JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Bounded Serial Compositional Runtime Enforcement

Saumya Shankar[*], **Thierry Jéron**[†], **Prisha Srinidi**[*], **and Srinivas Pinisetty**[*]
[*]Indian Institute of Technology Bhubaneswar, Bhubaneswar, India
[†]Univ Rennes, Inria, IRISA, Rennes, France

ABSTRACT Runtime enforcement is a monitoring technique used to ensure that a system behaves according to a set of formal properties. It employs an enforcer that transforms an untrusted sequence of events into one that satisfies the specified property. In cases of property violation, we allow the enforcer to temporarily delay input events (i.e., storing them in memory until the property can be satisfied) or to suppress them if the property can never be satisfied. This paper addresses the enforcement of multiple (untimed) properties modelled as finite automata, where their enforcers are composed serially, to promote modularity, i.e., rather than synthesizing a single enforcer for all properties, we generate individual enforcers and combine them serially. Also, to handle practical constraints, the paper considers that each enforcer operates with bounded (finite) memory.
We explore whether regular properties and their subclasses, specifically safety ("nothing bad happens") and co-safety ("something good eventually happens") can be enforced in the compositional setting under memory constraints. We define enforceability in terms of preserving key criteria like soundness, transparency. For properties that are not inherently serially enforceable, we identify specific conditions on their automata whose satisfaction allows certain groups of properties to become serially enforceable. To formalize and support these ideas, we present the Bounded Serial Compositional Runtime Enforcement framework. We provide a prototype implementation and evaluate the performance of the proposed serial enforcer to demonstrate practical feasibility.

KEYWORDS Formal verification, Runtime enforcement, Bounded-memory, Serial composition, Regular property, Safety and co-safety property, Automata

1. Introduction

Runtime Enforcement (RE) (Schneider 2000; Ligatti et al. 2005; Pinisetty, Preoteasa, et al. 2017; Falcone et al. 2016) is a technique to monitor the execution of a system at runtime and ensure its compliance with a set of formal requirements. For this, it employs an enforcement monitor (or an enforcer) which is generally synthesized from a property expressed in a high-level formalism. The primary function of this enforcer is to process input sequences from the system, which may be untrustworthy or non-compliant, and transform them into output sequences that satisfy the specified requirements. The effectiveness of an enforcer is underpinned by two fundamental principles: *soundness* and *transparency*. *Soundness* signifies that the transformation carried out by the enforcer should always ensure that the resulting output sequence aligns with the specified property. *Transparency* signifies that the enforcer should leave unaltered any correct input sequences that already satisfy the specified properties.

There are different enforcement frameworks that differ from each other in the supported enforcement operations, the supported classes of properties for which enforcers can be generated, etc. For example, regarding enforcement operations, in the case of violation, (Schneider 2000) permits blocking the execution, (Ligatti et al. 2005) permits correcting the input sequence by suppressing and inserting events, (Pinisetty, Preoteasa, et al. 2017; Shankar et al. 2022, 2024) allows temporarily delaying the events of the input sequence (i.e. storing the events in memory of the enforcer) and later releasing it when the property is satisfied, (Pinisetty, Roop, et al. 2017) does not permit delaying and considers instantaneous replacement of the events, etc.

JOT reference format:
Saumya Shankar, Thierry Jéron, Prisha Srinidi, and Srinivas Pinisetty. *Bounded Serial Compositional Runtime Enforcement*. Journal of Object Technology. Vol. 25, No. 1, 2026. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0)
http://dx.doi.org/10.5381/jot.2026.25.1.a11

Related work. In this work, we consider frameworks such as (Shankar et al. 2022, 2024) which consider *delaying* and *suppression* as key enforcement mechanisms employed by the enforcer in the event of violation. Specifically: *Delaying*: When incoming events from the input sequence do not currently satisfy the specified property but have the potential to do so with the arrival of future events, the enforcer withholds (i.e., delays) their release and stores them in memory (of the enforcer). Once a future event arrives that, together with the stored events, satisfies the property, the enforcer emits both the stored and the newly received event. *Suppression*: If certain events in the input sequence render it impossible to satisfy the property in the future, (such as when the system reaches a dead state from which no accepting state can be reached regardless of subsequent inputs) then those events are permanently suppressed.

In many application domains, enforcement mechanisms such as delaying and suppression are sufficient. Delaying is the simplest approach: if a property is not currently satisfied, the input is held until it becomes satisfied, resulting in minimal intervention in the input sequence. If a property can never be satisfied, suppression is applied to remove the offending input. Other mechanisms, such as editing the input sequence, require more intrusive modifications. In contrast, delaying and suppression generally incur lower computational overhead.

Delaying events inherently requires *memory* to store them temporarily. For this, the enforcer is allocated memory, referred to as a *buffer* in (Shankar et al. 2022, 2024). This buffer is considered bounded in the work, meaning it has limited capacity. This assumption is realistic and practical, especially in embedded systems applications where memory resources are inherently constrained (Talhi et al. 2008).

Now, these bounded buffers introduce interesting design challenges, for example, a critical situation may arise when a new event, which currently does not satisfy the property but may do so in the future, needs to be stored, yet the buffer is already full. To address this, (Shankar et al. 2022, 2024) proposes an RE paradigm that efficiently *cleans* the buffer to accommodate new events. This cleaning process involves suppressing some stored events to free up space. Importantly, this suppression is not done arbitrarily; instead, it is performed in a way that ensures the satisfaction or violation status of future events is preserved.

The overall enforcement mechanism follows these principles in the event of violation:

- Suppress an event if it does not currently satisfy the property and can never do so in the future.
- Buffer an event if it does not currently satisfy the property but may satisfy it later, provided there is space in the buffer.
- If the buffer is full, selectively suppress buffered events to create space for incoming ones (thus, handling memory constraint).
  This suppression is performed strategically: events involved in looping transitions within the buffered word are chosen for removal. These events are typically idempotent/repetitive, thus, their suppression often has little to no impact on the overall system behaviour or output. Furthermore, this suppression strategy is sound (see Def. 5 of (Shankar et al. 2022)). Moreover, since the buffer size is chosen to be greater than or equal to the number of states in the automaton representing the property, we are guaranteed to encounter such looping events, because any path without cycles between two states cannot exceed the total number of states in length.

Hence, suppression occurs in two scenarios: 1) when the system reaches a dead state, and 2) when the buffer is full.

Monolithic and compositional approaches to enforce multiple properties The enforcement frameworks cited above primarily concentrate on enforcing a single property. They synthesize a single enforcer from the given properties which corrects the executions. In case of multiple properties to enforce, one combines all properties by taking their conjunction, and then synthesize a single enforcer for the conjunction property. This is called the monolithic approach.

Other than the monolithic approach, the compositional approach (Pinisetty & Tripakis 2016; Pinisetty et al. 2022) of enforcing multiple properties offers certain advantages. A compositional approach means synthesizing individual enforcers for each property and then finding a way to compose them in a more flexible and modular manner. (Pinisetty & Tripakis 2016; Pinisetty et al. 2022) suggest serial ($\Rrightarrow$) and parallel ($\|$) composition schemes. Serial where the output of one enforcer serves as the input to the next/succeeding enforcer in a sequential manner, and parallel composition where multiple enforcers run concurrently, receiving the same input and their individual outputs are then merged using specific methods.

We have the following observations of our interest from (Pinisetty & Tripakis 2016; Pinisetty et al. 2022):

- In general, regular properties cannot be combined through either serial or parallel composition schemes. This means that if we have two enforcers, say, $enforcer_1$ and $enforcer_2$ for two properties, respectively, then, $enforcer_1 \Rrightarrow enforcer_2$, $enforcer_2 \Rrightarrow enforcer_1$ and $enforcer_1 \| enforcer_2$ generally do not enforce (satisfy) the corresponding properties.
- Indeed, composition is only viable for particular subclasses of regular properties (safety and co-safety properties), and even then, it is limited to properties within the same subclass, like when both the properties are safety (or co-safety) properties.
  - The order in which enforcers are combined (whether through serial or parallel composition) does not affect the outcome.
  - The enforcers created through composition will produce the same outcomes as a monolithic enforcer.
  - Given two properties, if one of them is identified to be a safety property, the serial composition approach is applicable if that safety property is considered as the first (upstream) property in the serial composition, while the other property can be any regular property and positioned as the second (downstream) property.

Frameworks in (Pinisetty & Tripakis 2016; Pinisetty et al. 2022) consider delaying as one of the enforcement mechanisms.

However, it considers infinite memory available at hand for each enforcer. In this work, we want to investigate if the compositional schemes will work if we now consider finite memory allocated to the enforcers of composition. This is important because allocating infinite memory to each of the enforcement monitors is practically infeasible.

Contributions. We explore the serial[1] compositionality of regular properties and subclasses of them (safety and co-safety properties) under memory constraints (bounded buffers) for each of the enforcers of the corresponding properties. In the process, we investigate the following: are the properties serially enforceable? By enforceable we mean that the final output obtained from the compositional approach is respecting criteria of a suitable enforcer such as soundness, transparency. Our findings reveal that the sub-classes of regular properties that were serially enforceable are not anymore enforceable when enforcers have limited memory. However, for certain subclasses, we identify sufficient conditions on their automata, which, when met, allow these properties to be enforced in a serial manner. We call this framework a *Bounded Serial Compositional Runtime Enforcement* framework.

To support our theoretical results, we provide counterexamples illustrating cases of non-enforceability and formal proofs for enforceable instances under the proposed conditions. We also provide a prototype implementation of the proposed framework to assess the performance of the serial enforcer in comparison to the monolithic enforcer, demonstrating their practical viability. To evaluate scalability, we conducted experiments by varying the length of the input trace and measuring the execution time across both the enforcement strategies, monolithic and serial. Additionally, we varied the number of properties to be enforced while keeping the input size fixed, to observe how enforcement time scales with increasing property count.

Outline. Section 2.1 introduces the preliminaries and notations. Section 2.2 provides an overview of the bounded-memory runtime enforcement framework proposed in (Shankar et al. 2022). Section 2.3 reviews the definitions from (Pinisetty & Tripakis 2016) concerning compositional runtime enforcement. The following sections outline the main contributions of this paper. Section 3 investigates the enforceability of composite properties. Section 4 provides the conditions for enforceability of composite properties. Section 5 provides a case study demonstrating how the proposed enforcement framework can be effectively applied and also presents evaluation of the serial enforcer on the case study. Finally, Section 6 concludes the paper.

The following section presents the background required to understand the work.

2. Background

2.1. Preliminaries and Notations

Languages. A (finite) word w over a finite alphabet Σ is a finite sequence of events/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 φ over Σ is any subset L of Σ^*.

Prefix-closed and extension-closed languages. 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 $\mathrm{pref}(w)$ denotes the *set of prefixes* of w and subsequently, $\mathrm{pref}(L) \stackrel{\text{def}}{=} \bigcup_{w \in L} \mathrm{pref}(w)$ is the set of prefixes of words in L.

A language $L \subset \Sigma^*$ is said to be prefix-closed if, whenever a word w belongs to L, every prefix of w also belongs to L. A language $L \subset \Sigma^*$ is said to be extension-closed if, whenever a word w belongs to L, every word that extends w (i.e., every string having w as a prefix) also belongs to L.

Concatenation. The concatenation of two words w and w' is denoted by $w \cdot w'$.

Subword. 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, if $w = w_0 a_1 w_1 ... a_n w_n$ for some $w_0, ..., w_n \in \Sigma^*$.

Deterministic and complete finite automata. A deterministic and complete finite automaton $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ where,

- Q is the set of locations (also called states),
- $q_0 \in Q$ is the initial location,
- Σ is the finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is the (total) transition function[2] and
- $F \subseteq Q$ is the set of accepting locations.

Languages of automata: The language of $\mathcal{A}$ starting from initial state q_0 is denoted $L(\mathcal{A}, q_0)$ (or simply $L(\mathcal{A})$) and is the set of all accepted words from q_0 : $L(\mathcal{A}, q_0) = \{\sigma \in \Sigma^* \mid \delta^*(q_0, \sigma) \in F\}$.

Product of automata. Let $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ and $\mathcal{A}' = (Q', q_0', \Sigma, \delta', F')$ be two automata over the same alphabet Σ. The product of $\mathcal{A}$ and $\mathcal{A}'$, denoted $\mathcal{A} \times \mathcal{A}'$, is defined as $(Q \times Q', (q_0, q_0'), \Sigma, \delta \times \delta', F \times F')$, where $\delta \times \delta' = ((q, q'), a) = (\delta(q, a), \delta'(q', a))$ where $q \in Q$ and $q' \in Q'$. We have $L(\mathcal{A} \times \mathcal{A}') = L(\mathcal{A}) \cap L(\mathcal{A}')$.

Let φ_1 and φ_2 be two properties, modelled by automata $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$ respectively and let $L(\mathcal{A}_{\varphi_1})$ and $L(\mathcal{A}_{\varphi_2})$ denote their languages. Then, the language of the conjunction $\varphi_1 \wedge \varphi_2$ is the intersection of languages $L(\mathcal{A}_{\varphi_1}) \cap L(\mathcal{A}_{\varphi_2})$ and is specified by the product automaton.

A word w satisfying φ is denoted by $w \models \varphi$, meaning w belongs to the language accepted by the automaton defining φ.

Coreach and reach. The coreachability function (often denoted as coreach) identifies the set of states from which a given set of states is reachable (in zero or more steps). The reachability

[1] There could be other ways of composing enforcers such as the parallel composition scheme (Pinisetty & Tripakis 2016). In the current work we specifically focus on the serial composition approach. We intend to look into and explore the parallel scheme in a future work.

[2] We also have an extended transition function δ^* denoted as $\delta^* : Q \times \Sigma^* \to Q$. It takes a state q and a string w and returns a state q'– the state that the automaton reaches when starting in state q and processing the sequence of inputs w.

function (often denoted as reach) identifies all states that are reachable from the given set of states.

Mathematically, given an automaton $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$, let $S \subseteq Q$ be the given set of states. Then, $coreach_{\mathcal{A}}(S)$ and $reach_{\mathcal{A}}(S)$ are defined as:

$$coreach_{\mathcal{A}}(S) = \{q \in Q \mid \exists\, w \in \Sigma^*,\, \delta^*(q,w) \in S\}$$

$$reach_{\mathcal{A}}(S) = \{q \in Q \mid \exists s \in S, \exists\, w \in \Sigma^*,\, \delta^*(s,w) = q\}$$

A dead state is a state from where there is no way for the automaton to reach an accepting state. Formally, a state q is considered dead if $q \notin coreach(F)$.

Classification of properties. A regular property is a language accepted by a finite automaton. Safety (resp. co-safety) properties are sub-classes of regular properties. Safety properties are the prefix-closed regular languages. Co-safety properties are the extension-closed regular languages. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time").

Safety and co-safety automata. The set of safety and co-safety properties can be characterized by safety and co-safety automata. For the safety automaton, no accepting state can be reached from a non-accepting one, i.e., F does not intersect $reach(Q \setminus F)$. For the co-safety automaton, no non-accepting state can be reached from an accepting one, i.e., $Q \setminus F$ does not intersect $reach(F)$.

It is easy to see that the product of two safety (resp. co-safety) automata is a safety (resp. co-safety) automaton.

Example 1 *(Regular, safety and co-safety). Consider the following properties on the alphabet $\Sigma = \{a, b, c\}$:*

- *RE_1: Traces should start with 'a' and end with 'b'.*
- *RE_2: Traces should start with an 'a' and either contain any number of 'a', or be followed by at least two events from {'b','c'} without any additional 'a' afterwards.*
- *S_1: Traces can have at most 2 'a' and after the first 'a', it is forbidden to have a 'b'.*
- *S_2: Traces should be such that after the first 'a', it is forbidden to have another 'a' and when followed by a 'b', it is forbidden to have a 'c' and when followed by an 'a', it is forbidden to have a 'c'.*
- *S_3: Traces should be such that after the first 'a', when followed by an 'a' or a 'c', it is forbidden to have a 'b' and when followed by a 'c', it is forbidden to have a 'c' or an 'a'.*
- *CS_1: Traces should start with an 'a' or a 'c', must contain at least 1 'b' and no 'c' should occur before the first 'b'.*
- *CS_2: Traces should start with an 'a' or a 'c' and have at least 1 'c' followed by a 'b'.*
- *CS_3: Traces should start with an 'a' or a 'c' and have at least one 'b'.*

The set of actions $\Sigma = \{a, b, c\}$. Properties RE_1 and RE_2 are regular properties and are defined by finite automata in Fig. 1 and 2 respectively. Properties S_1, S_2 and S_3 are safety properties defined by safety automata in Fig. 3, 4 and 5 respectively. Properties CS_1, CS_2 and CS_3 are co-safety properties defined by co-safety automata in Fig. 6, 7 and 8 respectively.

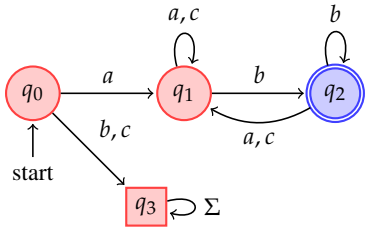

Figure 1 $\mathcal{A}_{RE_1}$

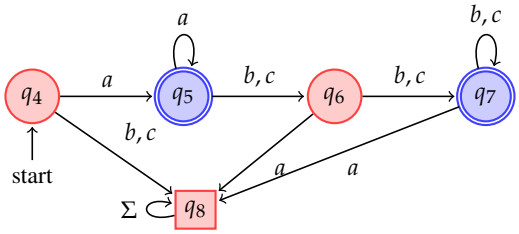

Figure 2 $\mathcal{A}_{RE_2}$

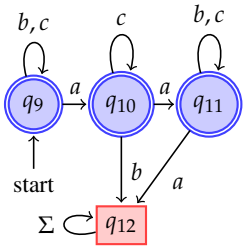

Figure 3 $\mathcal{A}_{S_1}$

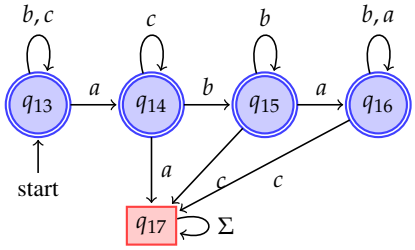

Figure 4 $\mathcal{A}_{S_2}$

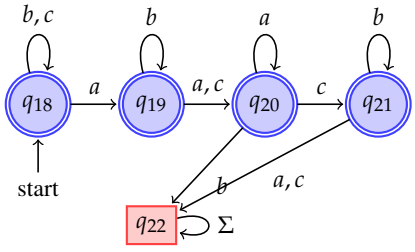

Figure 5 $\mathcal{A}_{S_3}$

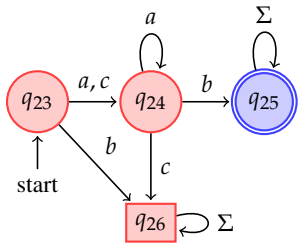

Figure 6 $\mathcal{A}_{CS_1}$

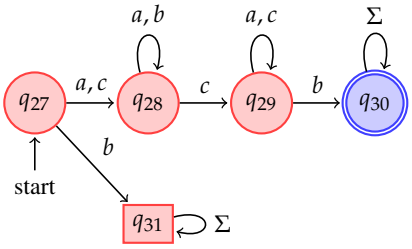

Figure 7 $\mathcal{A}_{CS_2}$

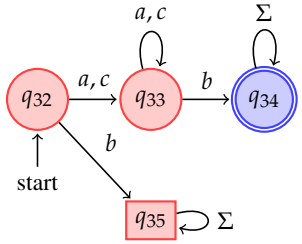

Figure 8 $\mathcal{A}_{CS_3}$

2.2. Bounded-memory runtime enforcement

Several runtime enforcement frameworks exist. Some are already mentioned in the introduction. In this paper, we follow the framework of (Shankar et al. 2022), an enforcement monitor is synthesized from a regular property φ modelled as an automaton $\mathcal{A}_\varphi$ and the buffer size is limited to k.

The input-output behaviour of an enforcement monitor is specified by an enforcement function. The enforcement function $E^{\varphi,k}$ transforms some input word σ which is possibly incorrect w.r.t. φ. The enforcement mechanism has the ability of buffering and suppressing events when a violation is detected. Thus, the output $E^{\varphi,k}(\sigma)$ is a subword of the input word σ.

Some requirements are defined on the enforcement function (Def. 3 of (Shankar et al. 2022)) such as *soundness* and *transparency* as well as *monotonicity*, *optimal suppression*. These are detailed below in Definition 1.

Definition 1 (Constraints on the Bounded Enforcer) *A bounded enforcer for φ is a function, $E^{\varphi,k} : \Sigma^* \to \Sigma^* \times \{\top, \bot\}$. $E^{\varphi,k}$ transforms some input word $\sigma \in \Sigma^*$ which is possibly incorrect w.r.t. φ into output $o \in \Sigma^*$, with the buffer size allocated as k. It outputs o (given by $E^{\varphi,k}_{\text{out}}(\sigma)$) and the mode information $\{\top, \bot\}$ (given by $E^{\varphi,k}_{\text{mode}}(\sigma)$) which tells if some events are suppressed by the enforcer (mode=$\bot$) or not (mode=$\top$). $E^{\varphi,k}$ satisfies the following constraints:*

Soundness:

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

Transparency:

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

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

Monotonicity:

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

Optimal Suppression:

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

Soundness **SndB** means that for any input word σ, if the output is not empty ($\neq \epsilon$), then it must satisfy φ.

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).

Monotonicity **MoB** 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., $E^{\varphi,k}$ is a growing function over relation $\preccurlyeq$. This means that whatever has been released by the enforcer as output cannot be undone. It can only be changed by appending more events to it.

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.

Let us now see, at a high level, the definition of an enforcement function Def. 4 of (Shankar et al. 2022) that incrementally constructs the output while ensuring that the constraints are satisfied; the corresponding enforcers are unique for a given property, constraints, and enforcement mechanism.

The function operates based on four key actions:

– If the enforcer receives an event that does not currently satisfy the property, but the property could still be satisfied in the future (i.e., the current state reached is not accepting,

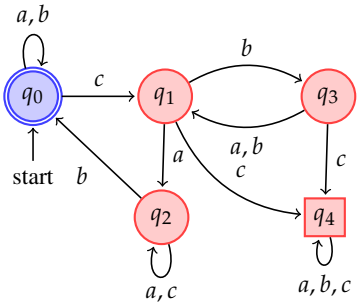

Figure 9 $\mathcal{A}_\varphi$

but there exists a path to an accepting state), it does not release the event immediately. Instead, it stores the event in its buffer, provided there is available space in the buffer.
- If the enforcer receives an event that does not currently satisfy the property, but the property could still be satisfied in the future, and the buffer is full, then it invokes cleaning of the buffer. Cleaning suppresses the minimum number of events from the buffer, including possibly the received event itself, s.t. the treatment of future events by the enforcer is not affected (i.e., the enforcer handles the future events the same way with or without cleaning of the buffer). This is guaranteed if the state reached in the property automaton by the uncleaned as well as cleaned buffer (with some events removed from the buffer) contents remain the same. Thus, the events that were engaged in looping transitions in the automata (the so-called idempotent/repetitive events) are chosen as the candidates for suppression.
- If the enforcer receives an event that does satisfy the property (i.e., it leads to an accepting state), it releases this event along with all previously buffered events. The buffer is then emptied.
- If the enforcer receives an event that neither satisfies the property currently nor can lead to satisfaction in the future (i.e., the current state is a dead state with no path to an accepting state), the event is suppressed– it is neither released nor stored in the buffer.

Example 2 (Enforcement function.) *To illustrate the enforcement function, let us consider the property φ in Fig. 9 (from Example 1 of (Shankar et al. 2022)) and the input sequence acbabbcacab. Suppose the max size of the buffer is 7.*

The behaviour of the enforcement function is given in Table 1 (adapted from (Shankar et al. 2022)) and is as follows:

- *When the enforcer receives the first event a, φ is satisfied, 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 Table 1.*
- *Upon receiving the second event c, φ is not satisfied as the state reached is q_1 which is a non-accepting state. But there exists a path to an accepting state q_0. Thus, the event c is added into the (empty) buffer.*
- *The events in rows 3-6 are also appended to the buffer, as the input events do not satisfy φ, but can satisfy φ in the future.*
- *When event c is received (row 7), neither φ is satisfied currently nor can it be satisfied in the future 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 are also appended to the buffer, as the input events does not satisfy φ, but can satisfy φ in the future.*
- *In the 10th row, when event a is received, φ is not satisfied, but can be satisfied in the future. Thus, it can be appended to the buffer; however, the buffer is already full. So, cleaning of the buffer is invoked to accommodate the received event.*
  The possible candidate subsequences resulting after cleaning the buffer (i.e., cbabbaca) are: "cbbaca","cbaaca","cbabbac","cbabbaa","caca","cbabba" which are obtained after suppressing events 'ba','bb', 'a', 'c', 'babb', 'aca' respectively from "cbabbaca". The set of longest subsequences chosen are "cbabbac" and "cbabbaa".
  Finally "cbabbaa" is chosen which is formed after removing the most obsolete (cyclic) event. Thus, the content of buffer is replaced by "cbabbaa" in the 10th row.
- *Finally in the 11th row, event b is received, which satisfies the property (i.e., it leads to an accepting state q_0). So, this event along with all previously buffered events are released as output, leaving the buffer empty.*

Table 1 Incremental computation by the bounded enforcement function.

	Input	State	Buffer	Output
1	*a*	q_0	ε	*a*
2	*ac*	q_1	*c*	*a*
3	*acb*	q_3	*cb*	*a*
4	*acba*	q_1	*cba*	*a*
5	*acbab*	q_3	*cbab*	*a*
6	*acbabb*	q_1	*cbabb*	*a*
7	*acbabbc*	~~q_4~~ $q1$	*cbabb*	*a*
8	*acbabbca*	q_2	*cbabba*	*a*
9	*acbabbcac*	q_2	*cbabbac*	*a*
10	*acbabbcaca*	q_2	*cbabba*~~*a*~~	*a*
11	*acbabbcacab*	q_0	ε	*acbabbaab*

2.3. Serial compositional runtime enforcement

In a composite system, there are several individual enforcers, one for each property. All individual enforcers have their own output buffers and are connected according to the scheme of compositionality applied, such as serial or parallel. In this sec-

tion, we recall the serial approach of composition of enforcers from (Pinisetty & Tripakis 2016; Pinisetty et al. 2022) (the internal memory for the purpose of delaying events is considered unbounded here).

Definition 2 *(Serial composition of enforcers.) Given n properties $\varphi_1, \varphi_2, \cdots, \varphi_n$, we can synthesize enforcers $E^{\varphi_1}, E^{\varphi_2}, \cdots, E^{\varphi_n}$ with unlimited buffers for each of them and compose them in a specific order using serial composition, denoted as $E^{\varphi_1} \Rrightarrow E^{\varphi_2} \Rrightarrow \cdots \Rrightarrow E^{\varphi_n}$. This notation represents a pipeline where the output of E^{φ_1} becomes the input to E^{φ_2}, and so on. Accordingly, for an input sequence σ, the output of this serial composition is defined as: $(E^{\varphi_1} \Rrightarrow E^{\varphi_2} \Rrightarrow \cdots \Rrightarrow E^{\varphi_n})(\sigma) = E^{\varphi_n}(\cdots E^{\varphi_2}(E^{\varphi_1}(\sigma)))$.*

Definition 3 *(Enforceability using the serial composition approach.) Given n properties $\varphi_1, \varphi_2, \cdots, \varphi_n$ in a system, these properties are enforceable using the serial composition of enforcers $E^{\varphi_1} \Rrightarrow E^{\varphi_2} \Rrightarrow \cdots \Rrightarrow E^{\varphi_n}$ (from Definition 2) iff $E^{\varphi_1} \Rrightarrow E^{\varphi_2} \Rrightarrow \cdots \Rrightarrow E^{\varphi_n}$ is an enforcer w.r.t. $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$ as per the constraints: soundness, transparency and monotonicity mentioned in (Pinisetty & Tripakis 2016).*

In general, regular properties are not enforceable via serial composition with unlimited buffers, except for subclasses safety or co-safety properties.

The following sections present the main contributions of this paper. First, in the next section, we evaluate the enforceability of composite properties by considering a bound on the buffer associated with each property.

3. Evaluating enforceability of composite properties

Here we investigate the enforceability of various combinations of safety, co-safety, and regular properties using the serial compositional approach. For each combination, we formally specify the properties, construct their corresponding automata, synthesize individual *bounded* enforcers, and derive serial enforcers. Our analysis focuses on the following question: Under the constraint of bounded-memory for all enforcers, is the given combination of properties enforceable?

We perform this analysis for pairs of properties, but the approach generalizes to combinations involving any number of properties. This is because the combination (i.e., intersection) of two regular properties (or respectively safety or co-safety properties) is itself a regular property (or respectively a safety or co-safety property).

We fix the buffer size for the individual enforcers as well as the monolithic enforcer as k which is the maximum of the number of states in the property automata.

As shown in (Pinisetty & Tripakis 2016; Pinisetty et al. 2022), two regular properties, a regular and a co-safety property, or a regular and a safety property are not serially enforceable even without memory bounds on each enforcer; thus, they remain not enforceable when bounds are imposed. Illustrating counterexamples to compositionality are provided in Examples 3, 4 and 5.

Example 3 *Consider regular properties RE_1 and RE_2 and check whether their serial composition is enforceable. An input sequence equal to aaaaaabc and buffer size as 5 (which is the maximum of the number of states in $\mathcal{A}_{RE_1}$ and $\mathcal{A}_{RE_2}$) is taken. Table 2 shows the incremental computation by the bounded monolithic and serial enforcers respectively, where q is the state reached upon the input event in the corresponding row, Buffer contains the buffered events by the enforcer and output is the events released by the enforcer.*

As shown in Table 2, the final outputs produced by the serial compositional enforcers in the order $E^{RE_1,k} \Rrightarrow E^{RE_2,k}$ and $E^{RE_2,k} \Rrightarrow E^{RE_1,k}$ for the input "aaaaaabc" are "aaaa" and "aaaaab", which is not sound with respect to the enforcer for RE_1 and RE_2, i.e.,

$$E^{RE_1,k} \Rrightarrow E^{RE_2,k}(aaaaaabc) = aaaa \not\models RE_1$$

$$E^{RE_2,k} \Rrightarrow E^{RE_1,k}(aaaaaabc) = aaaaab \not\models RE_2.$$

This demonstrates that the serial compositional approach fails to ensure soundness, and hence, in general, regular properties are not enforceable using a serial compositional approach.

Example 4 *Consider regular property RE_2 and co-safety property CS_1 and check whether their serial composition is enforceable. An input sequence equal to aaaaaacb and buffer size as 5 (which is the maximum of the number of states in $\mathcal{A}_{RE_2}$ and $\mathcal{A}_{CS_1}$) is taken. Table 3 shows the incremental computation by the bounded monolithic and serial enforcers respectively.*

As shown in Table 3, the final outputs produced by the serial compositional enforcers in the order $E^{RE_2,k} \Rrightarrow E^{CS_1,k}$ and $E^{CS_1,k} \Rrightarrow E^{RE_2,k}$ for the input "aaaaaacb" are "aaaaab" and "aaaaa", which is not sound with respect to the enforcer for RE_2 and CS_1 respectively, i.e.,

$$E^{RE_2,k} \Rrightarrow E^{CS_1,k}(aaaaaacb) = aaaaab \not\models RE_2$$

$$E^{CS_1,k} \Rrightarrow E^{RE_2,k}(aaaaaacb) = aaaaa \not\models CS_1.$$

This demonstrates that the serial compositional approach fails to ensure soundness, and hence, in general, a regular and co-safety property are not enforceable using a serial compositional approach.

Example 5 *Consider regular property RE_1 and safety property S_1 and check whether their serial composition is enforceable. An input sequence equal to aacccb and buffer size as 4 (which is the maximum of the number of states in $\mathcal{A}_{RE_1}$ and $\mathcal{A}_{S_1}$) is taken. Table 4 shows the incremental computation by the bounded monolithic and serial enforcers respectively.*

As shown in Table 4, the final outputs produced by the serial compositional enforcers in the order $E^{RE_1,k} \Rrightarrow E^{S_1,k}$ and $E^{S_1,k} \Rrightarrow E^{RE_1,k}$ for the input "aacccb" are "accc" and "acccb", which is not sound with respect to the enforcer for RE_1 and S_1, i.e.,

$$E^{RE_1,k} \Rrightarrow E^{S_1,k}(aacccb) = accc \not\models RE_1$$

$$E^{S_1,k} \Rrightarrow E^{RE_1,k}(aacccb) = acccb \not\models S_1.$$

This demonstrates that the serial compositional approach fails to ensure soundness, and hence, in general, a regular and a

Table 2 Counterexample to compositionality of the serial approach.

S No.	*Input*	*Monolithic*			*Serial*	
					$E^{RE_1,k} \Rrightarrow E^{RE_2,k}$	$E^{RE_2,k} \Rrightarrow E^{RE_1,k}$
		q	*Buffer (5)*	*output*		
1	a	$q_1 q_5$	a	ε	ε	ε
2	aa	$q_1 q_5$	aa	ε	ε	ε
3	aaa	$q_1 q_5$	aaa	ε	ε	ε
4	$aaaa$	$q_1 q_5$	$aaaa$	ε	ε	ε
5	$aaaaa$	$q_1 q_5$	$aaaaa$	ε	ε	ε
6	$aaaaaa$	$q_1 q_5$	$a \not\in aaaa$	ε	ε	ε
7	$aaaaaab$	$q_2 q_6$	$a \not\in aaab$	ε	$aaaaa$	ε
8	$aaaaaabc$	$q_1 q_7$	$a \not\in aabc$	ε	$aaaaa \not\models RE_1$	$aaaaab \not\models RE_2$

Table 3 Counterexample to compositionality of the serial approach

S No.	*Input*	*Monolithic*			*Serial*	
					$E^{RE_2,k} \Rrightarrow E^{CS_1,k}$	$E^{CS_1,k} \Rrightarrow E^{RE_2,k}$
		q	*Buffer (5)*	*output*		
1	a	$q_5 q_{24}$	a	ε	ε	ε
2	aa	$q_5 q_{24}$	aa	ε	ε	ε
3	aaa	$q_5 q_{24}$	aaa	ε	ε	ε
4	$aaaa$	$q_5 q_{24}$	$aaaa$	ε	ε	ε
5	$aaaaa$	$q_5 q_{24}$	$aaaaa$	ε	ε	ε
6	$aaaaaa$	$q_5 q_{24}$	$a \not\in aaaa$	ε	ε	ε
7	$aaaaaac$	~~$q_6 q_{26}$~~ $q_5 q_{24}$	$aaaaa$	ε	ε	ε
8	$aaaaaacb$	$q_6 q_{25}$	$a \not\in aaab$	ε	$aaaaab \not\models RE_2$	$aaaaa \not\models CS_1$

Table 4 Counterexample to compositionality of the serial approach.

S No.	Input	Monolithic			Serial	
					$E^{RE_1,k} \Rrightarrow E^{S_1,k}$	$E^{S_1,k} \Rrightarrow E^{RE_1,k}$
		q	*Buffer (4)*	*output*		
1	a	q_1q_{10}	a	ε	ε	ε
2	aa	q_1q_{11}	aa	ε	ε	ε
3	aac	q_1q_{11}	aac	ε	ε	ε
4	$aacc$	q_1q_{11}	$aacc$	ε	ε	ε
5	$aaccc$	q_1q_{11}	$aa\not{c}\ cc$	ε	ε	ε
6	$aacccb$	q_2q_{11}	ε	$aaccb$	$accc \not\models RE_1$	$acccb \not\models S_1$

safety property is not enforceable using a serial compositional approach.

In the remainder of this section, we investigate whether composition remains viable for specific groupings of subclasses of regular properties, namely, i) safety and co-safety properties, ii) groups of safety properties, and iii) groups of co-safety properties, when each enforcer is subject to memory bounds, given that these were compositional without such bounds (Pinisetty & Tripakis 2016; Pinisetty et al. 2022).

3.1. Cosafety-Safety: Not Enforceable

Proposition 1 *(Non-enforceability of a set of co-safety and safety properties) Consider a co-safety and a safety property φ_1, φ_2 and their corresponding automata $\mathcal{A}_{\varphi_1}, \mathcal{A}_{\varphi_2}$. When the bounded enforcers (with bound k where k is the maximum of the number of states in $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$) are combined serially as $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ (or in any order), then, in general, a co-safety and a safety property are not enforceable using the serial compositional approach i.e., $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ cannot always act as an enforcer with respect to $\varphi_1 \wedge \varphi_2$.*

Example 6 shows that the serial composition scheme does not work in general for a co-safety and a safety property.

Example 6 *Consider co-safety property CS_3 and safety property S_1 and check whether their serial composition is enforceable. An input sequence equal to aacccb and buffer size as 4 (which is the maximum of the number of states in $\mathcal{A}_{CS_3}$ and $\mathcal{A}_{S_1}$) is taken. Table 5 shows the incremental computation by the bounded monolithic and serial enforcers respectively.*

As shown in Table 5, the final outputs produced by the serial compositional enforcer in the order $E^{CS_3,k} \Rrightarrow E^{S_1,k}$ or $E^{S_1,k} \Rrightarrow E^{CS_3,k}$ for the input "aacccb" are "accc" and "acccb", which are not sound with respect to the enforcer for CS_3 and S_1 respectively, i.e.,

$$E^{CS_3,k} \Rrightarrow E^{S_1,k}(aacccb) = accc \not\models CS_3$$

$$E^{S_1,k} \Rrightarrow E^{CS_3,k}(aacccb) = acccb \not\models S_1.$$

3.2. Cosafety-Cosafety: Not Enforceable

Proposition 2 *(Non-enforceability of a set of co-safety properties) Consider co-safety properties φ_1, φ_2 and their corresponding automata $\mathcal{A}_{\varphi_1}, \mathcal{A}_{\varphi_2}$. When the bounded enforcers (with bound k) are combined serially as $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$, then, in general, the set of co-safety properties are not enforceable using the serial compositional approach i.e., $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ cannot always act as an enforcer with respect to $\varphi_1 \wedge \varphi_2$.*

Example 7 shows that the serial composition scheme does not work in general for a set of co-safety properties.

Example 7 *Consider co-safety properties CS_1 and CS_2 and check whether their serial composition is enforceable. An input sequence equal to aaaaaacb and buffer size as 5 (which is the maximum of the number of states in $\mathcal{A}_{CS_1}$ and $\mathcal{A}_{CS_2}$) is taken. Table 6 shows the incremental computation by the bounded monolithic and serial enforcers respectively.*

Remark 1 *(Observation on enforceability of the above co-safety properties under serial composition in reverse order.) We observe that when the enforcers are combined in the reverse order, i.e., $E^{CS_1,k} \Rrightarrow E^{CS_2,k}$, the given set of co-safety properties becomes enforceable using the serial compositional approach.*

Specifically, during enforcement $E^{CS_1,k}$ suppresses event c when at state q_{24}, as it would have otherwise led to the dead state q_{26}. The resulting output is then passed to $E^{CS_2,k}$. Due to the suppression of event c, the automaton $\mathcal{A}^{CS_2,k}$ can no longer reach an accepting state, causing it to produce ε as the overall output. This output satisfies the properties of soundness, transparency, and other enforcement conditions.

The enforceability in this setting is attributed to two key conditions:

– *Extension-closure of co-safety properties: Co-safety properties delay producing outputs until a good (accepting) prefix has been observed.*
– *Order of enforcers in composition: $E^{CS_1,k}$ is placed before $E^{CS_2,k}$, where $E^{CS_1,k}$ suppresses events which are leading to its dead states, and these suppressed events would otherwise enable $\mathcal{A}_{\varphi_2}$ to reach accepting states. Consequently, $E^{CS_2,k}$ outputs ε, maintaining correctness.*

This was not the case with the previous set of properties, which were not extension-closed. Thus, in that case, the second enforcer in the series could potentially produce outputs that might violate the property enforced by the first enforcer in the series.

3.3. Safety-Safety: Not Enforceable

Proposition 3 *(Non-enforceability of a set of safety properties using serial composition approach) Consider safety properties φ_1, φ_2 and their corresponding automata $\mathcal{A}_{\varphi_1}, \mathcal{A}_{\varphi_2}$. When the bounded enforcers (with bound k) are combined serially as $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ (or in any order), then, in general, the set of safety properties are not enforceable using the serial compositional approach i.e., $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ cannot always act as an enforcer with respect to $\varphi_1 \wedge \varphi_2$.*

Example 8 shows that the serial composition scheme does not work in general for a set of safety properties.

Example 8 *Consider safety properties S_2 and S_3 and check whether their serial composition is enforceable. Tables 7 and 8 show the incremental computation by the bounded monolithic and serial enforcers respectively.*

As shown in Table 7 and 8, the final outputs produced by the serial compositional enforcer in the order $E^{S_3,k} \Rrightarrow E^{S_2,k}$ and $E^{S_2,k} \Rrightarrow E^{S_3,k}$ for the inputs "aabcab" and "acba" are "acb" and "aca" respectively, which is not sound with respect to the enforcer for S_3 and S_2 respectively. i.e.,

$E^{S_3,k} \Rrightarrow E^{S_2,k}(aabcab) = acb \not\models S_3$. $E^{S_2,k} \Rrightarrow E^{S_3,k}(acba) = aca \not\models S_2$.

This demonstrates that the serial compositional approach fails to ensure soundness, and hence, in general, a set of safety properties is not enforceable using a serial compositional approach.

3.4. Intuition behind failure in serial composition

In serial composition, the output of the first enforcer is fed as input to the second enforcer. If the second enforcer suppresses certain events from the output of the first enforcer (for instance, due to memory constraints such as insufficient buffer space), it may happen that a suppressed event is essential for satisfying the first property. In other words, from the current state of the automaton modelling the first property, the suppressed event would have led to an accepting state. Consequently, the suppression of such an event may cause the first property to no longer be satisfied in the final output of the serial composition, thereby violating *soundness*.

For example, in Table 2, in $E^{RE_1,k} \Rrightarrow E^{RE_2,k}$, the second enforcer $E^{RE_2,k}$ suppresses 'b' and the final output of $E^{RE_1,k} \Rrightarrow E^{RE_2,k}$ is "*aaaaa*" which RE_1 no longer satisfies.

In the following section, we introduce and formalize a sufficient condition on the automata modelling the properties, under which certain combinations of properties become serially enforceable.

4. Conditions for Enforceability of Composite Properties

From the previous section, we see that, when an event is suppressed by one automaton, either because (1) the automaton transitions to a non-accepting state upon that event and the event is exhausted in attempts to reach an accepting state, (2) it is discarded during buffer clearing, or (3) the automaton transitions to a dead state, and this same event corresponds to an essential transition in the other automaton (i.e., a transition that must be taken to eventually reach an accepting state), then such suppression by the second enforcer will prevent the first property from being satisfied.

We formally define conditions on automata considering above cases. By verifying that the automata of the given properties satisfies the conditions, one can ensure serial enforceability prior to enforcement for certain sets of properties. Since the property set is small, the verification process introduces minimal overhead. Also, note that this analysis of checking whether a given set of properties is enforceable is done offline. Thus, it does not have any influence on the runtime overhead for enforcement.

Proposition 4 *(Serial enforceability of composite properties.) Consider a tuple of properties $\langle \varphi_1, \varphi_2 \rangle$, modelled as complete automata $\mathcal{A}_{\varphi_1} = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ and $\mathcal{A}_{\varphi_2} = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$. Let the condition $CN(\varphi_1, \varphi_2)$ be as follows : in the product automaton $\mathcal{A} = \mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$, there is no reachable state $(q_1, q_2) \in reach(q_0^1, q_0^2)$ and action $a \in \Sigma$ such that $\delta((q_1, q_2), a) = (q_1', q_2')$ with $(q_1', q_2') \in coreach(F_1) \times (Q_2 \setminus coreach(F_2))$ and $q_1' \neq q_1$. Now, if φ_1 is a regular property and φ_2 a safety property, or φ_1 and φ_2 are both co-safety properties and $CN(\varphi_1, \varphi_2)$ is satisfied, then both $\langle \varphi_1, \varphi_2 \rangle$ and $\langle \varphi_2, \varphi_1 \rangle$ are serially enforceable (as per Def. 3).*

The proposition says that a regular property φ_1 and a safety property φ_2 (or two co-safety properties, or a co-safety and a safety property, or two safety properties) are serially enforceable when their enforcers (with bounded buffers of size k) are combined serially in the order $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$, when their automata $\mathcal{A}\varphi_1$ and $\mathcal{A}\varphi_2$, respectively, satisfy the following condition: it should not be the case that, $\mathcal{A}\varphi_2$ transitions to a dead state on an event and this same event corresponds to an essential transition in $\mathcal{A}\varphi_1$.

Intuition of proof: The proof relies on an induction on the length of the input word σ. The induction works because we show if correctness holds up to σ, then handling one more event a in the pipeline of $E^{\varphi_1,k} \Rrightarrow E^{\varphi_2,k}$ does not introduce violations due to suppression or buffering.

Table 5 Counterexample to compositionality of the serial approach

S No.	Input	Monolithic			Serial	
					$E^{CS_3,k} \Rrightarrow E^{S_1,k}$	$E^{S_1,k} \Rrightarrow E^{CS_3,k}$
		q	*Buffer (4)*	*output*		
1	a	$q_{33}q_{10}$	a	ε	ε	ε
2	aa	$q_{33}q_{11}$	aa	ε	ε	ε
3	aac	$q_{33}q_{11}$	aac	ε	ε	ε
4	$aacc$	$q_{33}q_{11}$	$aacc$	ε	ε	ε
5	$aaccc$	$q_{33}q_{11}$	$aa \notin cc$	ε	ε	ε
6	$aacccb$	$q_{34}q_{11}$	ε	$aaccb$	$accc \not\models CS_3$	$acccb \not\models S_1$

Table 6 Counterexample to compositionality of the serial approach.

S No.	Input	Monolithic			Serial	
					$E^{CS_1,k} \Rrightarrow E^{CS_2,k}$	$E^{CS_2,k} \Rrightarrow E^{CS_1,k}$
		q	*Buffer (5)*	*output*		
1	a	$q_{24}q_{28}$	a	ε	ε	ε
2	aa	$q_{24}q_{28}$	aa	ε	ε	ε
3	aaa	$q_{24}q_{28}$	aaa	ε	ε	ε
4	$aaaa$	$q_{24}q_{28}$	$aaaa$	ε	ε	ε
5	$aaaaa$	$q_{24}q_{28}$	$aaaaa$	ε	ε	ε
6	$aaaaaa$	$q_{24}q_{28}$	$a \notin aaaa$	ε	ε	ε
7	$aaaaaac$	~~$q_{26}q_{29}$~~ $q_{24}q_{28}$	$aaaaa$	ε	ε	ε
8	$aaaaaacb$	$q_{25}q_{28}$	$a \notin aaab$	ε	ε	$aaaab \not\models CS_2$

Proof: Let us prove separately for enforceable sets. We prove for regular-safety and co-safety-co-safety. The proofs for co-safety-safety and safety-safety combinations follow the same structure as the proof for the regular-safety case.

Serial enforceability of regular-safety properties satisfying sufficient condition. Given one regular property RE and one safety property S. Let their enforcers be combined as $E^{RE,k} \Rrightarrow E^{S,k}$ and $E^{S,k} \Rrightarrow E^{RE,k}$, where k is the maximum size of buffer provided to both the enforcers. k will be equal to the maximum number of states in the automaton defining RE and S. Let us prove this using induction on the input sequence σ.

Induction basis. If $\sigma = \varepsilon$, output of $E^{RE,k} \Rrightarrow E^{S,k}$ or $E^{S,k} \Rrightarrow E^{RE,k}$ will be ε. Thus the proposition holds trivially.

Induction step. Assume that the proposition holds for every $\sigma \in \Sigma^*$. We now prove that the proposition holds for $\sigma \cdot a$ for any $a \in \Sigma$.

Let us consider the case where enforcer are combined as $E^{RE,k} \Rrightarrow E^{S,k}$. We have following cases:

1. Case 1: When automaton of RE reaches a non-accepting state on event a.

   The event is buffered and nothing is produced as output of $E^{RE,k} \Rrightarrow E^{S,k}$. The output for $\sigma \cdot a$ will be same as output

Table 7 Counterexample to compositionality of the serial approach.

S No.	Input	Monolithic			Serial	
		q	*Buffer (4)*	*output*	$E^{S_2,k} \Rrightarrow E^{S_3,k}$	$E^{S_3,k} \Rrightarrow E^{S_2,k}$
1	a	$q_{14}q_{19}$	ε	a	a	a
2	aa	$\cancel{q_{17}q_{20}}q_{14}q_{19}$	ε	a	a	a
3	aab	$q_{15}q_{19}$	ε	ab	ab	a
4	$aabc$	$\cancel{q_{17}q_{20}}q_{15}q_{19}$	ε	ab	ab	ac
5	$aabca$	$q_{16}q_{20}$	ε	aba	aba	ac
6	$aabcab$	$\cancel{q_{16}q_{22}}q_{16}q_{20}$	ε	aba	aba	$acb \not\models S_3$

Table 8 Counterexample to compositionality of the serial approach.

S No.	Input	Monolithic			Serial	
		q	*Buffer (4)*	*output*	$E^{S_2,k} \Rrightarrow E^{S_3,k}$	$E^{S_3,k} \Rrightarrow E^{S_2,k}$
1	a	$q_{14}q_{19}$	ε	a	a	a
2	ac	$q_{14}q_{20}$	ε	ac	ac	ac
3	acb	$\cancel{q_{15}q_{22}}q_{14}q_{20}$	ε	ac	ac	ac
4	$acba$	$\cancel{q_{17}q_{20}}q_{14}q_{20}$	ε	ac	$aca \not\models S_2$	ac

of σ. Since, we assumed that the proposition holds for σ, thus it holds for $\sigma \cdot a$ as well.

2. Case 2: When automaton of RE reaches an accepting state on event a, releases the input word or a subword of the input word as output to be fed to $E^{S,k}$. Now, there will be two cases on whether any subword of the word released by $E^{RE,k}$ is accepted by $E^{S,k}$ or not:

   (a) Case 2a: None of the subwords of the word released by $E^{RE,k}$ is accepted by $E^{S,k}$.
   The output in this case remains unchanged. Thus, the proposition holds trivially.

   (b) Case 2b: A subword of the word released by $E^{RE,k}$ is accepted by S.
   The output of $E^{S,k}$ is the resultant output of $E^{RE,k} \Rrightarrow E^{S,k}$. However, it may be case that the resultant output now violates RE because of some events being suppressed by $E^{S,k}$. There may be following cases where $E^{S,k}$ suppress events from output of $E^{RE,k}$:

   – Case: When $E^{S,k}$ suppresses some events whenever S reaches a dead state to satisfy property S, such that now property RE can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
   This case will not happen, as we assume that the automata of the corresponding properties satisfy the condition. Thus, $RE \cap S$ is satisfied.
   – Case: When $E^{S,k}$ suppresses some events (a suffix) because of reaching a non-accepting state, such that now RE can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
   The non-accepting states are dead states in case of safety properties. Thus, this case is similar to the above case.
   – Case: When $E^{S,k}$ suppresses some events during buffer cleaning, such that now RE can never

lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
This case will not happen as no events are buffered by $E^{S,k}$[3].

Thus, we see that in all the cases *soundness* is satisfied, i.e., the final output of the serial enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ satisfies $RE \cap S$.

Regarding *transparency* and *optimal suppression* constraints, since both enforcers independently satisfy these conditions, their serial composition $E^{RE,k} \Rrightarrow E^{S,k}$ also upholds them. Specifically, if the input trace can still be extended to satisfy both properties and no suppression has occurred so far, both $E^{RE,k}$ and $E^{S,k}$ will relay the events unchanged, ensuring that the composed enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ preserves the input trace without unnecessary suppression. However, if the input trace cannot be extended to satisfy both properties, or suppression has already occurred, then either $E^{RE,k}$ or $E^{S,k}$ (or both) will suppress events as per their respective constraints. This guarantees that the composed enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ also performs the necessary suppression to enforce both the properties.

Regarding the *monotonicity* constraint, we observe that the output of the serial enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ for an input trace σ is extended if the property is satisfied by the output corresponding to the extended input $\sigma \cdot a$. The part of the output that has already been released for the input σ remains unchanged. Thus, monotonicity is satisfied.

Regarding *optimal mode change*, The optimal mode change constraint requires that the mode transitions to $\bot$ (indicating that some events are suppressed from the input word) only when necessary, and once the mode changes to $\bot$, it cannot revert back to $\top$ (indicating no events are suppressed). And assuming that, the mode of the overall serial enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ is determined as follows: if the mode of any one of the enforcers is $\bot$, then the overall mode of the serial enforcer is $\bot$; otherwise, it remains $\top$.
Therefore, if each individual enforcer respects the optimal mode change constraint and the above condition governs the composition, then the serially composed enforcer $E^{RE,k} \Rrightarrow E^{S,k}$ also satisfies the optimal mode change constraint.

[3] The safety property enforcer does not buffer any events because its state space consists only of accepting and dead states. Events are immediately released when the enforcer reaches an accepting state and suppressed when it reaches a dead state. Since there are no non-accepting (intermediate) states where events are held, buffering is not required.

Let us consider the case where enforcers are combined as $E^{S,k} \Rrightarrow E^{RE,k}$. We have following cases:

1. Case 1: When automaton of S reaches a non-accepting state on event a.

   The event is suppressed by $E^{S,k}$ and thus by $E^{S,k} \Rrightarrow E^{RE,k}$. Thus, proposition holds.

2. Case 2: When automaton of S reaches an accepting state on event a, releases the input word or a subword of the input word as output to be fed to $E^{RE,k}$. Now, there will be two cases on whether any subword of the word released by $E^{S,k}$ is accepted by $E^{RE,k}$ or not:

   (a) Case 2a: None of the subwords of the word released by $E^{S,k}$ is accepted by $E^{RE,k}$.

   The output in this case remains unchanged. Thus, the proposition holds trivially.

   (b) Case 2b: A subword of the word released by $E^{S,k}$ is accepted by RE.

   The output of $E^{RE,k}$ is the resultant output of $E^{S,k} \Rrightarrow E^{RE,k}$. However, it may be the case that the resultant output now violates S because of some events being suppressed by $E^{RE,k}$. There may be following cases where $E^{RE,k}$ suppress events from output of $E^{S,k}$:

   – Case: When $E^{RE,k}$ suppresses some events whenever RE reaches a dead state, such that now S can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
   This case will not happen, as we assume that the automata of the corresponding properties satisfy the condition. Thus, $S \cap RE$ is satisfied
   – Case: When $E^{RE,k}$ suppresses some events (a suffix) because of reaching a non-accepting state, such that now S can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
   This case is invalid for safety properties, as they are prefix closed properties, meaning the prefixes (the word without the suppressed events) satisfy the property.
   – Case: When $E^{RE,k}$ suppresses some events during buffer cleaning, such that now S can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).
   This case will not happen as even if all the looping events are suppressed during buffer cleaning by $E^{RE,k}$, then upon the next essential event, automaton of S will not transiton to a dead state as the condition prohibits that.

   Thus, we see that in all the cases *soundness* is satisfied, i.e., the final output of the serial enforcer $E^{S,k} \Rrightarrow E^{RE,k}$ satisfies $S \cap RE$.

A similar reasoning as used for $E^{RE,k} \Rrightarrow E^{S,k}$ can be applied here to establish transparency, monotonicity, and other related constraints.

Thus, the proposition holds.

Serial enforceability of co-safety-co-safety properties satisfying the identified condition. The proposition says that a set of co-safety properties are serially enforceable if their corresponding automata satisfy the following condition: it should not be the case that, $\mathcal{A}\varphi_2$ transitions to a dead state on an event and this same event corresponds to an essential transition in $\mathcal{A}\varphi_1$.

Given two co-safety properties CS1 and CS2. Let their enforcers be combined as $E^{CS1,k} \Rrightarrow E^{CS2,k}$ and $E^{CS2,k} \Rrightarrow E^{CS1,k}$, where k is the maximum size of buffer provided to both the enforcers. k will be equal to the maximum number of states in the automaton defining CS1 and CS2. Let us prove this using induction on the input sequence σ.

Induction basis. If $\sigma = \varepsilon$, output of $E^{CS1,k} \Rrightarrow E^{CS2,k}$ will be ε. Thus the proposition holds trivially.

Induction step. Assume that the proposition holds for every $\sigma \in \Sigma^*$. We now prove that the proposition holds for $\sigma \cdot a$ for any $a \in \Sigma$.

Let us consider the case where enforcer are combined as $E^{CS1,k} \Rrightarrow E^{CS2,k}$. The case of $E^{CS2,k} \Rrightarrow E^{CS1,k}$ will simillarly follow. We have following cases:

1. Case 1: When automaton of CS1 reaches a non-accepting state on event a.

   The event is buffered and nothing is produced as output of $E^{CS1,k} \Rrightarrow E^{CS2,k}$. The output for $\sigma \cdot a$ will be same as output of σ. Since, we assumed that the proposition holds for σ, thus it holds trivially for $\sigma \cdot a$ as well.

2. Case 2: When automaton of CS1 reaches an accepting state on event a, releases the input word or a subword of the input word as output to be fed to $E^{CS2,k}$. Now, there will be two cases on whether any subword of the word released by $E^{CS1,k}$ is accepted by $E^{CS2,k}$ or not:

   (a) Case 2a: None of the subword of the word released by $E^{CS1,k}$ is accepted by $E^{CS2,k}$.

   The output in this case remains unchanged. Thus, the proposition holds trivially.

   (b) Case 2b: The subword of the word released by $E^{CS1,k}$ is accepted by $E^{CS2,k}$.

   The output of $E^{CS2,k}$ is the resultant output of $E^{CS1,k} \Rrightarrow E^{CS2,k}$. However, it may be the case that the resultant output now violates CS1 because of some events being suppressed by $E^{CS2,k}$. There may be following cases where $E^{CS2,k}$ suppresses events from output of $E^{CS1,k}$:

   – Case: When $E^{CS2,k}$ suppresses some events whenever CS2 reaches a dead state, such that now CS1 can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).

   This case will not happen, as we assume that the automata of the corresponding properties satisfy the condition. Thus, $CS1 \cap CS2$ is satisfied.

   – Case: When $E^{CS2,k}$ suppresses some events during buffer cleaning, such that now CS1 can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run).

   This case will not happen as even if all the looping events are suppressed during buffer cleaning by $E^{CS2,k}$, then upon the next essential event, automaton of CS1 will not transition to a dead state as the condition prohibits that.

   Note that the case of "$E^{CS2,k}$ suppressing some events (a suffix) because of reaching a non-accepting state, such that now CS1 can never lead to satisfaction because those were essential events (engaged in non-looping transitions in its accepting run)"– will not happen here. This is because co-safety properties are extension-closed; once an accepting state is reached, no extension (suffix) can lead to a non-accepting state.

   Thus, we see that in all the cases *soundness* is satisfied, i.e., the final output of the serial enforcer $E^{CS1,k} \Rrightarrow E^{CS2,k}$ satisfies $CS1 \cap CS2$. The proofs for transparency, monotonicity, and other related constraints will follow from the proof of serial enforceability of regular-safety properties.

Thus, the proposition holds.

Algorithm.: The proposed condition (Proposition 4) can be verified (in linear time relative to the size of the product automaton) using Algorithm 1. The algorithm begins by constructing the product automaton $\mathcal{A}_P$ from the input automata $\mathcal{A}_{\varphi_1}$ and $\mathcal{A}_{\varphi_2}$, which correspond to the specified properties φ_1 and φ_2. It then computes the *coreach* (defined in Sect. 2.1) of the final states for each automaton, i.e., $coreach(F_1)$ and $coreach(F_2)$. This information is used to identify dead states, for example by computing $Q_2 \setminus coreach(F_2)$.

Next, for each state reachable from the initial state of $\mathcal{A}_P$, (q_0^1, q_0^2), the algorithm checks whether, for any action $a \in \Sigma$, $\mathcal{A}_P$ transitions to a state such that, the corresponding state in $\mathcal{A}_{\varphi_2}$ differs from the source state, and this state is a dead state in $\mathcal{A}_{\varphi_2}$ (i.e., it belongs to $Q_2 \setminus coreach(F_2)$). If such a case is found, then serial composition does not work for the specified set of properties.

The following example (Example 9) illustrates the process of checking the condition for any specified automata using Algorithm 1.

Example 9 *(Checking enforceability of two properties using serial composition.) Consider two properties P1 and P2 modelled as automata $\mathcal{A}_{P1}$ and $\mathcal{A}_{P2}$ in Figs. 10 and 11. We get the following from the steps in Algorithm 1:*

Algorithm 1 Checking enforceability

Input: Automata $\mathcal{A}_{\varphi_1} = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ and $\mathcal{A}_{\varphi_2} = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$ over the same alphabet Σ, and their product automata $\mathcal{A}_P$ where $\mathcal{A}_P = \mathcal{A}_1 \times \mathcal{A}_2$

Step1: Compute $coreach(F_1)$ in $\mathcal{A}_{\varphi_1}$ and $coreach(F_2)$ in $\mathcal{A}_{\varphi_2}$

Step2: Traverse the product automaton $\mathcal{A}_{\varphi_1} \times \mathcal{A}_{\varphi_2}$ from the initial state (q_0^1, q_0^2) in depth-first order and find the set of states reached i.e., $reach(q_0^1, q_0^2)$.

for each (q_1, q_2) in $reach(q_0^1, q_0^2)$ **do**
  for each transition $(q_1, q_2) \xrightarrow{a} (q_1', q_2')$ **do**
    if $q_1' \neq q_1$ **then**
      if $(q_1', q_2') \in \{coreach(F_1) \times (Q_2 \setminus coreach(F_2))\}$ **then**
        $E^{\varphi_1,k} \Rightarrow E^{\varphi_2,k}$ does not work.
        exit

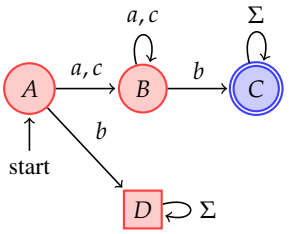

Figure 10 $\mathcal{A}_{P1}$

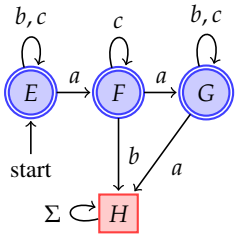

Figure 11 $\mathcal{A}_{P2}$

1. *Find* $coreach(\{C\})$ *and* $coreach(\{E, F, G\})$
   - $coreach(\{C\}) = \{C, B, A\}$
   - $coreach(\{E, F, G\}) = \{E, F, G\}$

2. *Find first state reached from* AE
   - $reach(\{AE\}) = \{AE, ...\}$

3. *For state* AE,
   (a) *for event* a, $AE \xrightarrow{a} BF$ *and* $E \neq F$, $BF \notin \{\{C, B, A\} \times (\{E, F, G, H\} \setminus \{E, F, G\})$ *i.e.,* $BF \notin \{\{C, B, A\} \times \{H\}$ *i.e.,* $BF \notin \{CH, BH, AH\}$
      - *Continue with next event* b
   (b) *for event* b, $AE \xrightarrow{b} DE$ *and* $E = E$
      - *Continue with next event* c
   (c) *for event* c, $AE \xrightarrow{c} BE$ *and* $E = E$
      - *Continue with next state*

4. *Find next state reached from* AE
   - $reach(\{AE\}) = \{AE, BF, ...\}$

5. *For state* BF,
   (a) *for event* a, $BF \xrightarrow{a} BG$ *and* $F \neq G$, $BG \notin \{CH, BH, AH\}$
      - *Continue with next event* b
   (b) *for event* b, $BF \xrightarrow{b} CH$ *and* $F \neq H$, $CH \in \{CH, BH, AH\}$
      - $E^{\mathcal{A}_{P1},k} \Rightarrow E^{\mathcal{A}_{P2},k}$ *does not work,* ***exit***

We see in this example that the condition is violated, thus the given two properties are serially not enforceable.

The following section presents a case study demonstrating how the proposed enforcement framework can be effectively applied to enforce multiple properties compositionally when their respective automata satisfy the sufficient condition, even when each enforcer operates with a bounded buffer.

5. Case study

We can develop a variety of case studies that demonstrate how our bounded compositional framework can effectively monitor system executions against specified properties. For example, in a web application handling user sessions, we can enforce the property that a user must be authenticated before accessing sensitive resources. Our framework composes monitors that enforce session validity and authentication, while operating under bounded-memory by buffering only a limited number of (idempotent/repetitive) recent events. This ensures that even with constrained resources, the system maintains secure execution behaviour.

We examine here the case of reentrant applications, which are particularly vulnerable at runtime, especially when interacting

with untrusted or external components. A "reentrant application" (or reentrant code) refers to software– often specific functions or routines– that is designed to be safely interrupted in the middle of its execution and then called again ("re-entered") before its previous executions are complete. This means that the code can handle multiple simultaneous invocations (such as from different threads or interrupt handlers) without causing data corruption or other errors.

Reentrant applications offer benefits in concurrency but also introduce security risks (Cecchetti et al. 2021; Rodler et al. 2018), such as reentrancy attacks, race conditions and deadlocks, etc. For example, the infamous DAO hack on Ethereum DAO (Wikipedia contributors n.d.) is a prime example, where the malicious contract called the withdrawal function again within the same call, which resulted in millions of dollars stolen before the state was updated. One key mitigation strategy is to update the contract's internal state before making any external calls, ensuring that even if a function is re-entered, the critical state has already been updated and is safe against manipulation. Additionally, using mutexes effectively prevents malicious "recursive" invocations.

Need for monitoring reentrant applications at runtime. Reentrant applications are vulnerable at runtime especially when interacting with external components. Monitoring these applications at runtime becomes essential to ensure that their execution adheres to expected sequences. For example, the DAO attack (Wikipedia contributors n.d.) can also be prevented through runtime monitoring, where a monitor detects unusual re-entries during execution. This allows the system to identify and respond to malicious behaviour in real time, ensuring that even if the code allows reentrancy, harmful actions are intercepted at runtime before it causes harm.

Runtime enforcement frameworks such as (Shankar et al. 2022, 2024) are particularly suitable for monitoring non-critical reentrant applications, where delaying tasks and executing them "asynchronously" helps prevent the system from entering incorrect execution sequences or freezing. Moreover, delaying incoming events, such as requests, can serve as a mechanism to regulate processing rates, ensuring smoother and safer system behaviour.

5.1. Implementation and performance evaluation

We implemented both our bounded serial compositional enforcer and the monolithic enforcer, and conducted a performance evaluation to compare their efficiency. The evaluation uses properties (defined in Example 10) from the reentrant application scenario. These properties satisfy the condition given in Proposition 4 and thus are serially enforceable.

Example 10 *(Properties and their automata of reentrant applications.) We see some example properties which can be formalized and enforced on reentrant applications to ensure its correctness with the modelling of the properties using automata (given in Figs. 12-16). The set of actions* $\Sigma = \{w, u, c, r\}$*, where* w *denotes a reentrant call,* u *denotes state update,* c *denotes invocation of the critical section and* r *denotes release.*

- P_1*: No reentrant call before state update: A function that modifies the contract state should not allow re-entry into itself or other sensitive functions until the state update is completed.*
- P_2*: Single Invocation (Mutex Enforcement): In one session, only one invocation of the critical section.*
- P_3*: No reentrant call in a critical section: While executing within a critical section, no reentrant call by the same function to the critical section should be made.*
- P_4*: Limiting the number of state updates: In one session, the maximum number of state updates allowed is three.*
- P_5*: No Access After Release: After being released, resources cannot be accessed in the critical section.*

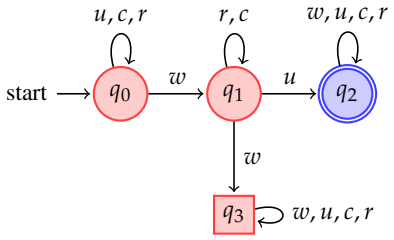

Figure 12 $\mathcal{A}_{P_1}$

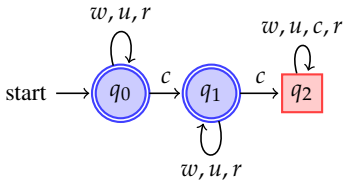

Figure 13 $\mathcal{A}_{P_2}$

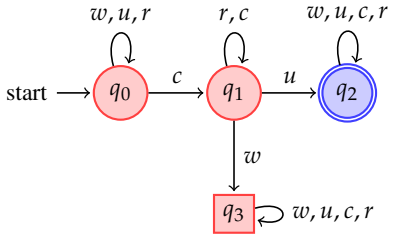

Figure 14 $\mathcal{A}_{P_3}$

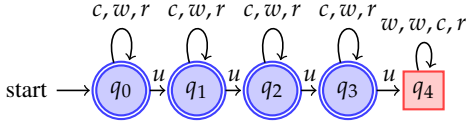

Figure 15 $\mathcal{A}_{P_4}$

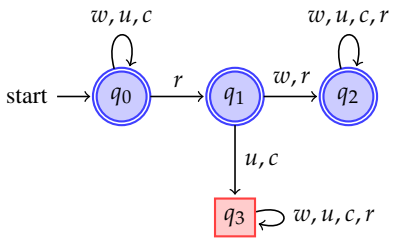

Figure 16 $\mathcal{A}_{P_5}$

We measured execution time by enforcing these properties on input traces of varying lengths. Additionally, we assessed scalability by varying the number of properties enforced while keeping the input trace length fixed. All experiments were performed on a system with an Intel Core i7-9700K CPU (3.60 GHz × 8), 32 GB RAM, running Ubuntu 20.04.2 LTS. The implementation of the enforcers, along with the performance analysis, is provided at: https://github.com/prisha-srinidi/BCRE.

5.1.1. Growth of time by the monolithic enforcer vs the serial enforcer against the size of input The length of the input trace, fed to the monolithic and serial enforcers, was varied linearly to observe the time taken by the enforcers. Table 9 (and Plot 17) gives the observations where column `Input` indicates the length of the input trace, column $\mathrm{T_M(s)}$[4] (resp. $\mathrm{T_S(s)}$) indicates the time taken in seconds to give an output by the monolithic (resp. serial) enforcer. We have the following observations from Table 9; for the given input sizes:

- The time taken by the monolithic enforcer is more than the time taken by the serial enforcer.
- With the linear increase in the input size, the time taken by the monolithic and serial enforcers varies linearly.

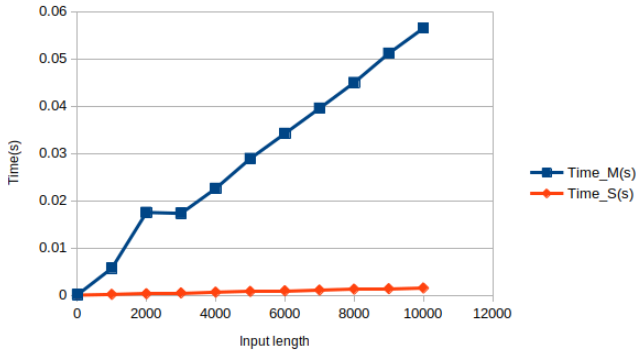

Figure 17 Effect on time of monolithic and serial enforcers by varying the input size

5.1.2. Growth of time by the monolithic enforcer vs the serial enforcer against the number of properties The number of properties to be enforced using the monolithic and serial enforcers are varied to observe how the time taken increases with the increase in the number of properties while keeping the input size fixed at 1000.

We have the following observation from Table 10 (and Plot 18), for the given input sizes: for a given number of properties and input size, the serial enforcer takes less time compared to the monolithic enforcer.

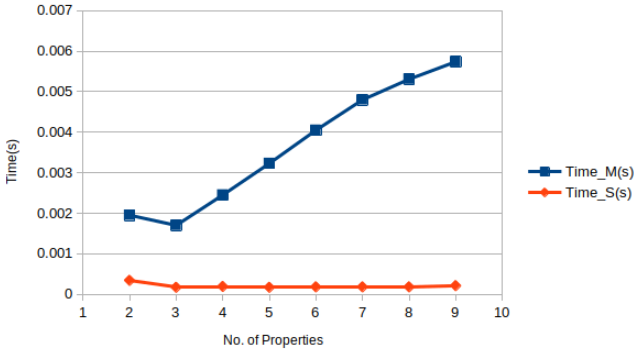

Figure 18 Effect on time by varying the number of properties

Additional performance evaluations can be conducted, for instance, by varying the buffer size and observing how the execution time changes, or by increasing the complexity of the properties and analyzing the corresponding impact on execution time. We plan to perform these evaluations in future.

Table 9 Effect on time of monolithic and serial enforcers by varying the input size

`Input`	$\mathrm{T_M(s)}$	$\mathrm{T_S(s)}$
10	0.000117	0.00002
1000	0.005724	0.000173
2000	0.017515	0.000315
3000	0.017313	0.000479
4000	0.022592	0.000642
5000	0.028928	0.000793
6000	0.034236	0.000931
7000	0.039569	0.001074
8000	0.045002	0.001275
9000	0.051197	0.001369
10000	0.056523	0.001515

[4] The total time taken by the monolithic enforcer includes only the online enforcement time and not the offline time for product construction.

Table 10 Effect on time by varying the number of properties

Properties	$T_M(s)$	$T_S(s)$
2	0.00195	0.00034
3	0.0017	0.00017
4	0.00245	0.00019
5	0.00323	0.00017
6	0.00405	0.00018
7	0.0048	0.00018
8	0.00531	0.00018
9	0.00574	0.00021

6. Conclusion

This paper introduces a serial compositional runtime enforcement framework for regular properties and their subclasses, where each enforcer operates under a bounded-memory constraint. Within this framework, enforcers can delay (buffer) or suppress events to ensure compliance with specified properties. We investigate the enforceability of regular, safety, and co-safety properties using the serial composition strategies. For properties that are not directly serially enforceable, we identify conditions on their automata which, when satisfied, enable certain groups of properties to be enforced serially. We provide formal proofs of enforceability wherever applicable and present counterexamples to illustrate cases where enforcement is not possible. A prototype implementation of the framework has been developed and we evaluate its performance using example properties in a reentrant application case study. The results demonstrate that the framework incurs less execution time compared to the monolithic approach, demonstrating their practical viability. Future work will focus on exploring enforceability in parallel composition.

References

Cecchetti, E., Yao, S., Ni, H., & Myers, A. C. (2021). Compositional security for reentrant applications. *CoRR*, *abs/2103.08577*. Retrieved from https://arxiv.org/abs/2103.08577

Falcone, Y., Jéron, T., Marchand, H., & Pinisetty, S. (2016, July). Runtime enforcement of regular timed properties by suppressing and delaying events. *Sci. Comput. Program.*, *123*(C), 2–41. doi: 10.1016/j.scico.2016.02.008

Ligatti, J., Bauer, L., & Walker, D. (2005). Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, *4*(1-2), 2–16. doi: 10.1007/s10207-004-0046-8

Pinisetty, S., Pradhan, A., Roop, P., & Tripakis, S. (2022, oct). Compositional runtime enforcement revisited. *Form. Methods Syst. Des.*, *59*(1–3), 205–252. doi: 10.1007/s10703-022-00401-y

Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., & Marchand, H. (2017). Predictive runtime enforcement. *Formal Methods Syst. Des.*, *51*(1), 154–199. doi: 10.1007/s10703-017-0271-1

Pinisetty, S., Roop, P. S., Smyth, S., Tripakis, S., & Hanxleden, R. v. (2017). 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).

Pinisetty, S., & Tripakis, S. (2016, 06). Compositional runtime enforcement. In (p. 82-99). doi: 10.1007/978-3-319-40648-0_7

Rodler, M., Li, W., Karame, G. O., & Davi, L. (2018). Sereum: Protecting existing smart contracts against re-entrancy attacks. *CoRR*, *abs/1812.05934*. Retrieved from http://arxiv.org/abs/1812.05934

Schneider, F. B. (2000, February). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, *3*(1), 30–50. doi: 10.1145/353323.353382

Shankar, S., Pradhan, A., Pinisetty, S., Rollet, A., & Falcone, Y. (2024). Bounded-memory runtime enforcement with probabilistic and performance analysis. *Formal Methods Syst. Des.*, *62*(1), 141–180. doi: 10.1007/S10703-024-00446-1

Shankar, S., Rollet, A., Pinisetty, S., & Falcone, Y. (2022). Bounded-memory runtime enforcement. In O. Legunsen & G. Rosu (Eds.), *Model checking software* (pp. 114–133). Cham: Springer International Publishing.

Talhi, C., Tawbi, N., & Debbabi, M. (2008). Execution monitoring enforcement under memory-limitation constraints. *Inf. Comput.*, *206*(2-4), 158–184. doi: 10.1016/j.ic.2007.07.009

Wikipedia contributors. (n.d.). *The dao (organization) — wikipedia.* https://en.wikipedia.org/wiki/The_DAO.

About the authors

Saumya Shankar received her Ph.D. in Computer Science and Engineering from the Indian Institute of Technology Bhubaneswar in 2024. Prior to defending her thesis, she joined the University of Auckland, New Zealand, as a research associate, later transitioning to a postdoctoral role. She is currently an Assistant Professor at IIIT Bangalore. Her research interests include formal methods and software engineering, with a particular focus on runtime verification and enforcement. Her work has been published in notable venues, including SPIN 2022, RV 2022, FMSD 2023, TIME 2023, ISEC 2023, and HSCC 2025. You can contact the author at saumyashankarsinha@gmail.com or visit https://www.iiitb.ac.in/faculty/saumya-shankar.

Thierry Jéron is Senior Researcher at Inria centre at Rennes University (France). His research interests comprise formal methods inspired by model-checking techniques from automata-like models and their timed extensions. In particular, he contributes to verification, automatic generation of test cases, on-line monitoring, diagnosis. He also contributes to the use of dynamic partial order reduction to the verification of distributed programs. You can contact the author at Thierry.Jeron@inria.fr or visit https://www.irisa.fr/prive/jeron/.

Prisha Srinidi received B.Tech. degree in computer science and engineering from Indian Institute of Technology Bhubaneswar, India. You can contact the author at 21cs01057@iitbbs.ac.in or visit https://github.com/prisha-srinidi.

Srinivas Pinisetty received a Ph.D. degree in Computer Science in January 2015 at INRIA, University of Rennes 1, Rennes, France. After completing masters in Computer Science at Eindhoven University Technology (TU/e) in 2009, he continued as a PDEng trainee at TU/e for two years. For his master's thesis project, he worked at ASML, Veldhoven, Netherlands in 2009, and as a Software Design Engineer trainee at Oce Technologies, Venlo, Netherlands in 2011. He is currently an Associate Professor in the School of Electrical and Computer Sciences at IIT Bhubaneswar. Prior to joining IIT Bhubaneswar he has worked as postdoctoral researcher at the University of Aalto, Finland, and later at the University of Gothenburg | Chalmers, Sweden.

His research interests include formal methods, and software engineering in general, and runtime verification and enforcement in particular. He also currently focuses on applications of runtime monitoring combined with AI/ML techniques in domains such as healthcare, and runtime monitoring approaches for safety of learning-based controllers. You can contact the author at spinisetty@iitbbs.ac.in or visit https://secs.iitbbs.ac.in/index.php/srinivaspinisetty/.