Predictive Runtime Enforcement*

Srinivas Pinisetty
Aalto University, Finland
srinivas.pinisetty@aalto.fi

Viorel Preoteasa
Aalto University, Finland
viorel.preoteasa@aalto.fi

Stavros Tripakis
Aalto and UC Berkeley
stavros.tripakis@gmail.com

Thierry Jéron
INRIA Rennes - Bretagne Atlantique, France
Thierry.Jeron@inria.fr

Yliès Falcone
Univ. Grenoble Alpes, Inria, LIG, Grenoble, France
Ylies.Falcone@imag.fr

Hervé Marchand
INRIA Rennes - Bretagne Atlantique, France
Herve.Marchand@inria.fr

ABSTRACT

Runtime enforcement (RE) is a technique to ensure that the (untrustworthy) output of a black-box system satisfies some desired properties. In RE, the output of the running system, modeled as a stream of events, is fed into an enforcement monitor. The monitor ensures that the stream complies with a certain property, by delaying or modifying events if necessary. This paper deals with *predictive* runtime enforcement, where the system is not entirely black-box, but we know something about its behavior. This *a-priori* knowledge about the system allows to output some events immediately, instead of delaying them until more events are observed, or even blocking them permanently. This in turn results in better enforcement policies. We also show that if we have no knowledge about the system, then the proposed enforcement mechanism reduces to a classical non-predictive RE framework. All our results are formalized and proved in the Isabelle theorem prover.

CCS Concepts

•General and reference → Verification; •Theory of computation → Program verification; •Software and its engineering → Software verification; Formal software verification;

Keywords

Monitoring; Runtime enforcement; Automata; Monitor synthesis

1. INTRODUCTION

Runtime enforcement (RE) is a technique [12, 6, 7] to monitor the execution of a system at runtime and ensure its compliance against a set of formal requirements. Using an enforcement monitor (EM), an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies with a property (e.g., a safety requirement). RE aims to ensure that: (i) the output sequence must comply with the property (soundness) and (ii) if the input already complies with the property, it should be left unchanged (*transparency*). We focus on *online enforcement of regular properties*. For a given property φ, we synthesize an enforcement monitor that operates at runtime. The general context is depicted in Figure 1, where an enforcement monitor is placed between an event emitter and an event receiver. The emitter and receiver execute asynchronously. An enforcement monitor takes a sequence of events σ as input and transforms it into a sequence of events o that is correct with respect to property φ. The monitor stores input events and releases them as output only when certain conditions warrant it. For example, consider a property formalizing some desired transactional behavior. Then, the monitor stores and delays some input events (not releasing them immediately) as long as the transaction is not properly completed.

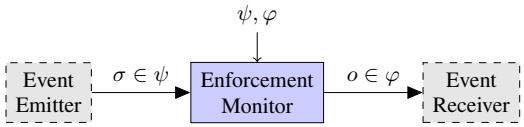

Figure 1: Enforcement monitor context.

Existing RE mechanisms (cf. [12, 6, 7, 10]) make no assumption on the input sequence σ, which can be any sequence of events over some alphabet Σ, i.e., $\sigma \in \Sigma^*$. This can be seen as considering the event emitter to be a *black box*, i.e., its behavior to be completely unknown. In this paper, we study RE for *grey box* systems, i.e., when we know something about the behavior of the event emitter. In particular, instead of letting σ range over Σ^*, we suppose that it ranges over some given property $\psi \subseteq \Sigma^*$.

In the domain of network security, one can use an EM as a firewall or a Network Intrusion Detection (NID) system to detect and prevent some attacks. Some network flows may not be interpreted in the same manner at different end-points, and may deceive firewalls and NID's. TCP/IP scrubber eliminates ambiguities from network flows enabling firewall systems to correctly predict end-host response [8]. The knowledge of the system ψ can be considered as a protocol scrubber such as a TCP/IP scrubber [8] that models well-behaved protocol behavior.

A priori knowledge of the system behavior may help to improve monitoring mechanisms. This paper is a study on how enforcement mechanisms can benefit from a model of the system already available, or built using some static-analysis techniques. For non-safety properties (in our setting, these are non prefix-closed regular lan-

*This work was partially supported by the Academy of Finland, the U.S. National Science Foundation (awards #1329759 and #1139138), and by UC Berkeley's iCyPhy Research Center (supported by IBM and United Technologies).

SAC 2016, April 04 - 08, 2016, Pisa, Italy

ISBN 978-1-4503-3739-7/16/04...$15.00

DOI: http://dx.doi.org/10.1145/2851613.2851827

guages), input events are blocked (stored in the EM's memory) until receiving all the events that allow to satisfy the desired property. If we have some knowledge of the system, we can sometimes react more quickly, for instance, if we can predict that the input will inevitably satisfy the property in the future. Prediction thus allows to avoid delaying the output of events unnecessarily, and thus provides better Quality of Service. Moreover, in some particular scenarios where the actions in the input alphabet are dependent, without prediction, blocking some events (in case of non-safety properties) can lead the system into a deadlock situation, and thus non-safety properties can not be enforced [5, 4]. For example, a requirement such as *"Every request should be followed by an acknowledgement"* can not be enforced in practice, and only having some knowledge of the system will allow to enforce such requirements. Our predictive enforcement framework allows to circumvent such situations and to enforce non-safety properties over dependent actions.

Concretely, the *predictive runtime enforcement* problem is, given two regular languages φ (modeling the property to enforce) and ψ (modeling the assumptions on system behavior), to automatically synthesize an enforcement monitor which operates as in Figure 1. As in standard (non-predictive) RE [12, 6, 10], the monitor must satisfy: (i) *soundness* (the output o satisfies φ), (ii) *transparency* (o is a prefix of the input σ). But in addition, we require a new constraint: (iii) *urgency*, which states that the observed input sequence σ must be released immediately (i.e., $o = \sigma$), if either σ already satisfies φ, or every possible extension of σ (that can be obtained from the knowledge of ψ) will result in the input satisfying φ. Urgency ensures that input events are released as soon as possible. We provide a functional definition of an enforcement monitor as a function that transforms words, and prove that it satisfies the desired properties described above. We also propose an algorithm implementing the enforcement function. Our results are formalized and proved in the Isabelle theorem prover [9]. The Isabelle theories are available at:

https://github.com/isabelle-theory/PredictiveRuntimeEnforcement

2. PRELIMINARIES AND NOTATIONS

A (finite) word over a finite alphabet Σ is a finite sequence $w = a_1 a_2 \cdots a_n$ of elements of Σ. The *length* of w is n and is noted $|w|$. The empty word over Σ is denoted by ϵ_Σ, or ϵ when clear from the context. The *concatenation* of two words w and w' is noted $w \cdot w'$. A word w' is a *prefix* of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$, and $w' \prec w$ if additionally $w' \neq w$; conversely w is said to be an *extension* of w'. The 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 $\mathcal{L}$ of Σ^*. $\mathcal{L}$ is *prefix-closed* if all prefixes of all words from $\mathcal{L}$ are also in $\mathcal{L}$: $\mathcal{L} = \{w \mid \exists w' \in \mathcal{L} : w \preccurlyeq w'\}$. Similarly, a language $\mathcal{L}$ over Σ is *extension-closed* if all extensions of all words from $\mathcal{L}$ are in $\mathcal{L}$: $\mathcal{L} = \{w \mid \exists w' \in \mathcal{L} : w' \preccurlyeq w\}$.

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

A *deterministic and complete automaton* $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ is a tuple where, Q is the set of *locations*, $q_0 \in Q$ is the initial location, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function and $F \subseteq Q$ is the set of accepting locations. In this paper we work only with deterministic and complete automata (and often simply use the term "automata"). If incomplete or non-deterministic automata arise, we can determinize and complete them first.

Function δ is lifted to words by setting $\delta(q, \epsilon) = q$, and $\delta(q, a \cdot \sigma) = \delta(\delta(q, a), \sigma)$. A word σ is *accepted* by $\mathcal{A}$ *starting from location* q if $\delta(q, \sigma) \in F$, and σ is *accepted* by $\mathcal{A}$ if σ is accepted starting from the initial location q_0. The *language* of $\mathcal{A}$ *starting from location* q is $\mathcal{L}(\mathcal{A}, q) = \{\sigma \mid \delta(q, \sigma) \in F\}$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is $\mathcal{L}(\mathcal{A}, q_0)$.

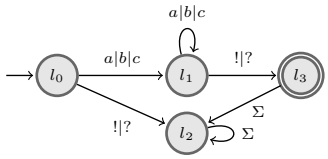

Figure 2: Property to enforce: φ

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 by $\mathcal{A} \times \mathcal{A}'$, is defined as $(Q \times Q', (q_0, q_0'), \Sigma, \delta \times \delta', F \times F')$, where $(\delta \times \delta')((q, q'), a) = (\delta(q, a), \delta'(q', a))$. The *complement* of $\mathcal{A}$ denoted $\overline{\mathcal{A}}$, is defined as $(Q, q_0, \Sigma, \delta, Q \setminus F)$. We have $\mathcal{L}(\mathcal{A} \times \mathcal{A}', (q, q')) = \mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}', q')$ and $\mathcal{L}(\overline{\mathcal{A}}, q) = \Sigma^* \setminus \mathcal{L}(\mathcal{A}, q)$.

A *regular property* is a language accepted by an automaton. In the sequel, we consider only regular properties and we refer to them as just properties. Safety (resp. co-safety) properties are sub-classes of regular properties [1]. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). Formally, safety (resp. co-safety) properties are the prefix-closed (resp. extension-closed) regular languages. An automaton $\mathcal{A} = (Q, q_0, \Sigma, \delta, F)$ is a *safety* automaton if $\forall a \in \Sigma, \forall q \in Q \setminus F : \delta(q, a) \notin F$, and a *co-safety* automaton if $\forall a \in \Sigma, \forall q \in F : \delta(q, a) \in F$.

3. MOTIVATING EXAMPLE

As motivating example, we consider the enforcement of file format requirements. Consider a scenario where an application writes to a file, using multiple write operations. At the end of the sequence of writes, the file must obey a required format. The required format constraint might not hold in the middle of the sequence of writes (so, the property is not prefix-closed).

Let us now consider a specific requirement. Consider a simple application (say application 1) that allows to write a string containing characters from the set $\{a, b, c\}$. We also have special end-of-string characters $\{!, ?\}$: the string should end with one of these characters, which cannot occur elsewhere in the string. The string is valid only if this required format condition holds. The automaton in Figure 2 defines this requirement. Its alphabet is $\Sigma = \{a, b, c, !, ?\}$. Location l_0 is initial, and the only accepting location is l_3.

Consider another application (say application 2) that makes use of application 1. If we have no knowledge about the sequence of write operations application 2 will perform (where each write operation writes a character), then the EM must buffer all the writes without saving to disk until one of the special characters is received. Once it receives a special character, it can "flush" its buffer.

Input ψ_1: Suppose that we have some knowledge of application 2, and we know what strings it can produce. Consider the automaton in Figure 3 modeling strings that application 2 can output (that will be given as input to the EM). So, we now know that the input that the EM receives is three characters (each of them belonging to $\{a, b, c\}$) ending with the special character !. Thus, the input sequences that the EM will receive are $\psi = \{abc!,\ aac!,\ \ldots\}$. Suppose that the input is $\sigma = abc!$. Without prediction, the EM will buffer events a, b, and c in its memory until it sees an !. But

[1]Similarly to some monitoring frameworks [11, 10], we consider safety and co-safety properties over finite words.

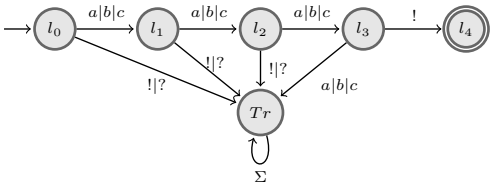

Figure 3: Possible input sequences: ψ_1

with prediction, each event will be output (written) immediately after it is read.

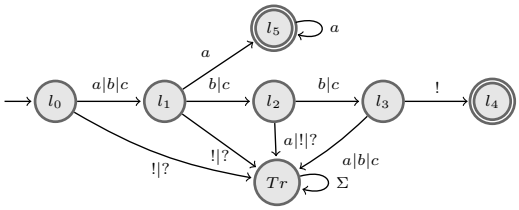

Figure 4: Possible input sequences: ψ_2

Input ψ_2: In the predictive setting, it is not always possible to output events immediately, and in some situations we may require to buffer some events in the memory of the EM. For example, instead of ψ_1, consider ψ_2 (defined by the automaton in Figure 4) defining possible input sequences. Here, if the second character is "a", then the third character should also be an "a", and a special character at the end is not necessary for such strings. So, here when the EM sees the first character, it cannot output it immediately. It has to wait until it receives the second character, and if the second character is not an "a", then it outputs the first character followed by the second character. If the third character is not an "a", it can be output immediately (without waiting for a special character).

4. PREDICTIVE RUNTIME ENFORCEMENT

In this section, we formalize the predictive runtime enforcement problem. Roughly speaking, the purpose of enforcement monitoring is to read some (possibly incorrect) sequence produced by a running system (input to the enforcement monitor), and to transform it into an output sequence that is correct w.r.t. a property φ that we want to enforce. At an abstract level, an enforcement monitor can be seen as a function that transforms words. An enforcement function for a given property φ takes as input a word over Σ and outputs a word over Σ that belongs to φ.

Now in the predictive case, instead of considering Σ^* as the language of possible inputs, we consider $\psi \subseteq \Sigma^*$, that defines the set of possible sequences that the EM receives at runtime as input. ψ can be obtained from an abstract model of the system, or can be built from some knowledge we can extract from an existing system using static-analysis.

Similar to enforcement monitoring mechanisms in [12, 6, 7, 10], several constraints are required on how an enforcement function transforms words. Our EM cannot insert (or suppress) events, and cannot change their order, and is allowed to block when a violation is detected. The notions of soundness and transparency are similar to the non-predictive case [6, 12], where soundness expresses that the output must satisfy property φ, and transparency generally aims at preventing the input sequence from being modified unnecessarily.

In our predictive setting, we introduce an additional requirement called *urgency*, expressing that if the input sequence received so far does not satisfy the property, it is still released as output immediately if all possible input events that the EM will receive in the future, will allow to satisfy φ.

Formally, given properties $\varphi, \psi \subseteq \Sigma^*$, an enforcement function $E_{\psi,\varphi} : \Sigma^* \to \Sigma^*$, should satisfy the following constraints:

Soundness

$$\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \neq \epsilon \implies E_{\psi,\varphi}(\sigma) \in \varphi \qquad \textbf{(Snd)}$$

Transparency

$$\forall \sigma \in \Sigma^* : E_{\psi,\varphi}(\sigma) \preccurlyeq \sigma \qquad \textbf{(Tr1)}$$

$$\forall \sigma \in \Sigma^* : \sigma \in \varphi \implies E_{\psi,\varphi}(\sigma) = \sigma \qquad \textbf{(Tr2)}$$

Monotonicity

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \preccurlyeq \sigma' \implies E_{\psi,\varphi}(\sigma) \preccurlyeq E_{\psi,\varphi}(\sigma') \qquad \textbf{(Mo)}$$

Urgency

$$\forall \sigma \in \Sigma^* : (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi) \implies E_{\psi,\varphi}(\sigma) = \sigma \qquad \textbf{(Ur)}$$

A function satisfying these constraints is called a *predictive enforcement monitor*. As we shall show later in Section 5, for any φ, ψ, a predictive EM always exists (Theorem 1), and can be computed using the algorithm described in Section 6.

Before giving more intuition about the constraints above, let us note that soundness and transparency are similar to non-predictive setting, except that here we restrict soundness to input words that belong to ψ.

Soundness: (**Snd**) means that for any input word belonging to ψ, if the output of the EM is not empty (ϵ), then it must satisfy φ.

Note that the condition that the output be non-empty is unavoidable. The output of the EM may be ϵ, and ϵ may not belong to the language accepted by some properties such as some non-safety properties. For example, ϵ does not belong to the language accepted by the automaton in Figure 2. If instead we had formalized soundness as $\forall \sigma \in \psi : E_{\psi,\varphi}(\sigma) \in \varphi$, then if the output for some non-safety property is ϵ, then this soundness condition cannot be satisfied. For example, let the automaton in Figure 2 define the property φ we want to enforce, and the automaton in Figure 4 define the set of input sequences. The sequence baa is a valid input sequence. But, none of the prefixes of this sequence (including ϵ) satisfy the property φ, and the output of the EM will be ϵ in such cases.

Transparency: (**Tr1**) expresses that, the output of the EM should be a prefix of the input. This constraint corresponds to the fact that the EM is allowed only to block events, but it is not allowed to insert (or suppress) events, and also cannot change their order.

(**Tr2**) expresses that, if the input already satisfies the property, then the output should be equal to the input. (**Tr1**) is not enough to ensure that, as for some properties (safety for example), blocking everything (producing ϵ output for any input sequence) will satisfy both (**Snd**) and (**Tr1**) constraints.

Monotonicity: (**Mo**) expresses that the output of the EM for an extended input word σ' of an input word σ, extends the output produced by the EM for σ.

Urgency: (**Ur**) expresses that an input word σ should be output immediately ($E_{\psi,\varphi}(\sigma) = \sigma$) if we know that any continuation of σ which can be generated by ψ will at some point satisfy φ.

Constraints (**Mo**) and (**Ur**) are related to online behavior of the EM, releasing events as output as soon as possible.

LEMMA 1. *(**Tr2**) is a consequence of (**Ur**):* $(\mathbf{Ur}) \implies (\mathbf{Tr2})$.

REMARK 1. *When we have no knowledge about the system (i.e., $\psi = \Sigma^*$), soundness and transparency constraints reduce to non-predictive case. Urgency in this case becomes equivalent to (***Tr2***).*

LEMMA 2. *When $\psi = \Sigma^*$, (***Ur***) is equivalent to (***Tr2***).*

We also expect that, if $\psi \subseteq \varphi$ (i.e., if every behavior that can be generated by the system already satisfies φ), then the EM should immediately output everything that it receives.

LEMMA 3. $\psi \subseteq \varphi \implies (\forall \sigma \in \Sigma^* : E_{\psi,\varphi}(\sigma) = \sigma)$.

5. FUNCTIONAL DEFINITION

In this section, we provide a definition of an enforcement function that incrementally builds the output. This functional definition presents an abstract view, describing how to transform an input word according to the property φ. We also prove that this functional definition satisfies all the constraints of an enforcement mechanism.

DEFINITION 1 (ENFORCEMENT FUNCTION). *Given properties $\varphi, \psi \subseteq \Sigma^*$, the enforcement function is $E_{\psi,\varphi} : \Sigma^* \to \Sigma^*$, and is defined as:*

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

where:

- $\kappa_{\psi,\varphi}(\sigma)$ is defined as:

$$\kappa_{\psi,\varphi}(\sigma) = (\forall \sigma_{\text{con}} \in \Sigma^* : \sigma \cdot \sigma_{\text{con}} \in \psi \implies \exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma_{\text{con}} \wedge \sigma \cdot \sigma' \in \varphi)$$

- $\text{store}_{\psi,\varphi} : \Sigma^* \to \Sigma^* \times \Sigma^*$ is defined as:

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

$$\text{store}_{\psi,\varphi}(\sigma \cdot a) = \begin{cases} (\sigma_s \cdot \sigma_c \cdot a, \epsilon) & \text{if } \kappa_{\psi,\varphi}(\sigma_s \cdot \sigma_c \cdot a), \\ (\sigma_s, \sigma_c \cdot a) & \text{otherwise} \end{cases}$$

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

The enforcement function takes a word over Σ as input, and produces a word over Σ as output. The function store takes a word over Σ as input, and computes a pair of words over Σ. The first component of the output of function store (extracted by Π_1) is the output of the enforcement function.

The first element of the output of the function store is a prefix of the input that will be the output of the enforcement function (the property is satisfied by this prefix followed by any continuation of the input including ϵ); the second element is the suffix of the input which the enforcer cannot output yet. The function store is defined inductively: initially, for an empty input, both elements are empty; if σ is read, $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$, and another new event $a \in \Sigma$ is observed, there are two possible cases based on whether function $\kappa_{\psi,\varphi}$ returns `true` or not.

The function $\kappa_{\psi,\varphi}$ takes a word over Σ as input and returns a Boolean as output. This function tests the hypothesis of the urgency constraint, (**Ur**). $\kappa_{\psi,\varphi}$ returns `true` if for every continuation σ_{con} such that $\sigma \cdot \sigma_{\text{con}} \in \psi$, if there is a prefix σ' of σ_{con} such that $\sigma \cdot \sigma' \in \varphi$. Thus, if the sequence σ provided as input to this function satisfies φ, then this condition will be satisfied since for every continuation σ_{con}, ϵ is a prefix of σ_{con} such that $\sigma \cdot \epsilon \in \varphi$. The function $\kappa_{\psi,\varphi}$ returns `false` if the input sequence σ does not satisfy φ, and there is a continuation of σ that will not allow to satisfy φ.

REMARK 2. *When $\psi = \Sigma^*$, following Lemma 2, $\kappa_{\psi,\varphi}(\sigma)$ can be simplified, and defined as $\kappa_{\psi,\varphi}(\sigma) = (\sigma \in \varphi)$.*

Lemma 4 introduces some properties of the enforcement function, and auxiliary function $\kappa_{\psi,\varphi}$.

LEMMA 4. *For all $\sigma, \sigma', \sigma_s, \sigma_c \in \Sigma^*$ we have*

1. $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c) \implies \sigma = \sigma_s \cdot \sigma_c$
2. $E_{\psi,\varphi}(\sigma) \neq \varepsilon \implies \kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$
3. $\kappa_{\psi,\varphi}(\sigma) \wedge \sigma \preccurlyeq \sigma' \implies \sigma \preccurlyeq E_{\psi,\varphi}(\sigma')$
4. $\sigma \in \varphi \implies \kappa_{\psi,\varphi}(\sigma)$

Property 1 of Lemma 4 states that for any input sequence σ, if $\text{store}_{\psi,\varphi}(\sigma) = (\sigma_s, \sigma_c)$, the concatenation of the two output words of $\text{store}_{\psi,\varphi}$ which is $\sigma_s \cdot \sigma_c$ is equal to the input word σ.

Property 2 of Lemma 4 states that for any input sequence σ, if the output of the enforcement function $E_{\psi,\varphi}(\sigma)$ is not empty, then $\kappa_{\psi,\varphi}(E_{\psi,\varphi}(\sigma))$ holds. This means that the sequence that is released as output $E_{\psi,\varphi}(\sigma)$, will certainly be extended in the future to satisfy φ.

Property 3 of Lemma 4 states that for any two sequences σ and σ', if $\kappa_{\psi,\varphi}(\sigma)$ holds and if σ is a prefix of σ', then σ is also a prefix of the output of the enforcement function for σ'.

Property 4 of Lemma 4 states that for any input sequence σ, if σ belongs to property φ, then the function $\kappa_{\psi,\varphi}$ for input σ returns true.

THEOREM 1. *Given two properties ψ, and φ, the enforcement function $E_{\psi,\varphi}$ as per definition 1 satisfies the (***Snd***), (***Tr1***), (***Tr2***), (***Ur***) and (***Mo***) constraints.*

REMARK 3 (SAFETY PROPERTIES). *Using knowledge of the input and predicting future extensions of the input is useful when the property φ is not prefix closed. In case if the property φ is a safety property (i.e. prefix closed), then having knowledge of the input has no advantage (as the enforcer does not need to buffer any event in its memory, also in the case when $\psi = \Sigma^*$).*

6. ENFORCEMENT ALGORITHM

In Section 5, we provided an abstract view of our enforcement monitoring mechanism, defining it as a function that transforms words. However, it is not immediate to see how this function can be implemented. In particular, how the component function $\kappa_{\psi,\varphi}$ can be implemented is not straightforward. In this section, we provide algorithms that implement $\kappa_{\psi,\varphi}$, as well as the overall enforcement functions.

Let automaton $\mathcal{A}_\varphi = (Q_\varphi, q_\varphi, \Sigma, \delta_\varphi, F_\varphi)$ define property $\varphi = \mathcal{L}(\mathcal{A}_\varphi, q_\varphi)$, and automaton $\mathcal{A}_\psi = (Q_\psi, q_\psi, \Sigma, \delta_\psi, F_\psi)$ define property $\psi = \mathcal{L}(\mathcal{A}_\psi, q_\psi)$. We recall that φ models the property that we want to enforce, and ψ models the property of possible input sequences.

We devise an on-line algorithm, with input $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$, which is an infinite loop that waits for input events (letters of the alphabet). We know that any input sequence that we get satisfies ψ eventually. An iteration of the algorithm is triggered by an input event. If the sequence of events obtained already followed by the current event does not satisfy $\kappa_{\psi,\varphi}$ then we hold this event. Otherwise we output all events held by earlier iterations. We start by implementing function $\kappa_{\psi,\varphi}$.

Implementation of $\kappa_{\psi,\varphi}$: We first introduce an automaton $\mathcal{B}_\varphi$ based on $\mathcal{A}_\varphi$. Let $\mathcal{B}_\varphi = (Q_\varphi, q_\varphi, \Sigma, \delta'_\varphi, F_\varphi)$, where δ'_φ is defined as

$$\delta'_\varphi(q, a) = \begin{cases} \delta_\varphi(q, a) & \text{if } q \notin F_\varphi \\ q & \text{otherwise} \end{cases}$$

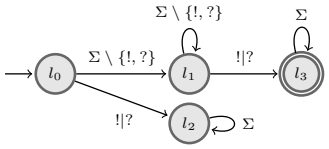

Figure 5: Automaton $\mathcal{B}_\varphi$

In $\mathcal{B}_\varphi$ we retain all the transitions in $\mathcal{A}_\varphi$ that are from non-accepting locations. Any transition from an accepting location in $\mathcal{A}_\varphi$ is directed to the same accepting location. Thus, in automaton $\mathcal{B}_\varphi$, we will not have transitions from accepting to non accepting locations.

LEMMA 5. *If $q \in Q_\varphi$ then*

$$\forall \sigma \in \Sigma^* : \sigma \in \mathcal{L}(\mathcal{B}_\varphi, q) \iff (\exists \sigma' \in \Sigma^* : \sigma' \preccurlyeq \sigma \wedge \sigma' \in \mathcal{L}(\mathcal{A}_\varphi, q))$$

Intuitively, a word σ is accepted by $\mathcal{B}_\varphi$ starting from $q \in Q_\varphi$ if it is an extension of a word accepted by $\mathcal{A}_\varphi$ starting also from q. We can see also that the property $\mathcal{L}(\mathcal{B}_\varphi, q)$ is a co-safety property (i.e., extension-closed).

EXAMPLE 1. *Let $\mathcal{A}_\varphi$ be the automaton in Figure 2. In Figure 5, we can see the automaton $\mathcal{B}_\varphi$ that we obtain from $\mathcal{A}_\varphi$. The only transition in $\mathcal{A}_\varphi$ from an accepting to a non-accepting location is the transition from location l_3 to location l_2. In automaton $\mathcal{B}_\varphi$ this transition is replaced with a self-loop in location l_3. All the other transitions that are in $\mathcal{A}_\varphi$ remain in $\mathcal{B}_\varphi$.*

THEOREM 2. *If $\sigma \in \Sigma^*$, $p = \delta_\psi(q_\psi, \sigma)$, and $q = \delta_\varphi(q_\varphi, \sigma)$ then $\kappa_{\psi,\varphi}(\sigma) \iff \mathcal{L}(\mathcal{A}_\psi \times \overline{\mathcal{B}_\varphi}, (p, q)) = \emptyset$*

Theorem 2 shows that testing $\kappa_{\psi,\varphi}(\sigma)$ is equivalent to checking emptiness of a regular language.

Enforcement algorithm: Let us now see the algorithm in detail, that requires automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$ as input. Algorithm Enforcer (see Algorithm 1) is an infinite loop that scrutinizes the system for input events. In the algorithm, p holds the current state of automaton $\mathcal{A}_\psi$ and q holds the current state of $\mathcal{A}_\varphi$. Initially p, q are assigned the initial states of automata $\mathcal{A}_\psi$ and $\mathcal{A}_\varphi$, respectively. Sequence σ_c corresponds to σ_c in the functional definition, and contains the sequence of events that are already received, and not released as output yet. Automaton $\mathcal{C}$ is the product of automata $\mathcal{A}_\psi$, and $\overline{\mathcal{B}_\varphi}$. Primitive await_event is used to wait for a new input event. Function release takes a sequence of events, and releases them as output of the enforcer.

The algorithm proceeds as follows. The memory σ_c is initialized to ϵ, and the current state information of $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ are initialized with their initial states. It then enters an infinite loop where it waits for an input event. Upon receiving an action a, the current states of $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ are updated, with the state that we reach in these automata, from their current state, reading action a. Later, the algorithm checks whether the language accepted by the automaton $\mathcal{C}$ from state (p, q) is empty. If the language accepted by the automaton $\mathcal{C}$ from state (p, q) is empty, then all the events that are in the memory of the enforcer σ_c followed by the event received a is released as output, and the memory of the enforcer is emptied (σ_c is set to ϵ). Otherwise the event a is added to the memory of the enforcer.

LEMMA 6. *If $\sigma = a_1 \cdots a_n$ is the sequence of events received so far by the enforcement algorithm, and if $\sigma_1, \ldots, \sigma_k$ are the sequences released by the algorithm for σ, then*

$$E_{\psi,\varphi}(\sigma) = \sigma_1 \cdot \cdots \cdot \sigma_k \text{ and } \sigma = E_{\psi,\varphi}(\sigma) \cdot \sigma_c$$

Algorithm 1 Enforcer

```
1:  σ_c ← ϵ
2:  p, q ← q_ψ, q_φ
3:  C ← A_ψ × complement(B_φ)
4:  while true do
5:      a ← await_event()
6:      p, q ← δ_ψ(p, a), δ_φ(q, a)
7:      if L(C, (p, q)) = ∅ then
8:          release(σ_c · a)
9:          σ_c ← ϵ
10:     else
11:         σ_c ← σ_c · a
```

where σ_c corresponds to σ_c in the algorithm, equivalent to σ_c in the definition of $E_{\psi,\varphi}$.

Lemma 6 states that for input σ, if we concatenate the output sequences released by the enforcement algorithm, it will be equal to the output of the enforcement function $E_{\psi,\varphi}(\sigma)$. The input sequence σ is equal to the output of the enforcement function $E_{\psi,\varphi}(\sigma)$ followed by the sequence in the memory of the enforcer σ_c.

REMARK 4 (COMPLEXITY). *The predictive runtime enforcement method has an off-line and an on-line component. In particular, the product automaton $\mathcal{C}$ computed in line 3 of Algorithm 1 can be computed off-line, before the actual on-line monitoring starts. In fact, the test for emptiness in line 7 of the algorithm can also be computed off-line, for every possible pair of states (p, q) in the product (how to check emptiness is well-known in automata theory). Then the results can be stored in a table with size the number of states in the product state space, i.e., the product of the states in $\mathcal{A}_\psi$ and in $\mathcal{B}_\varphi$. This results in quadratic space complexity, but constant time complexity for the on-line emptiness check. The 1-step reaction implemented in line 6 can also be done in constant time, by storing the transition tables of the two automata (these can be stored separately for each automaton, therefore requiring less space than the product). Overall, this gives a constant time on-line complexity for Algorithm 1.*

7. RELATED WORK

Runtime enforcement was initiated by of Schneider [12] and has been extensively studied since. According to how a monitor is allowed to correct the input sequence, several enforcement models have been proposed. Security automata [12] focus on safety properties, and block the execution as soon as an illegal sequence of actions (not compliant with the property) is recognized. Suppression automata [7] allow to suppress events from the input sequence. Insertion automata [7] allow to insert events to the input sequence. Edit-automata [7] or so-called generalized enforcement monitors [6] allow to perform any of these primitives. In all these approaches, the system is considered as a black-box. In our approach, we make use of available knowledge about the system, and we do not allow to suppress or insert events.

Recently, Bloem et al. [1] presented a framework to synthesize enforcement monitors for reactive systems, called *shields*, from a set of safety properties. This work focuses on reactive systems, and it is not possible to block actions and to release them later (or to halt

the system). The shield must always act instantaneously (upon erroneous input, some output must be produced instantaneously). In some cases, when a property violation is unavoidable, the shield allows deviation for k consecutive steps. In case if a second violation occurs within k steps, then the shield enters a *fail-safe* mode, where it ensures only correctness, and no longer minimizes deviation. In our approach, when it is not possible to act instantaneously, we allow to buffer input events. Moreover, we release some events as output only after being sure that the property will be satisfied eventually with subsequent output events.

Another recent approach by Dolzehnko et al in [4] introduces Mandatory Result Automata (MRAs). MRAs extend edit-automata by refining the input-output relationship of an enforcement mechanism and thus allow a more precise description of the enforcement abilities of an enforcement mechanism. In order to handle such scenarios, their approach makes use of knowledge about the actions and their effect on the monitored system (i.e., the input alphabet is split into actions and results). Moreover, the MRA model assumes synchronizable actions (i.e., after receiving an action another action cannot be received until the previous action returned a result). In our approach, we consider actions that are transmitted between (asynchronous) event emitter and receiver and hence do not consider the effect of actions.

All the previously mentioned approaches do not consider any model of the system (system is considered as a black-box). In [13], Zhang et al. propose predictive semantics for runtime verification, enabling verification monitor to foresee property satisfaction or violation before the observed execution satisfies or violates it.

Some recent work by Chabot et al. [3] uses knowledge of the program to extend enforcement. In their approach, the monitor's enforcement power is extended by giving it access to statically gathered information about the program's possible behavior. The approach of [3] works for safety properties, but, as the authors explicitly state, there is no guarantee that it would work for non-safety properties. Our approach works for any regular property. Moreover, as discussed in Remark 3, having knowledge of the input has no advantage for safety properties. Furthermore, we also make use of knowledge of the system to also predict possible futures, and to output events earlier whenever possible.

Our work is related to supervisory control [2], where a new "controlled" system is obtained by composing a system with a controller in closed-loop. The controlled system must meet a given specification and does not produce illegal actions as output. In supervisory control the controller controls the system, because it feeds-back into the system, in closed loop. But in our case, the loop remains open. The enforcement monitor does not feed-back into the system. Moreover, it is not mandatory to have a model of the system in our approach. Remark 1 discusses that our constraints reduce to non-predictive case when $\psi = \Sigma^*$.

8. CONCLUSION AND FUTURE WORK

This paper extends existing work in runtime enforcement by proposing a *predictive* RE framework. The framework generalizes RE from black-box to grey-box systems, that is, systems for which some a-priori knowledge is available. We show how knowledge about a system's behavior can benefit enforcement, by allowing a monitor to anticipate ("predict") future input events and as a result become more responsive at its output. Compared to earlier works on enforcement, this is achieved by introducing an additional constraint called *urgency*. Urgency ensures that monitors react as soon as possible, often outputting events immediately after they are received, instead of buffering them indefinitely. The property to be enforced as well as the knowledge about the system are modeled as deterministic automata (i.e., regular languages). We show how to synthesize enforcement mechanisms for any regular property and provide algorithms implementing these mechanisms in polynomial memory and constant online time.

Several interesting extensions and alternatives remain to be explored in the future. There are some recent works [10] related to enforcement for real-time systems. For real-time systems, if it is possible to output earlier (using prediction), it is certainly beneficial. We believe it is important to study whether similar approach for prediction can be used for enforcing real-time properties. In the future, we also plan to extend our work experimentally, implementing the proposed algorithms, and further study the feasibility of applying our approach in some particular scenarios.

9. REFERENCES

[1] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: Runtime enforcement for reactive systems. In *TACAS*, volume 9035 of *LNCS*. Springer, 2015.

[2] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] H. Chabot, R. Khoury, and N. Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers & Security*, 30(4):194–207, 2011.

[4] E. Dolzhenko, J. Ligatti, and S. Reddy. Modeling runtime enforcement with mandatory results automata. *Int. J. Inf. Sec.*, 14(1):47–60, 2015.

[5] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Information Systems Security*, volume 5352 of *LNCS*, pages 41–55. Springer Berlin Heidelberg, 2008.

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

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

[8] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proceedings IEEE INFOCOM 2000, Israel*, pages 1381–1390, 2000.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

[11] G. Rosu. On safety properties and their monitoring. *Sci. Ann. Comp. Sci.*, 22(2):327–365, 2012.

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

[13] X. Zhang, M. Leucker, and W. Dong. Runtime verification with predictive semantics. In *NASA Formal Methods - 4th International Symposium*, volume 7226 of *LNCS*, pages 418–432. Springer, 2012.