when needed. In addition, we propose a new notion of optimality. At an abstract level, we model enforcers as functions that transform words. We define the constraints that an enforcement function (for some φ) should satisfy. We present algorithms describing how the proposed enforcement functions can be implemented. All our results are formalized. The proposed algorithms are implemented in Python and are evaluated using some example properties, and also using properties based on application scenarios related to concurrency and autonomous vehicles. The overhead of enforcers is observed to be reasonable.

2 Preliminaries and Notations

Languages: A (finite) word w over a finite alphabet Σ is a finite sequence of elements of Σ. The length of w, denoted as $|w|$, is the number of elements in w. The empty word over Σ is denoted by ϵ. The sets of all words and all non-empty words are denoted by Σ^* and Σ^+ respectively. A language or a property over Σ is any subset of Σ^* and is denoted by φ.

The concatenation of two words w and w' is denoted by $w \cdot w'$. A word w' is a prefix of word w, denoted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$, and $w' \prec w$ if additionally $w' \neq w$; conversely w is said to be an extension of w'. The set $\text{pref}(w)$ denotes the *set of prefixes* of w and subsequently, $\text{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathcal{L}} \text{pref}(w)$ is the set of prefixes of words in $\mathcal{L}$. A language $\mathcal{L}$ is *prefix-closed* if $\text{pref}(\mathcal{L}) = \mathcal{L}$ and *extension-closed* if $\mathcal{L} \cdot \Sigma^* = \mathcal{L}$.

A word $w' = a_1...a_n$ is a subword of w, denoted $w' \triangleleft w$, if w' can be obtained by deleting letters from w or, equivalently, $w = w_0 a_1 w_1 ... a_n w_n$ for some $w_0, ..., w_n \in \Sigma^*$. Given a n-tuple of symbols $e = (e_1, ..., e_n)$, for $i \in [1, n]$, $\Pi_i(e)$ is the projection of e on its i-th element ($\Pi_i(e) = e_i$).

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

Deterministic and complete automata: A deterministic and complete automaton A is a tuple $A = (Q, q_0, \Sigma, \delta, F,)$ where, Q is the set of states, $q_0 \in Q$ is the initial state, Σ is the finite alphabet, $\delta : Q \times \Sigma \to Q$ is the (total) transition function and $F \subseteq Q$ is the set of accepting states. A dead state of an automaton is a state from where there is no way for the automaton to reach an accepting state [5]. The transition function δ is extended to words by setting $\delta(q, \epsilon) = q$, and $\delta(q, a \cdot \sigma) = \delta(\delta(q, a), \sigma)$, for any $q \in Q, a \in \Sigma, \sigma \in \Sigma^*$.

[5] A dead state is represented by a square throughout the paper.

Languages of automata: A word σ is accepted by A starting from state q if $\delta(q,\sigma) \in F$, and σ is accepted by A if σ is accepted starting from the initial state q_0. The language of A starting from state q is denoted $L(A,q)$ and is the set of all accepted words from $q : L(A,q) = \{\sigma \in \Sigma^* | \delta(q,\sigma) \in F\}$. The language of A, denoted $L(A)$, is $L(A,q_0)$, i.e., the language of A from the initial state q_0. A word w satisfying φ is denoted by $w \models \varphi$, meaning w belongs to the language accepted by the automaton defining φ. The next lemma relates accepted words and the states reached by their prefixes in an automaton.

Lemma 1. $\forall \sigma, \sigma' \in \Sigma^* : \sigma \cdot \sigma' \in L(A) \iff (\sigma' \in L(A, \delta(q_0, \sigma)))$

Lemma 1 states that given any two words $\sigma, \sigma' \in \Sigma^*$, the word obtained by concatenating them ($\sigma \cdot \sigma'$) belongs to the language of A if and only if the word σ' belongs to the language accepted by A starting from the state reached by reading σ in A (i.e., from $\delta(q_0,\sigma)$).

Minimal automata: An automaton A is minimal, if there does not exist an automaton A' with states Q' such that $L(A) = L(A')$ and $|Q'| < |Q|$.

Any non-deterministic, incomplete and non-minimal automaton can be determinized, completed and minimized. Hence, in this paper we consider only deterministic, complete and minimal automata and the term *"automaton"* refers to a *"deterministic, complete and minimal automaton"*.

Equivalence of two words: Two words σ and σ' are φ-equivalent, noted $\sigma \sim_\varphi \sigma'$ if all their continuations evaluate equivalently with respect to φ. Formally: $\sigma \sim_\varphi \sigma'$ iff $\forall \sigma'', \sigma.\sigma'' \models \varphi \Leftrightarrow \sigma'.\sigma'' \models \varphi$.

Lemma 2. *If φ is defined by a deterministic and minimal automaton with transition function δ and initial state q_0:* $\sigma \sim_\varphi \sigma' \Leftrightarrow \delta(q_0,\sigma) = \delta(q_0,\sigma')$.

3 Runtime Enforcement with Unbounded Buffer

We adapt an RE framework based on [15,9] where enforcers are synthesized from regular properties modeled as automata. The input-output behaviour of an enforcer is specified by an enforcement function. The enforcement function E^φ transforms some input word σ which is possibly incorrect w.r.t. φ into a word satisfying φ. Enforcement mechanisms in [15,9] cannot change the order of events, cannot suppress/insert events and have only the ability of blocking and delaying events (by storing them internally in a buffer) when a violation is detected. Thus, when considering the mechanisms in [15,9], the output produced by the enforcer $E^\varphi(\sigma)$ is a prefix of the input word σ. In this work, in addition to buffering events, we also consider suppressing events when there is no possible continuation (future) of the current observation that can lead to the satisfaction of the desired property φ in the future.

Definition 1 (Enforcer). *Given property $\varphi \subseteq \Sigma^*$, a runtime enforcer for φ is a function, $E^\varphi : \Sigma^* \to \Sigma^*$, satisfying the constraints in Tab. 1.*

Table 1: Constraints on an enforcer.

Soundness	**(Snd)**	$\forall \sigma \in \Sigma^* : E^\varphi(\sigma) \neq \epsilon \implies E^\varphi(\sigma) \in \varphi$
Monotonicity	**(Mo)**	$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies E^\varphi(\sigma) \preccurlyeq E^\varphi(\sigma')$
Transparency	**(Tr1)**	$\forall \sigma \in \Sigma^* \setminus \mathrm{pref}(\varphi) : E^\varphi(\sigma) \lhd \sigma$
	(Tr2)	$\forall \sigma \in \mathrm{pref}(\varphi) : E^\varphi(\sigma) \preccurlyeq \sigma$
Optimal Suppression	**(Opt)**	$\forall \sigma \in \Sigma^*, \forall a \in \Sigma : \sigma \in \mathrm{pref}(\varphi) \wedge \sigma \cdot a \notin \mathrm{pref}(\varphi) \implies \forall \sigma_{\mathrm{con}} \in \Sigma^* : E^\varphi(\sigma \cdot a \cdot \sigma_{\mathrm{con}}) = E^\varphi(\sigma \cdot \sigma_{\mathrm{con}})$

Soundness means that for any input word σ, if the output $E^{\varphi}(\sigma)$ is not empty ($\neq \epsilon$), then it must satisfy φ. *Monotonicity* expresses that the output of the enforcer for an extended input word σ' of an input word σ, extends the output produced by the enforcer for σ, i.e., E^{φ} is a growing function over relation $\preccurlyeq$. *Transparency* is expressed as a conjunction of constraints **(Tr1)** and **(Tr2)**. **(Tr1)** expresses that for an input word σ, if there is no possible continuation of σ that can lead to the satisfaction of φ in the future (i.e., σ is not a prefix of a word that belongs to φ), the output produced by the enforcer is a subword of σ (i.e., obtained by discarding/suppressing some events from σ). **(Tr2)** expresses that for an input word σ, if there is any possible continuation of σ that can lead to the satisfaction of φ in the future (i.e., σ is a prefix of a word that belongs to φ), the output produced by the enforcer is a prefix of σ (i.e., no event from σ can be suppressed/discarded). *Optimal suppression* **(Opt)** expresses that for any word σ which is a prefix of a word that belongs to φ, when extended with an event $a \in \Sigma$ such that $\sigma \cdot a$ does not have any extension that satisfies φ, event a should be suppressed by the enforcer. Let us see the definition of an enforcement function that incrementally builds the output.

Definition 2 (Enforcement Function). *Given a property $\varphi \subseteq \Sigma^*$, the enforcement function is $E^{\varphi} : \Sigma^* \to \Sigma^*$, and is defined as $E^{\varphi}(\sigma) = \Pi_1(\text{store}^{\varphi}(\sigma))$, where $\text{store}^{\varphi} : \Sigma^* \to \Sigma^* \times \Sigma^*$ is defined as:*

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

$$\text{store}^{\varphi}(\sigma \cdot a) = \begin{cases} (\sigma_s \cdot \sigma_c \cdot a, \epsilon) & \text{if } \sigma_s \cdot \sigma_c \cdot a \in \varphi, \\ (\sigma_s, \sigma_c \cdot a) & \text{if } \sigma_s \cdot \sigma_c \cdot a \notin \varphi \wedge \sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi), \\ (\sigma_s, \sigma_c) & \text{otherwise,} \end{cases}$$

$$\text{with } (\sigma_s, \sigma_c) = \text{store}^{\varphi}(\sigma).$$

Function store^{φ} takes a word over Σ as input and returns a pair of words over Σ. The first element of the output of function store^{φ} corresponds to the output of the enforcement function; it is a prefix of (a subword of) the input that satisfies φ; and the second element is a suffix of (a subword of) the input that the enforcer cannot output yet. It corresponds to the buffer of the enforcer. Function store^{φ} is defined inductively: initially, for an empty input, both elements are empty; if σ is read, $\text{store}^{\varphi}(\sigma) = (\sigma_s, \sigma_c)$, and another new event $a \in \Sigma$ is observed, there are three possible cases depending on whether $\sigma_s \cdot \sigma_c \cdot a$ satisfies φ or not or is a prefix of a word that satisfies φ or not.

- If $\sigma_s \cdot \sigma_c \cdot a$ satisfies φ, then the concatenation of the buffer content (σ_c) and event a is released as output (i.e., appended to σ_s), and the buffer σ_c is emptied (i.e., σ_c is set to ϵ);
- If $\sigma_s \cdot \sigma_c \cdot a$ does not satisfy φ, but is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi)$), then the output remains unchanged and the new event a is appended to the buffer σ_c;
- Otherwise ($\sigma_s \cdot \sigma_c \cdot a$ does not satisfy φ and also is not a prefix of a word that satisfies φ, i.e., $\sigma_s \cdot \sigma_c \cdot a \notin \text{pref}(\varphi)$), then both the output and the buffer remain unchanged (i.e., the new event a is suppressed).

Proposition 1 (Soundness, Monotonicity, Transparency and Optimality). *Given property $\varphi \subseteq \Sigma^*$, the enforcement function E^{φ} as per Def. 2 is an enforcer as per Def. 1.*

Example 1 (Enforcement Function). Let us consider property φ defined by automaton A_φ in Fig. 2. State q_0 is the initial and accepting state. Let the input word be *acbabbcacab*. Tab. 2 illustrates the behaviour of the enforcer when the considered input word is processed incrementally event-by-event.

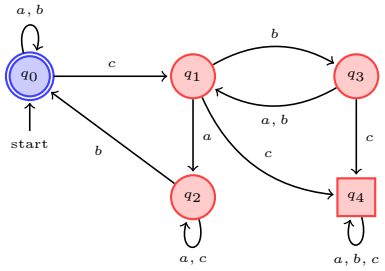

Fig. 2: Property φ of Example 1

	***Input*(σ)**	***Buffer*(σ_c)**	***Output*(σ_s)**
1	*a*	ϵ	*a*
2	*ac*	*c*	*a*
3	*acb*	*cb*	*a*
4	*acba*	*cba*	*a*
5	*acbab*	*cbab*	*a*
6	*acbabb*	*cbabb*	*a*
7	*acbabbc*	*cbabb*	*a*
8	*acbabbca*	*cbabba*	*a*
9	*acbabbcac*	*cbabbac*	*a*
10	*acbabbcaca*	*cbabbaca*	*a*
11	*acbabbcacab*	ϵ	*acbabbacab*

Table 2: Incremental computation by the enforcement function of Example 1

When the enforcer receives the first event a (where σ_c and σ_s are initially both ϵ), φ is satisfied ($\sigma_s \cdot \sigma_c \cdot a \in \varphi$), i.e., the state reached (i.e., q_0), by the enforcer upon event a is an accepting state of A_φ, so the event a is emitted, as can be seen from the 1st row of Tab. 2. Upon receiving the second event c, the φ is not satisfied (i.e., $\sigma_s \cdot \sigma_c \cdot c \notin \varphi$), but $\sigma_s \cdot \sigma_c \cdot c$ is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot c \in \text{pref}(\varphi)$), thus, the event c is added into the (empty) buffer σ_c. The events in rows 3-6 of Tab. 2 are also appended to the contents of σ_c, as $\sigma_s \cdot \sigma_c$ followed by the new input event does not satisfy φ, but is a prefix of a word that satisfies φ. When event c is received (row 7), neither φ is satisfied (i.e., $\sigma_s \cdot \sigma_c \cdot c \notin \varphi$) nor $\sigma_s \cdot \sigma_c \cdot c$ is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot c \notin \text{pref}(\varphi)$) since, state reached (i.e., q_4) upon event c is a dead state of A_φ, thus, event c is suppressed. The events in rows 8-10 of Tab. 2 are also appended to the content of σ_c, as $\sigma_s \cdot \sigma_c$ followed by the new event consumed in each of those steps does not satisfy φ, but is a prefix of a word that satisfies φ. Then, φ is satisfied again (i.e., in row 11, $\sigma_s \cdot \sigma_c \cdot b \in \varphi$) meaning that the state reached (i.e., q_0) upon event b is an accepting state. Hence, the events in σ_c (i.e., *cbabbaca*) and the received event (i.e., b) are emitted by the enforcer (added to σ_s), and buffer σ_c is emptied.

4 Bounded-Memory Runtime Enforcement

In this section, we present the bounded-memory enforcement framework for some property $\varphi \subseteq \Sigma^*$ and buffer size $k \in \mathbb{N}$. Recall that, the enforcer defined in Sect. 3 is equipped with an unbounded buffer to store some events. We now lift the enforcement mechanism defined in Sect. 3 to the bounded-memory case; the buffer capacity is specified as an additional parameter to the enforcer.

4.1 Preliminaries for Bounded-Memory Enforcement

A bounded-memory enforcer is denoted by $E^{\varphi,k}$. Enforcer $E^{\varphi,k}$ for a given property φ is equipped with a buffer of size k and is able to transform an input word σ which is possibly incorrect w.r.t. φ into an output word that is correct w.r.t. φ. In addition, a bounded-memory enforcer also outputs status information indicating whether any event was discarded or not. Thus, enforcer

$E^{\varphi,k} : \Sigma^* \to \Sigma^* \times \{\top, \bot\}$, outputs a tuple consisting of output word (element of Σ^*) and mode information (which is an element of $\{\top, \bot\}$) permitting to warn the user. For any input word $\sigma \in \Sigma^*$, we refer to the output word of the enforcer from the tuple using $E^{\varphi,k}_{\text{out}}(\sigma)$, and the information of the current mode of the enforcer using $E^{\varphi,k}_{\text{mode}}(\sigma)$. For any enforcer $E^{\varphi,k}$, $\text{buff}(E^{\varphi,k}(\sigma))$ is its buffer content after reading σ.

Mode of the enforcer: The mode of the enforcer, $E^{\varphi,k}_{\text{mode}}$ can be $\{\top, \bot\}$, where $\top$ represents nominal mode (none of the elements from the buffer were suppressed; the output of the enforcer is a prefix of the input word), and $\bot$ represents degraded mode (some of the elements from the buffer were suppressed; the output of the enforcer is a subword of the input word) respectively.

Remark 1. When the buffer is full and cannot accommodate new events, it has to be cleaned (by discarding some events present in it) in order to store new events. When any event from the buffer is discarded, the mode of the enforcer changes from $\top$ to $\bot$, and the enforcer remains in $\bot$ mode from then onwards.

4.2 Bounded-Memory Runtime Enforcement: Problem Definition

We formalize the bounded-memory RE mechanism. Several constraints are required on how an enforcer transforms words. We consider that in order to correct an input sequence, the enforcer has abilities similar to the one defined in Sect. 3 such as buffering and suppressing an event when there is no possible continuation that will allow satisfying φ. Thus, constraints such as soundness and monotonicity are adapted from the unbounded setting.

In the bounded setting, since the internal buffer of the enforcer is finite, compared to the unbounded setting we need to handle the additional situation on how the enforcer can continue operating when its buffer is full. We introduce an additional optimality constraint that defines how to optimally (minimally) drop events present in the buffer to accommodate new events and to continue operating. The notion of transparency also slightly changes; it takes into account the mode of the enforcer indicating whether the buffer was full sometime and if any events were dropped or not.

Definition 3 (Bounded Enforcer). *A bounded enforcer for φ is a function, $E^{\varphi,k} : \Sigma^* \to \Sigma^* \times \{\top, \bot\}$, satisfying the following constraints:*

Soundness: $$\forall \sigma \in \Sigma^* : E^{\varphi,k}_{\text{out}}(\sigma) \neq \epsilon \implies E^{\varphi,k}_{\text{out}}(\sigma) \in \varphi \quad \textbf{(SndB)}$$

Monotonicity: $$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies E^{\varphi,k}_{\text{out}}(\sigma) \preccurlyeq E^{\varphi,k}_{\text{out}}(\sigma') \quad \textbf{(MoB)}$$

Transparency:

$$\forall \sigma \in \Sigma^* : E^{\varphi,k}_{\text{mode}}(\sigma) = \bot \vee \sigma \notin \text{pref}(\varphi) \implies E^{\varphi,k}_{\text{out}}(\sigma) \lhd \sigma \quad \textbf{(Tr1B)}$$

$$\forall \sigma \in \Sigma^* : E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge \sigma \in \text{pref}(\varphi) \implies E^{\varphi,k}_{\text{out}}(\sigma) \preccurlyeq \sigma \quad \textbf{(Tr2B)}$$

Optimal Suppression:

$$\begin{aligned} &\forall \sigma \in \Sigma^*, \forall a \in \Sigma : E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge \sigma \in \text{pref}(\varphi) \wedge \sigma \cdot a \notin \text{pref}(\varphi) \\ &\quad \implies \forall \sigma_{\text{con}} \in \Sigma^* : E^{\varphi,k}_{\text{out}}(\sigma \cdot a \cdot \sigma_{\text{con}}) = E^{\varphi,k}_{\text{out}}(\sigma \cdot \sigma_{\text{con}}) \end{aligned} \quad \textbf{(OptsB)}$$

Optimal Mode Change:

$$
\begin{aligned}
&E^{\varphi,k}_{\text{mode}}(\epsilon) = \top \wedge \forall \sigma \in \Sigma^*, \forall a \in \Sigma : \\
&\{E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge (|\sigma_{[|E^{\varphi,k}_{\text{out}}(\sigma)|+1\cdots]}| < k) \implies E^{\varphi,k}_{\text{mode}}(\sigma \cdot a) = \top\} \\
&\wedge \\
&\{E^{\varphi,k}_{\text{mode}}(\sigma) = \bot \vee \big((|\sigma_{[|E^{\varphi,k}_{\text{out}}(\sigma)|+1\cdots]}| > k) \vee (\sigma \cdot a \notin \text{pref}(\varphi))\big) \implies \\
&E^{\varphi,k}_{\text{mode}}(\sigma \cdot a) = \bot\} \\
&\wedge \\
&\{E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge (|\sigma_{[|E^{\varphi,k}_{\text{out}}(\sigma)|+1\cdots]}| = k) \wedge (\sigma \cdot a \in \varphi) \implies E^{\varphi,k}_{\text{mode}}(\sigma \cdot a) = \top\} \\
&\wedge \\
&\{E^{\varphi,k}_{\text{mode}}(\sigma) = \top \wedge (|\sigma_{[|E^{\varphi,k}_{\text{out}}(\sigma)|+1\cdots]}| = k) \wedge (\sigma \cdot a \notin \varphi) \implies E^{\varphi,k}_{\text{mode}}(\sigma \cdot a) = \bot\}
\end{aligned}
\tag{\textbf{OptmB}}
$$

Soundness **SndB** and *Monotonicity* **MoB** constraints are the same as in the unbounded case. *Transparency* is expressed as a conjunction of **Tr1B** and **Tr2B**. **Tr1B** expresses that for an input word σ, if the mode is degraded ($\bot$), or if there is no possible continuation of σ that can lead to the satisfaction of φ in the future (i.e., σ is not a prefix of a word that belongs to φ), the output produced is a subword of σ (i.e., obtained by suppressing some events from σ). **Tr2B** expresses that for an input word σ, if the mode is nominal ($\top$) and if there is at least one possible continuation of σ that can lead to the satisfaction of the φ in the future (i.e., σ is a prefix of a word that belongs to φ), the output produced is a prefix of σ (i.e., no event from σ can be suppressed). *Optimal suppression* **OptsB** expresses that for any word σ, if the mode is ($\top$), and σ is a prefix of a word that belongs to φ, when σ is extended with an event $a \in \Sigma$ such that $\sigma \cdot a$ does not have any extension that will satisfy φ, then event a should be suppressed. *Optimal mode change* **OptmB** is to ensure that the mode should change to $\bot$ only when necessary, and when the mode changes to $\bot$, it cannot revert back to $\top$. For any input σ if the mode is $\top$, it indicates that none of the input events from σ have been discarded (so the output is a prefix of the input and the remaining events from σ are stored in the buffer). If the mode is $\bot$, it indicates that some of the events from the input have been suppressed. It is defined inductively considering the mode reached when reading some given word σ and how the mode should change when a new event a is received, checking the conditions such as whether the input word belongs to φ (or the set of prefixes of words belonging to φ) or not, and if the number of elements in the suffix of the input word that is not released is less than (equal to/greater than) k.

4.3 Functional Definition

We define a bounded-memory enforcer as a function that incrementally builds the output. This definition provides an abstract view of the transformation of an input word performed by a bounded-memory enforcer for some property.

Definition 4 (Bounded Enforcement Function). *A bounded enforcement function is* $E^{\varphi,k} : \Sigma^* \to \Sigma^* \times \{\top, \bot\}$, *and is defined as:*
$E^{\varphi,k}(\sigma) = (\Pi_1(\text{store}^{\varphi,k}(\sigma)), \Pi_3(\text{store}^{\varphi,k}(\sigma)))$, *where:*

$\text{store}^{\varphi,k} : \Sigma^* \to \Sigma^* \times \Sigma^* \times \{\top, \bot\}$ *is defined as:*

- $\text{store}^{\varphi,k}(\epsilon) = (\epsilon, \epsilon, \top)$
- $\text{store}^{\varphi,k}(\sigma \cdot a) =$

$$\begin{cases} (\sigma_s \cdot \sigma_c \cdot a, \epsilon, \mathit{mode}) & \mathit{if}\ \sigma_s \cdot \sigma_c \cdot a \in \varphi, \\ (\sigma_s, \sigma_c \cdot a, \mathit{mode}) & \mathit{if}\ \sigma_s \cdot \sigma_c \cdot a \notin \varphi \wedge \sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi) \wedge |\sigma_c \cdot a| \leq k, \\ (\sigma_s, \text{clean}^{\varphi,k}(\sigma_s, \sigma_c \cdot a), \bot) & \mathit{if}\ \sigma_s \cdot \sigma_c \cdot a \notin \varphi \wedge \sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi) \wedge |\sigma_c \cdot a| > k, \\ (\sigma_s, \sigma_c, \bot) & \mathit{otherwise} \end{cases}$$

with:

- $(\sigma_s, \sigma_c, \mathit{mode}) = \text{store}^{\varphi,k}(\sigma)$
- $E^{\varphi,k}_{\text{out}}(\sigma) = \Pi_1(E^{\varphi,k}(\sigma))$, *and* $E^{\varphi,k}_{\text{mode}} = \Pi_3(E^{\varphi,k}(\sigma))$
- $\text{clean}^{\varphi,k} : \Sigma^* \times \Sigma^+ \to \Sigma^+$

$$\begin{aligned} \text{clean}^{\varphi,k}(\sigma_s, \sigma_{ca}) = \sigma'_c \in \text{maxC}(\text{candidates}^{\varphi,k}(\sigma_s, \sigma_{ca})) \quad & s.t. \\ \forall \sigma''_c \in \text{maxC}(\text{candidates}^{\varphi,k}(\sigma_s, \sigma_{ca})), & \\ \exists i \in [1, |\sigma_{ca}|] : \sigma_{ca[i]} \neq \sigma''_{c[i]} & \\ \implies \exists j \in [1, |\sigma_{ca}|] : \sigma_{ca[j]} \neq \sigma'_{c[j]} \wedge j < i. & \end{aligned}$$

- $\text{maxC} : 2^{\Sigma^+} \to 2^{\Sigma^+}$

  $\text{maxC}(W) = \{y \in W \mid \forall z \in W : |z| \leq |y|\}$
- $\text{candidates}^{\varphi,k} : \Sigma^* \times \Sigma^+ \to 2^{\Sigma^+}$

$$\begin{aligned} \text{candidates}^{\varphi,k}(\sigma_1, \sigma_2) = \{ & \sigma_{2_{[1\ldots i-1]}} \cdot \sigma_{2_{[j+1\ldots k]}} \mid 1 \leq i \leq j < k \\ & \wedge \sigma_1 \cdot \sigma_2 \sim_\varphi \sigma_1 \cdot \sigma_{2_{[1\ldots i-1]}} \cdot \sigma_{2_{[j+1\ldots k]}} \} \end{aligned}$$

The bounded enforcement function $E^{\varphi,k}$ takes a word over Σ as input, and produces a word over Σ and mode (an element from the set $\{\top, \bot\}$) as output. Function $\text{store}^{\varphi,k}$ takes a word over Σ as input and computes a pair of words over Σ and mode as output. The first element of the output of function $\text{store}^{\varphi,k}$ corresponds to the output of the enforcement function; it is a prefix of (a subword of) the input that satisfies φ; and the second element is a suffix of (a subword of) the input that the enforcer cannot output yet. It corresponds to the buffer of the enforcer. The third element indicates the mode of the enforcer.

Function $\text{store}^{\varphi,k}$ is defined inductively: initially for ϵ, the output and buffer content are both ϵ and mode is initially $\top$; if σ is read, $\text{store}^{\varphi,k}(\sigma) = (\sigma_s, \sigma_c, \mathit{mode})$, and another new event $a \in \Sigma$ is observed, then there are four possible cases based on whether $\sigma_s \cdot \sigma_c \cdot a$ satisfies φ or not, etc.

- If $\sigma_s \cdot \sigma_c \cdot a$ satisfies φ then the concatenation of the buffer (σ_c) and the event a is released as output (i.e, appended to σ_s), and the buffer σ_c is emptied (i.e., σ_c is set to ϵ);
- If $\sigma_s \cdot \sigma_c \cdot a$ does not satisfy φ, but is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi)$), and the buffer has capacity to accommodate the received event a (i.e., $|\sigma_c \cdot a| \leq k$) then the output remains unchanged and the new event a is appended to buffer σ_c;
- If $\sigma_s \cdot \sigma_c \cdot a$ does not satisfy φ, but is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi)$), and the buffer is full (i.e., $|\sigma_c \cdot a| > k$) then the output remains unchanged and function $\text{clean}^{\varphi,k}$ is called to clean the buffer (received event is also considered for cleaning) in order to accommodate the event. The mode changes to $\bot$;

- If $\sigma_s \cdot \sigma_c \cdot a$ does not satisfy φ nor is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot a \notin \text{pref}(\varphi)$), then both the output and the buffer remain unchanged (i.e., the new event a is suppressed). The mode changes to $\perp$.

Function clean$^{\varphi,k}$ takes two words over Σ as input; one that corresponds to word released as output (σ_s) and another σ_{ca} which is the current buffer content (σ_c) followed by the received event (a). It produces a word σ'_c as output which should be a subword of $\sigma_c \cdot a$ (obtained by minimally removing events from σ_{ca} such that equivalence of $\sigma_s \cdot \sigma'_c$ is preserved w.r.t $\sigma_s \cdot \sigma_{ca}$). The output word σ'_c should be of maximal length among all the subwords of $\sigma_c \cdot a$ that preserve equivalence and the events discarded from $\sigma_c \cdot a$ are the most obsolete (earliest received events are considered for deletion in this approach; this is an implementation choice but one could choose another strategy) ones.

For this purpose, first function candidates$^{\varphi,k}$ provides the (non-empty) set of all possible candidate subwords of $\sigma_c \cdot a$ preserving equivalence, then function maxC selects all the longest subwords (so that least number of events are discarded from $\sigma_c \cdot a$ and the output is as close as possible to the input). Then, one subword is selected uniquely (the subword discarding the most *obsolete* event from $\sigma_c \cdot a$). This is done by comparing the indexes of $\sigma_c \cdot a$ with the indexes of received subwords from function maxC. The contents of the buffer is then substituted by the output of function clean$^{\varphi,k}$.

Example 2. To illustrate the enforcement function in Def. 4, let us consider the same property φ and the input sequence *acbabbcacab* considered in Example 1 . Suppose the max size of the buffer is 7.

The behaviour of the enforcement function in Def. 4 is the same as the enforcement function in Def. 2 in rows 1-9 of Tab. 3 as the buffer is not full. But, in the 10th row, when event a is received, φ is not satisfied (i.e., $\sigma_s \cdot \sigma_c \cdot a \notin \varphi$), but $\sigma_s \cdot \sigma_c \cdot a$ is a prefix of a word that satisfies φ (i.e., $\sigma_s \cdot \sigma_c \cdot a \in \text{pref}(\varphi)$); however, the buffer is already full (i.e., $|\sigma_c \cdot a| > k$). So, function clean$^{\varphi,k}$ is invoked to accommodate the received event. The possible candidate subwords provided by function candidates$^{\varphi,k}$ and the set of longest subwords picked by function maxC are:

	***Input*(σ)**	***Buffer*(σ_c)**	***Output*(σ_s)**
1	a	ϵ	a
2	ac	c	a
3	acb	cb	a
4	$acba$	cba	a
5	$acbab$	$cbab$	a
6	$acbabb$	$cbabb$	a
7	$acbabbc$	$cbabb$	a
8	$acbabbca$	$cbabba$	a
9	$acbabbcac$	$cbabbac$	a
10	$acbabbcaca$	$cbabba\not{a}$	a
11	$acbabbcacab$	ϵ	$acbabbaab$

Table 3: Incremental computation by the bounded enforcement function (for φ in Fig. 2).

candidates$^{\varphi,7}$ $(a, cbabbac \cdot a) = \{cbbaca, cbaaca, cbabbac, cbabbaa, caca, cbabba\}$
maxC$\{cbbaca, cbaaca, cbabbac, cbabbaa, caca, cbabba\}$=$\{cbabbac, cbabbaa\}$

Function clean$^{\varphi,k}$ chooses *cbabbaa* which is formed after removing event c from $\sigma_c \cdot a$, since event c is the event engaged in minimal cycle (the longest subword of $\sigma_c \cdot a$) and is the most obsolete (cycle) event of $\sigma_c \cdot a$. Thus, the content of σ_c is replaced by *cbabbaa* in the 10th row.

Remark 2. If $n \in \mathbb{N}$ is the number of states in A_φ, and the buffer size $k \geq n$, then it is ensured that the set computed by the function candidates$^{\varphi,k}$ in Def. 4 will be non-empty. This is because the length of a path without cycles between 2 states cannot be greater than the number of states.

Proposition 2 (SndB, MoB, Tr1B, Tr2B, OptsB and OptmB.). *Let $n \in \mathbb{N}$ be the number of states in A_φ. If $k \geq n$, $E^{\varphi,k}$ as per Def. 4 is a bounded enforcer for φ as per Def. 3.*

The following proposition states that when k is considered to be ∞, for any word σ, the output produced by the bounded enforcer for σ is equal to the output produced by the ideal enforcer (as per Def. 2).

Proposition 3 (Case of an infinite/unbounded buffer).
$\forall \sigma \in \Sigma^* : E^{\varphi,\infty}(\sigma) = E^{\varphi}(\sigma)$.

Definition 5 (∞-compatible). *Enforcer $E^{\varphi,k}$ is compatible with $E^{\varphi,\infty}$, noted ∞-compatible$(E^{\varphi,k})$, if $\forall \sigma \in \Sigma^* : E^{\varphi,\infty}(\sigma)\cdot\text{buff}(E^{\varphi,\infty}(\sigma)) \sim_\varphi E^{\varphi,k}_{\text{out}}(\sigma)\cdot\text{buff}(E^{\varphi,k}(\sigma))$.*

A bounded enforcer $E^{\varphi,k}$ is compatible with an ideal unbounded enforcer $E^{\varphi,\infty}$ for φ, if for any input word σ, the concatenation of the output and the buffer content of the unbounded enforcer $E^{\varphi,\infty}$ is φ-equivalent to the concatenation of the output and the buffer content of the bounded enforcer $E^{\varphi,k}$.

Proposition 4 (Optimality of enforcement functions). *Consider any bounded enforcer $F^{\varphi,k}$ (Def. 3). We have: $\forall \sigma \in \Sigma^*, \forall a \in \Sigma$:*

$$\begin{aligned}&(E^{\varphi,k}_{\text{out}}(\sigma) \cdot \text{buff}(E^{\varphi,k}(\sigma)) = F^{\varphi,k}_{\text{out}}(\sigma) \cdot \text{buff}(F^{\varphi,k}(\sigma)))\wedge \\ &\qquad (|E^{\varphi,k}_{\text{out}}(\sigma \cdot a) \cdot \text{buff}(E^{\varphi,k}(\sigma \cdot a))| < |F^{\varphi,k}_{\text{out}}(\sigma \cdot a) \cdot \text{buff}(F^{\varphi,k}(\sigma \cdot a))|) \\ &\implies \neg(\infty\text{-}compatible(F^{\varphi,k}))\end{aligned}$$

Proposition 4 expresses that an enforcer $E^{\varphi,k}$ as per Def. 4 is optimal; if for any other enforcer $F^{\varphi,k}$, the length of the concatenation of its output and the buffer content is greater than the length of the concatenation of output and the buffer content of $E^{\varphi,k}$ for some input, then the output produced by $F^{\varphi,k}$ is not ∞-compatible. Proposition 4 means that there does not exist an enforcer that can clean the buffer in a better way (by discarding less events and being ∞-compatible with the ideal enforcer).

4.4 Enforcement Algorithm

In Sect. 4.3, we provided an abstract view of our bounded-memory enforcement monitoring mechanism, defining it as a function that transforms words. In this section, we provide the overall enforcement algorithm.

Let automaton $A_\varphi = (Q_\varphi, q_0, \Sigma, \delta_\varphi, F_\varphi)$ define φ. We recall that φ models the property that we want to enforce. We devise an online algorithm, which takes A_φ, and the buffer size $k \in \mathbb{N}$ as input parameters.

Enforcement algorithm: In the Algorithm 1, sequence σ_c is the same as in Def. 4. State q holds the state of A_φ reached by taking events that have been emitted by the enforcer, which correspond to events in σ_s in Def. 4. Function `await_event` is used to wait for a new input event. Function `release` takes a sequence of events and releases it as output of the enforcer. Function `suppress` removes an event. Function `clean` deletes events from the buffer in case the buffer is full for an incoming event.

1 Algorithm Enforcer(A_φ, k)

```
1: σ_c ← ϵ
2: q ← q_0
3: while true do
4:     a ← await_event()
5:     if δ_φ(q, σ_c · a) ∈ F_φ then
6:         q ← δ_φ(q, σ_c · a)
7:         release(σ_c · a)
8:         σ_c ← ϵ
9:     else
10:        if L(A_φ, δ_φ(q, σ_c · a))=∅
11:          then
12:            suppress(a)
13:        else
14:            if |σ_c · a| ≤ k then
15:                σ_c ← σ_c · a
16:            else
17:                σ_c ← clean(σ_c · a, A_φ, k, q)
```

Function clean($\sigma_{ca}, \mathcal{A}_\varphi, k, q$)

```
1: function CLEAN(σ_ca, A_φ, k, q)
2:     for i in 1... k+1  do
3:         j=1
4:         while (i + j ≤ k + 2) do
5:             if j = 1 then
6:                 if δ_φ(q, σ_ca[j...j+i−1]) = q then
7:                     return σ_ca[j+i...k+1]
8:             else
9:                 if δ_φ(q, σ_ca[1...j−1] · σ_ca[j...j+i−1]) =
10:                    δ_φ(q, σ_ca[1...j−1]) then
11:                return σ_ca[1...j−1] · σ_ca[j+i...k+1]
12:            j++
```

The algorithm proceeds as follows. Buffer σ_c is initialized to ϵ, and q is initialized with the initial state of A_φ (i.e., q_0). It then enters into an infinite loop waiting for an input event. Upon receiving an event a, if the state reached from q upon $\sigma_c \cdot a$ is an accepting state (i.e., state in F_φ), then state q is updated ($q \leftarrow \delta_\varphi(q, \sigma_c \cdot a)$) , all the events of $\sigma_c \cdot a$ are released as output, and σ_c is emptied (set to ϵ). However, if $\delta_\varphi(q, \sigma_c \cdot a)$ is a dead state ($L(A_\varphi, \delta_\varphi(q, \sigma_c \cdot a)) = \emptyset$), then the received event is suppressed and the enforcer continues with the next event, otherwise it is buffered into σ_c (i.e., $\sigma_c \leftarrow \sigma_c \cdot a$), provided the buffer is not full. If the buffer is full, then function `clean` is invoked with $\sigma_c \cdot a$.

Function `clean` enters a loop where in every iteration i ($1 \leq i \leq k+1$), it checks if A_φ makes a cycle upon substrings of length i at every index j ($j = 1, ..$) of σ_{ca} (representing $\sigma_c \cdot a$). If a substring of length i from index j of σ_{ca} can be read on a cycle, then the subword formed by removing that substring from σ_{ca} is returned by function `clean` . For instance, in 10th row of Tab. 3, in Example 2, since the state reached (i.e., s_2) upon event a is not in F_φ, $L(A_\varphi, \delta_\varphi(s_1, cbabbaca)) \neq \emptyset$, and the buffer is full, thus, function `clean` is invoked with *cbabbaca*. In iteration $i = 1$, for every index j ($j = 1, ..$) of σ_{ca}, the substrings of length 1 are computed, which are: {[c], [b], [a], [b], [b], [a], [c], [a]}. Function `clean` checks if A_φ makes a cycle upon these substrings. Since by taking event c at index $j = 7$ of σ_{ca}, A_φ makes a cycle, thus the substring c is removed from σ_{ca} and *cbabbaa* (subword of $\sigma_c \cdot a$) is returned to the enforcer by function `clean`. Following it, the enforcer continues with the next event.

Remark 3. Note that function `clean` in Algorithm 1 produces the same output as function clean$^{\varphi,k}$ in Def. 4. Instead of providing σ_s (which corresponds to the output of the enforcement function in Def. 4) as input, we here provide the state q reached in $\mathcal{A}_\varphi$ upon the sequence released as output. Also the time complexity of function clean$^{\varphi,k}$ in Def. 4 is $\mathcal{O}(k^2)$, however, in the implementation, for efficiency reasons, function `clean` in Algorithm 1 directly computes the maximal subword of σ_{ca} by discarding the most obsolete elements that correspond to a (minimal)

cycle in $\mathcal{A}_\varphi$, which is chosen by function clean$^{\varphi,k}$ in Def. 4 from all the maximal candidate subwords.

5 Implementation and Evaluation

We implemented algorithms of Sect. 4.4 and work out some examples mainly *i*) to measure the performance [6] of the bounded-memory enforcer; how its performance varies compared to the ideal enforcer (mainly the additional overhead that will be induced by `clean`), by varying the complexity of the properties, input and buffer size and *ii*) to see the practicability and usefulness of the bounded-memory enforcer using example application scenarios. The bounded-memory enforcer and the ideal enforcer are implemented in 140 and 88 LoC respectively in Python. The implementation along with a brief description about the application scenarios, properties and the performance analysis using them is provided at https://github.com/saumyashankarsinha/BMRE.git.

5.1 Performance analysis

We take an example property P_1 which expresses, *"The word should start with one or more elements from set C={a, b, c} and should end with one or more elements from set D={1, 2}"*. P_1 is defined by the automaton A_{P_1} in Fig. 3, where state q_0 is the initial state and q_2 is the accepting state. This example has been chosen because it allows the automaton's size to be readily scaled up while maintaining its structure.

For measuring the performance, the size of buffer ($k = 4$) and number of states in A_{P_1} (=4) were fixed and the length of input sequence was varied from 10 to 10,000 with an increment of 1000 each time. The input sequences were chosen in such a way that the P_1 is satisfied by the latter events, so that the events are buffered and function `clean` is invoked everytime. It permits to calculate the worst time taken by the bounded-memory enforcer. The time taken by both the ideal enforcer (E^{P_1}) and the bounded-memory enforcer ($E^{P_1,4}$) was measured (all in seconds) and was averaged over 100 iterations. summarises the result where, `Input` indicates the length of the input word, `Time(s)` indicates the time taken by E^{P_1} to output the word, `No.` and `T1(s)` under `clean` indicates number of times function `clean` is called and the total time taken by it respectively (i.e., time taken to check if the buffer if full or not, time taken by function `clean` to delete events from the buffer and time taken to update the buffer with the word returned by function `clean`), and `T2(s)` indicates the time taken by $E^{P_1,4}$.
Considering $k = 4$, for A_{P_1}, we have the following observations from Tab. 4:

- The time taken by both unbounded and bounded enforcers increases linearly with the trace length (by considering traces such that the number of times function `clean` is invoked increases linearly).
- When comparing the time taken by the bounded and unbounded enforcer, if we subtract the additional time (T1) taken by the bounded enforcer to clean the buffer from the total time (T2) taken by it, then the resultant time (T2-T1) is similar to the time taken by the unbounded enforcer. Thus,

[6] Experiments were conducted on an Intel Core i7-9700K CPU at 3.60GHz × 8, with 32 GB RAM, and running on Ubuntu 18.04.5 LTS.

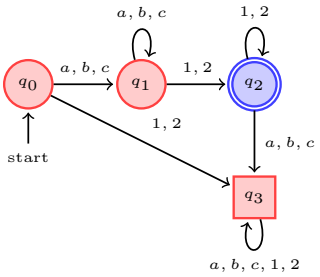

Fig. 3: A_{P_1}

Input	E^{P_1}	$E^{P_1,4}$			
	Time(s)	`clean`			
		No.	T1(s)	T2(s)	T2-T1(s)
10	0.00001	4	0.00002	0.00003	0.00001
1000	0.00094	994	0.00492	0.00586	0.00094
2000	0.00185	1994	0.00952	0.01136	0.00184
3000	0.00276	2994	0.01426	0.01701	0.00275
4000	0.00364	3994	0.01877	0.02239	0.00362
5000	0.00457	4994	0.02350	0.02806	0.00456
6000	0.00548	5994	0.02814	0.03360	0.00546
7000	0.00642	6994	0.03314	0.03956	0.00642
8000	0.00736	7994	0.03784	0.04518	0.00734
9000	0.00827	8994	0.04247	0.05068	0.00821
10000	0.00922	9994	0.04754	0.05679	0.00924

Table 4: Effect on time taken by enforcers by varying the length of input sequences.

when cleaning the buffer is not necessary (e.g, for some input traces/when the buffer size is large), the performance of the bounded enforcer is similar to the unbounded enforcer.
- The average time taken for cleaning (per call) is 0.0048 ms is low/reasonable.

Varying the complexity of the property. We also considered properties with similar structures as P_1 but with an increasing number of states, and evaluated them with input sequences of varying sizes. It was observed that the time taken by both the enforcers increases linearly with the number of states (as the number of transitions increase).

Varying the buffer size. On increasing the buffer size, the average time taken by function `clean` increases linearly, however the time taken by $E^{\varphi,k}$ decreases linearly, since the number of times function `clean` is called decreases.
These are further discussed at *varying_complexity_of_property_and_buffer_sizes*.

Remark 4. We choose to clean the buffer when it is full. Indeed we consider that the computing time of the cleaning function is lower than the time between two received events. One may clean the buffer when it is partially full. This can be done with a simple modification in the framework by considering another additional parameter as input indicating when function `clean` should be triggered (when the buffer is 80% full, 90% full etc).

Remark 5 (Enforcing multiple properties). Our framework is able to consider multiple properties. It consists of computing their intersection and then synthesize the corresponding enforcer.

6 Potential Application Scenarios

We discuss the usefulness and applicability of the bounded-memory enforcer in real-world context. We formulate some example properties in different scenarios to express the desired behaviour in domains like autonomous vehicle, operating system, databases, etc. The formulated properties are such that it deals with the

suppression of the *idempotent* events when required, making them suitable to be enforced by a bounded-memory runtime enforcer. We also analyse the performance of the bounded-memory enforcer against these real-world properties.

Scenario 1. Autonomous Vehicle (AV): An AV or a self-driving car is a vehicle that is capable of sensing its environment and moving safely with little or no human intervention. They rely on sensors, actuators, complex algorithms, machine learning systems, and powerful processors to execute software. Let us consider two example properties in AVs to express some desired behaviour.

(a) Logging in AV: *"When the path planning steering commands, like {Move Left, Move Right, Move Forward, Stop} are logged for better testing and validation solutions, each time it is issued, to a remote location (due to memory constraints in AV), then logging of these commands should be done, when the vehicle reaches a Stop state, on the remote logging application."*

(b) Switching to manual driving mode in autonomous vehicle: *"Upon pressing of the manual mode button, the switching of manual driving mode from autonomous driving mode will be done if the following three conditions are satisfied: checking whether a driver's hand is holding a steering wheel; checking whether the driver's foot is placed on a brake pedal; checking whether the driver's gaze is facing forward."* It is here assumed that once an event is received, meaning that the condition respective to that event is satisfied, it remains satisfied.

Scenario 2. Concurrency: In the context of concurrent systems, each hardware/software component is designed to obey or to meet certain consistency rules. Concurrent use of shared resources can be a source of indeterminacy leading to issues such as deadlocks, and resource starvation. Let us consider two example properties related to concurrency.

(a) Lock: *"For database items {A, B}, any transaction accessing both A and B must access A before accessing B."*

(b) Critical Section Problem: "*If a process wishes to enter the critical section, it must first execute the try section and wait until it acquires access to the critical section. After the process has executed its critical section and is finished with the shared resources, it can release them for other processes' use.*"

Remark 6. We modeled all the policies for each of the scenarios as DFA, obtained the respective bounded-memory enforcers and have measured their performance. More details, including description about the scenarios, policies and their formulation as DFA, and the performance analysis using them is provided along with the implementation in our repository at: *Potential_Application_Scenarios*.

7 Related Work

RE has been pioneered by Schneider et al. in [21]. In this framework the enforcement mechanism enforces properties described as Büchi automata and synthesizes a *security automaton (SA)* which is executed in parallel with the system under scrutiny and terminates whenever the property is going to be violated. Later, Bloem et al. propose in [3] an approach to synthesize an enforcement mechanism (named *shield*) using a 2-player game approach. A shield is *k-stabilizing*

i.e., whenever a property violation is unavoidable, it allows deviating from the property for k consecutive steps. A similar approach is proposed by Wu et al. in [23] adding the ability to handle burst errors and Bielova et al. in [2] who specified how bad traces are fixed so that the system exhibits a reasonable behaviour. All these approaches deal only with safety properties and cannot memorize actions.

In [12], Ligatti et al. extend the work of Schneider et al. by noticing that SA (only) act as a sequence recognizer, thus they propose the model of *edit-automata (EA)*: this models allows inserting or suppressing events during the RE and to use a memory in order to store the suffix of an invalid execution until it becomes valid. EA permits enforcing a larger set of properties, namely the *renewal* properties. Later, Falcone et al. generalise in [9] EA with the *generalised enforcement monitors*. Their model enforces *response* properties (from the *Safety-Progress* classification [5]), which are similar to *infinite renewal* properties, but separates explicitly the specification of the property to enforce from the enforcer. Many extensions of these approaches exist, e.g. in a timed context [7,14,13,16] or considering uncontrollable events [18,17,19].

The above models do not consider memory constraints. Few attempts with memory limitations of the enforcer have been proposed. Fong proposes in [10] the model of *Shallow History Automata (SHA)* as SA that do not keep track of the order of event occurring, and generalizes it as α-SA, a variant of SA endowed with a morphism abstracting the current input sequence. Then he defines a complete lattice of security policy classes. Talhi et al. [22] extend SA and EA to define their bounded-history versions, that is the versions of the mechanism that can remember up to a certain number of events in the trace. Beauquier et al. [1] consider finite set of states as a memory limitation on EAs and show that finite EAs are strictly less expressive than EAs and characterize the conditions of the enforceability of a property. These approaches, considering a limited memory, mainly focus on characterizing the set of enforceable properties. To the best of our knowledge, our framework is the first to define how to synthesize an enforecr that provides a solution when the memory of the enforcer is full at runtime.

8 Conclusion and Future Work

This paper presents a complete RE framework for regular properties with a bounded memory. In this approach, the enforecr has the ability to delay (buffer) or suppress events, and the maximal size of the memory is known. We introduce the notion of nominal and degraded modes, the last one corresponding to the situation where the maximal size of the memory has been reached. We redefine the notion of transparency and propose a way to reduce optimally the content of the memory in order to maintain a behaviour satisfying the property. We provide a functional definition of the enforcer and an algorithmic version. We have implemented the framework in a prototype and evaluated its performance using multiple example properties.

Future works include enriching the framework in order to consider the possibility to recover a nominal behaviour when the degraded one has been reached, and to extend the proposed approach in a timed context.

References

1. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite and pushdown edit automata. Int. J. Inf. Sec. **12**(4), 319–336 (2013). https://doi.org/10.1007/s10207-013-0195-8, http://dx.doi.org/10.1007/s10207-013-0195-8
2. Bielova, N., Massacci, F.: Predictability of enforcement. In: Proceedings of the Third International Conference on Engineering Secure Software and Systems. p. 73–86. ESSoS'11, Springer-Verlag, Berlin, Heidelberg (2011)
3. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: Runtime enforcement for reactive systems. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 533–548. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
4. Dolzhenko, E., Ligatti, J., Reddy, S.: Modeling runtime enforcement with mandatory results automata. Int. J. Inf. Secur. **14**(1), 47–60 (Feb 2015). https://doi.org/10.1007/s10207-014-0239-8, https://doi.org/10.1007/s10207-014-0239-8
5. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: Runtime Verification. pp. 40–59. Springer (2009)
6. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transf. **14**(3), 349–382 (2012). https://doi.org/10.1007/s10009-011-0196-8, https://doi.org/10.1007/s10009-011-0196-8
7. 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). https://doi.org/10.1016/j.scico.2016.02.008
8. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Lectures on Runtime Verification - Introductory and Advanced Topics, pp. 103–134 (2018). https://doi.org/10.1007/978-3-319-75632-5_4, https://doi.org/10.1007/978-3-319-75632-5_4
9. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods Syst. Des. **38**(3), 223–262 (2011). https://doi.org/10.1007/s10703-011-0114-4, https://doi.org/10.1007/s10703-011-0114-4
10. Fong, P.W.L.: Access control by tracking shallow execution history. In: IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004. pp. 43–55 (2004). https://doi.org/10.1109/SECPRI.2004.1301314
11. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Sec. **4**(1-2), 2–16 (2005). https://doi.org/10.1007/s10207-004-0046-8, https://doi.org/10.1007/s10207-004-0046-8
12. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. **12**(3) (Jan 2009). https://doi.org/10.1145/1455526.1455532, https://doi.org/10.1145/1455526.1455532
13. 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). https://doi.org/10.1007/s10703-014-0215-y
14. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.L.: Runtime enforcement of timed properties. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey,

September 25-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7687, pp. 229–244. Springer (2012). https://doi.org/10.1007/978-3-642-35632-2_23
15. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Formal Methods Syst. Des. **51**(1), 154–199 (2017). https://doi.org/10.1007/s10703-017-0271-1, `https://doi.org/10.1007/s10703-017-0271-1`
16. Pinisetty, S., Roop, P.S., Smyth, S., Tripakis, S., Hanxleden, R.v.: Runtime enforcement of reactive systems using synchronous enforcers. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. pp. 80–89 (2017)
17. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal enforcement of (timed) properties with uncontrollable events. Mathematical Structures in Computer Science pp. 1–46 (2017). https://doi.org/10.1017/S0960129517000123
18. Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H.: Enforcement of (timed) properties with uncontrollable events. In: Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. pp. 542–560 (2015). https://doi.org/10.1007/978-3-319-25150-9_31
19. Renard, M., Rollet, A., Falcone, Y.: Runtime enforcement of timed properties using games. Formal Aspects of Computing **32**(2), 315–360 (2020)
20. Roc su, G.: On safety properties and their monitoring. Scientific Annals of Computer Science **22**(2), 327–365 (2012). https://doi.org/10.7561/SACS.2012.2.327
21. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (Feb 2000). https://doi.org/10.1145/353323.353382
22. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Information and Computation **206**(2), 158–184 (2008). https://doi.org/https://doi.org/10.1016/j.ic.2007.07.009, `https://www.sciencedirect.com/science/article/pii/S0890540107001320`, joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA '06)
23. Wu, M., Zeng, H., Wang, C.: Synthesizing runtime enforcer of safety properties under burst error. In: NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. pp. 65–81 (2016). https://doi.org/10.1007/978-3-319-40648-0_6