You Should Better Enforce than Verify

Yliès Falcone

You should Better Enforce than Verify*

Yliès Falcone
Ylies.Falcone@inria.fr

INRIA, Rennes - Bretagne Atlantique, France

Abstract. This tutorial deals with runtime enforcement and advocates its use as an extension of runtime verification. While research efforts in runtime verification have been mainly concerned with detection of misbehaviors and acknowledgement of desired behaviors, runtime enforcement aims mainly to circumvent misbehaviors of systems and to guarantee desired behaviors. First, we propose a comparison between runtime verification and runtime enforcement. We then present previous theoretical models of runtime enforcement mechanisms and their expressive power with respect to enforcement. Then, we overview existing work on runtime enforcement monitor synthesis. Finally, we propose some future challenges for the runtime enforcement technique.

Runtime verification [1, 2] is a well established technique which consists in using a monitor to supervise, at runtime, the execution of an underlying program against a set of expected properties. A monitor is a decision procedure with an output function (*e.g.*, a state machine when dealing with regular properties) processing (step by step) an execution sequence of the monitored program, and producing a sequence of verdicts (truth values of a truth-domain) indicating fulfillment or violation of a property. Whilst the detection might sometimes be a sufficient assurance for some systems, the occurrence (resp. non-occurrence) of property violations (resp. validations) might be unacceptable for others.

Runtime enforcement [3–6] of the desired property is a possible solution to ensure expected behaviors and avoid misbehaviors. Within this technique the monitor not only observes the current program execution, but also modifies it. It uses an internal memorisation mechanism, in order to ensure that the expected property is fulfilled: it still reads an input sequence but now produces a new sequence of events in such a way that the property is enforced. The precise and formal relation between input and output sequences is usually ruled by two constraints: soundness and transparency. From an abstract point of view, those constraints entail the monitor to minimally modify the input sequence in order to ensure the desired property. When the program behaves well, the enforcement monitor lets the program execute with the least influence. If the program behavior is about to exhibit a deviation w.r.t. the expected property, the monitor uses its internal memorization mechanism to prevent the misbehavior.

Practical applications of runtime enforcement. There have been many practical applications of the theory of runtime enforcement (*e.g.*, in [7–9] for program safety, or in [10, 11] for access control policies). Most of them are built on

* A longer version with more results and examples is available on the author's webpage.

Schneider's model of security automata. Although in this tutorial we will see an ideational difference between enforcement and verification, in practice there is not always a clear distinction between these disciplines. As so, even early runtime verification frameworks were often designed to, say, "execute some code" when a property is violated; hence modifying the initial program execution. For instance, when a property gets violated:

– JPAX [12], RMOR [13] allow to specify call-back functions that get called;
– Temporal Rover [14] allows to specify a bunch of code to be executed;
– MOP [15] augments monitors with exception handlers.

Nevertheless, reactions to errors are seldom used or at least lacks a systematic and formal study. Furthermore, it is clear that preventing bad behaviors would be more desirable than providing reactions to them ("better safe than sorry").

Tutorial outline. This tutorial focuses on the efforts towards building a theory of runtime enforcement which is, as we believe, emerging as a new activity. We advocate its use as an important complementary activity to runtime verification.

1 Underlying concepts

Given an alphabet E, a sequence σ on E is a total function $\sigma : I \to E$ where I is either the interval $[0, n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself. The empty sequence is denoted by ϵ. We denote by E^* the set of finite sequences over E and by E^ω the set of infinite sequences over E. $E^* \cup E^\omega$ is noted E^∞. We will assume some familiarity with the notions of sequence, prefix, and continuation. We will use $\sigma_{\ldots n}$, for $n \in \mathbb{N} \setminus \{0\}$, to denote the prefix of σ of length n.

Execution sequences. In runtime verification and enforcement techniques, as we are not aware of the program specification, the monitored program is often regarded as a generator of sequences. Thus, the runtime activity focuses on a restricted alphabet Σ_c of concrete events or operations the program can perform. Such sequences can be made of *e.g.*, resource-access events on a secure system, or kernel operations on an operating system. In a software context, these events may represent a relevant subset of instructions (*e.g.*, variable modifications or procedure calls). These operations determine the truth value of properties. Thus, in order to compare program's executions with the property, these concrete events should be abstracted in a finite set of *abstract events* Σ_a. This abstraction is an underlying correspondence $\Sigma_c \leftrightarrow \Sigma_a$, mapping every occurrence of a concrete event to the occurrence of an abstract event[1]. To simplify notations, in this tutorial we will talk uniformly about *execution sequences*, and use a unified alphabet Σ. Execution sequences, *i.e.*, possibly non-terminating runs, range over Σ^∞.

Policies vs properties. As often referred in the verification literature, a property is a set of *single* execution sequences, *i.e.*, a property partitions the set of possible execution sequences. Schneider [3] distinguishes properties from policies. A policy is defined over sets of execution sequences, *i.e.*, a policy partitions

[1] This is exactly the purpose of program instrumentation (cf. Section 2.1). Note also that the problem might be slightly more complex when dealing with *parametric events*, events that also depend of concrete execution values (see. [16] for instance).

the set of sets of execution sequences. Properties thus represent a subset of the set of policies. Only properties are suitable for a monitoring approach since they can be decided on single executions; through a predicate applying on execution sequences *in isolation*. On the contrary, policies which are not properties cannot be monitored since they would require information from other executions. For instance [3], forbidding information flow from two variables in a program is a policy and not a property since checking it would require many executions to determine if values are correlated. Moreover, in this tutorial, as we are dealing with runtime techniques, we will consider only properties defined on linear executions, excluding specific properties defined on branching execution sequences [17]. Runtime frameworks have considered properties on finite, infinite, or both finite and infinite sequences. We will note Π the property under scrutiny and $\Pi(\sigma)$ when the sequence σ belongs to Π.

1.1 Classification of properties

In the validation community, two classifications of properties have been mainly used: the *Safety-Liveness* and the *Safety-Progress* classifications.

The Safety-Liveness dichotomy. Noticing that different properties lead to different kinds of proofs on programs, Lamport suggested in [18] that two classes of properties should be distinguished:

safety [18] properties stating that something bad does not happen (*e.g.*, deadlock-freedom, partial correction, FIFO ordering);

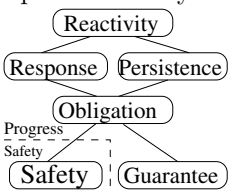
liveness [19] properties stating that a good thing eventually happens (*e.g.*, starvation-freedom, program termination).

From an abstract point of view, the difference between these properties is as follows. When safety properties are falsified it is always by a finite sequence. However, liveness properties cannot be falsified by finite sequences. That is to say, for a liveness property, any finite sequence is the prefix of an infinite one satisfying the property. For more results detailing the organization of properties within this class, we refer the reader to [19–22].

The Safety-Progress hierarchy. Pnueli et al. introduced the Safety-Progress classification of properties [23, 24], as a hierarchy between regular (linear time) properties defined as sets of *infinite* execution sequences. Unlike the Safety-Liveness dichotomy, the Safety-Progress classification is a hierarchy, and provides a finer-grain classification in a uniform way according to four views [25]: a language-theoretic one (seeing properties as sets of sequences), a logical one (seeing properties as LTL formulas), a topological one (seeing properties as open or closed sets), and an automata one (seeing properties as accepted words of Streett automata [26]). Connections between the various views endow this classification with means to translate and see a given property differently.

The Safety-Progress classification first defines basic classes over infinite execution sequences. Classes are informally defined as follows. *Safety* properties are the properties for which whenever a sequence satisfies a property, *all its prefixes* satisfy this property. *Guarantee* properties are the properties for which whenever a sequence satisfies a property, *there are some prefixes* (at least one) satisfying

this property (*e.g.*, total correctness, program termination). *Response* properties are the properties for which whenever a sequence satisfies a property, *an infinite number of its prefixes* satisfy this property (*e.g.*, success of all processes entering critical section or weak fairness). *Persistence* properties are the properties for which whenever a sequence satisfies a property, *all but finitely many* of its prefixes satisfy this property (*e.g.*, entering nominal regime).



Furthermore, two extra classes can be defined as (finite) Boolean combinations (union and intersection) of basic classes. *Obligation* properties are combinations of safety and guarantee properties (*e.g.*, exceptions). *Reactivity* properties are combinations of response and persistence properties (*e.g.*, strong fairness). This latest is the most general class containing all linear temporal properties [23]. See [25, 27] for more details.

2 Runtime verification vs runtime enforcement

In this section, we compare runtime verification and runtime enforcement. We first give an abstract picture of runtime verification and its main concepts. These concepts are mostly shared with runtime enforcement. Second, we introduce runtime enforcement and exhibit differences between the two fields.

2.1 Runtime verification

We shall now introduce runtime verification at an abstract level. For more details, the reader may refer to surveys [1, 2]. A candidate definition of "runtime verification" may be formulated as follows:

Definition 1 (Runtime Verification). *Runtime Verification is the discipline of computer science dedicated to the analysis of system executions (possibly leveraged by static analysis) by studying specification languages and logics, dynamic analysis algorithms, system instrumentation, and system guidance.*

However, the following definition has been the most admitted one [2]:

"Runtime verification is the discipline [...] that allows to determine whether a run of a system satisfies or violates a given correctness property."

This definition leaves aside the topic of program guidance that runtime verification took into account early in its scope [28]. However, we believe that this definition is representative of the research efforts in the past decade: determining how a run of the system under scrutiny gives information about a property.

Flavors of runtime verification. Two kinds of approaches are usually distinguished in runtime verification [29]:

Detection of concurrency errors: Debugging is hard to achieve on multi-threaded systems due to the large numbers of possible behaviors and the difficulty to establish causality between events. Runtime verification techniques for concurrency errors extract information from the run of the system in order to determine if such transient errors may happen on other executions (even if the current execution exhibited no errors). For instance, several methods and tools were proposed to detect data races (*e.g.*, [30, 31]), deadlocks (*e.g.*, [32]), or atomicity errors (*e.g.*, [33]).

Verification of user-provided specifications: It consists in checking whether or not the system satisfies a given specification. Several approaches were proposed from verification of simple assertions at a single location in the program to the verification of temporal assertions at several locations in the program. We refer to [1] for a study and a classification of existing approaches.

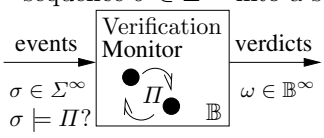
Basic concepts. As one can notice from examining Definition 1, and as several authors [2, 4, 34] pointed out, runtime verification has been *only concerned with sequence recognition*. Let us now elaborate more on the basic principles of runtime verification by depicting its ingredients.

Trace. In order to analyze a program at runtime, its concrete execution needs to be abstracted. In this perspective, the program under scrutiny is instrumented so as to produce a sequence of concrete events, a *trace*. An hypothesis is that the vocabulary of concrete events Σ_c should match with the vocabulary Σ_a in which the property is expressed. Program instrumentation then consists in inserting code at relevant places in the program to capture the occurrence of events in Σ_c and associate each of them with an event in Σ_a. The various locations in the program, where events are picked up, are named *locations*. Determining these locations relies on an analysis of the program, either manual or automatic. When manual, it consists in manually inserting monitor's code in relevant places in the program. When automatic, the instrumentation relies on an analysis that can be either static, dynamic, or both. Several approaches for program instrumentation were considered (*e.g.*, manually in [12], with Aspect-Oriented Programming [35], or byte-code insertion [36]). All these methods share the common objective to be simple, efficient, with limited impact to program's performance. Moreover, instrumentation may be realized on source code or on object code.

The monitor and its placement. Once settled, the trace is fed to one of the central concept in runtime verification: the *verification monitor*. There are various alternatives for monitor placement wrt. the program [37]. Usually the monitor runs in the same memory space as the program: *inline* placement. In this case, the monitor's code can be inserted within program's code either at observation points or by routine calls. In the second case, the monitor is placed in another memory space: *outline* placement (*e.g.*, in a thread or different process).

The monitor may also analyze the program in different ways, in a *lock-step* manner or a *posteriori*, *i.e.*, a verdict is either incrementally produced (*online* analysis) or once the program is terminated (*offline* analysis).

The monitor's purpose. The monitor behavior amounts to translate an execution sequence $\sigma \in \Sigma^\infty$ into a sequence of verdicts $\omega \in \mathbb{B}^\infty$, for a truth domain $\mathbb{B}$.



A monitor, for a property Π, behaves as a function $[\![\Pi]\!]_{\mathbb{B}}(\cdot) : \Sigma^\infty \to \mathbb{B}^\infty$ that provides an evaluation $\omega = [\![\Pi]\!]_{\mathbb{B}}(\sigma)$ for an execution sequence σ in the truth-domain $\mathbb{B}$ under consideration. Thus, one of the problems to be addressed is that each partial evaluation $[\![\Pi]\!]_{\mathbb{B}}(\sigma_{\ldots n}) = \omega_{\ldots n}$ of a *finite* sequence should not only give some relevant information on $\Pi(\sigma_{\ldots n})$, but also possibly on $\Pi(\sigma)$. In this context, the principle expressing whether or not it is worth monitoring a property, *i.e.*, *monitorability*, get raised several definitions.

What is monitorable - definitions of monitorability. The first characterization of monitorable properties was given by Viswanathan and Kim in [38]. Monitorable properties were characterized as a strict subset of safety properties. The authors showed that, due to the undecidability of some problems, a verification monitor is limited by some computability constraints. Monitorable properties are precisely defined as the safety decidable properties[2].

Pnueli *et al.* gave a more general notion of monitorable properties [39] relying on the notion of verdict determinacy for an *infinite* sequence.

Definition 2 (Monitorability [39]). *Considering a finite sequence $\sigma \in \Sigma^*$, a property $\Pi \subseteq \Sigma^\infty$ is negatively determined (resp. positively determined) by an execution sequence σ if σ and* each of its possible extension *does not satisfy (resp. does satisfy) Π. Then, Π is σ-monitorable, i.e., monitorable after reading σ, if σ has an extension s.t. Π is negatively or positively determined by this extension. Finally, Π is monitorable, if it is σ-monitorable for every $\sigma \in \Sigma^*$.*

The idea is that it becomes unnecessary to continue the execution of a Π-monitor after reading σ if Π is not σ-monitorable. In [40], Bauer et al. gave a first under-approximation of monitorable properties following this definition. They noticed that, in the Safety-Liveness classification, safety and co-safety[3] properties are monitorable according to this definition. Later in [41, 27], Falcone et al. tackled the question of monitorability within the Safety-Progress classification of properties. They established a characterization of monitorable properties as a super-set of obligation properties. Furthermore, they provided a syntactic criterion on Streett automata to determine whether or not the property recognized by an automaton is monitorable.

Noticing that the classical definition of monitorability may lead to inconsistencies, Falcone et al. proposed an *alternative* definition of monitorability [27, 41]. Indeed, following the classical definition of monitorability, for some obligation properties, some correct and incorrect execution sequences would not be distinguishable. They proposed a definition of monitorability, parameterized by a truth-domain $\mathbb{B}$, allowing to discard properties leading to ambiguities in $\mathbb{B}$.

Definition 3 (Monitorability [41]). *A property Π is said to be monitorable with the truth-domain $\mathbb{B}$ iff $\forall \sigma_{good} \in \Pi, \forall \sigma_{bad} \notin \Pi : [\![\Pi]\!]_{\mathbb{B}}(\sigma_{good}) \neq [\![\Pi]\!]_{\mathbb{B}}(\sigma_{bad})$.*

A property Π is monitorable wrt. the truth-domain $\mathbb{B}$, if it possible to distinguish correct from incorrect sequences within this truth domain. In other words, a property is monitorable, for a truth-domain, if it is possible to build a monitor that would not produce the same verdict for incorrect and correct sequences.

Synthesis of runtime verification monitors. Generally, runtime verification monitors are generated from LTL-based specifications, as seen in [15, 42]. Alternatively, ω-regular expressions have been used as a basis for generating monitors,

[2] A non-decidable safety property is a safety property for which the test used to decide whether a given sequence belongs to the property is not computable.

[3] A property Π is a co-safety property if its negation $\neg\Pi$ is a safety property.

as for example in [43]. To the author's knowledge, RuleR [44] is the system accepting the most expressive specification formalism. In RuleR, specifications are written as a set of rules and are then translated into an automaton-like language. An exhaustive list of works on monitor synthesis is far beyond the scope of this tutorial. We refer to [1, 2, 28] for more information on this topic.

Summary. All in all, runtime verification is a technique mainly used to detect expected or unexpected behaviors of a program at runtime. It consists in instrumenting the underlying program in order to be able to observe relevant events. These events are then fed to a decision procedure, a monitor, that states a verdict regarding property fulfilment or violation.

2.2 Runtime enforcement

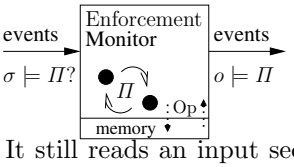
Runtime enforcement [3–6] is an extension of runtime verification that aims at answering the following questions, which are often left unanswered during a runtime verification process[4]:

- What happens when the property is violated ?
- Is it possible to *prevent* program's misbehaviors?

We propose a definition of runtime enforcement:

Definition 4 (Runtime Enforcement). *Runtime enforcement is technique dedicated to* ensure *that a run of a system satisfies a given desired property.*

Basic concepts. Runtime verification and runtime enforcement share many concepts together. The concepts of trace, monitor placement, previously presented in Section 2.1, still apply for runtime enforcement. The main conceptual differences lie in the monitor and his purpose.



The used mechanism is an *enforcement monitor*. It shares the same features with a verification monitor. In particular, it is also dedicated to a property Π, but it is augmented with a memorisation mechanism $\mathcal{M}$. It still reads an input sequence $\sigma \in \Sigma^\infty$ but outputs a new sequence $o \in \Sigma^\infty$. To do so, the monitor is endowed with a set Op of enforcement primitives that, by operating on the memorisation mechanism $\mathcal{M}$, are used to suppress or insert actions using the memory content and the current input, *i.e.*, each $op \in Op$ is a function $op : \mathcal{M} \times \Sigma^* \to \mathcal{M} \times \Sigma^*$. The upshot is that the monitor behaves as a function $[\![\Pi]\!]_{Op}(\cdot) : \Sigma^\infty \to \Sigma^\infty$ providing $o = [\![\Pi]\!]_{Op}(\sigma)$ when input σ.

Property Enforcement. The relation between input and output sequences should adhere the two following constraints that were enunciated in the work of Schneider, Bauer, Ligatti, and Walker:

soundness: the output sequences should be correct wrt. the property;
transparency: the input correct sequences should not be modified.

[4] Reaction to specification violation was originally in the scope of runtime verification [28]. Our point is that not much attention has been given to perform reactions in a completely formal and systematic way.

Thus, the enforcement monitor and its use of the memorization mechanism should be designed so as to guarantee those constraints. According to how an enforcement monitor transforms input sequences, several definitions of property enforcement[5] were proposed [4, 45, 34, 46]. We shall now present them with the unified view of an enforcement monitor as a function that transforms sequences.

Definition 5 (Property enforcement). *An enforcement monitor dedicated to a property Π, abstracted as a function $[\![\Pi]\!]_{Op}(\cdot) : \Sigma^\infty \to \Sigma^\infty$ is said to enforce Π conservatively when (1), precisely when (2), delayed-precisely when (3), effectively wrt. the equivalence relation $\approx$ when (4); where (1), (2), (3), (4) are defined, for all $\sigma \in \Sigma^\infty$, as follows:*

$$\exists o \in \Sigma^\infty : [\![\Pi]\!]_{Op}(\sigma) = o \wedge \Pi(o) \tag{1}$$

$$(1) \wedge \Pi(\sigma) \Rightarrow \sigma = o \wedge \forall i < |\sigma| : \quad [\![\Pi]\!]_{Op}(\sigma_{\dots i}) = \sigma_{\dots i} \tag{2}$$

$$(1) \wedge \Pi(\sigma) \Rightarrow \sigma = o \wedge \forall i < |\sigma|, \exists j \leq i : [\![\Pi]\!]_{Op}(\sigma_{\dots i}) = \sigma_{\dots j} \tag{3}$$

$$(1) \wedge \Pi(\sigma) \Rightarrow \sigma \approx o \tag{4}$$

An enforcement monitor enforces a property:

- *conservatively* when it adheres only to soundness;
- *precisely* when it follows soundness, transparency, and it produces outputs in a lock-step manner with the input sequence and stops outputting actions as soon as the current input deviates from the property;
- *delayed-precisely* when it follows soundness, transparency, and it produces outputs in a lock-step manner with the input sequence and it can suppress actions and later insert them (when becoming correct again);
- *effectively* when it follows soundness and transparency related to an equivalence relation $\approx$.

Note that, when the considered equivalence relation is the equality, effective enforcement amounts to delayed-precise enforcement, except that effective enforcement relaxes the constraints on the output sequence when input an incorrect sequence which does not have any correct continuation.

In the remainder of this tutorial, we will discuss some questions presented for runtime verification in the scope of runtime enforcement. We first present the models of enforcement monitors. Then we will review known result in the study of enforceable properties that corresponds to the study of the monitorability of properties in runtime verification. We will also present some enforcement monitor synthesis approaches.

3 Models of Enforcement Monitors

We first present the main models[6] of enforcement monitors. Then we present derived models that take memory limitation into account.

3.1 General models

We shall give a perspective on the main models of runtime enforcement monitors.

[5] Property enforcement amounts to monitorability in runtime verification.

[6] We only give informal pictures of the various models we introduce. These models will be formally presented during the tutorial presentation.

Security Automata. In his seminal work [3], Schneider introduced Security Automata (SA) as the first runtime mechanisms dedicated to property enforcement. SA are a variant of Büchi automata that execute in parallel with the underlying program. These automata are endowed with the ability to stop the underlying program as soon as a violation of the considered property is detected.

Edit-Automata. Ligatti et al. [4, 47] later introduced Edit-Automata (EA). They noticed that, by only halting the program, Schneider's SA were too restricted. According to its current input and its control state, an EA can either:

– **insert** an action (by either replacing the current input or inserting it), or
– **suppress** the current input (possibly *memorized in the control state* for later).

Variants of EA have been defined: Insertion Automata (only inserting actions), Suppression Automata (only suppressing inputs). In EA-like enforcement mechanisms, memorization of events (*i.e.*, suppression) is realized using control states.

A hierarchy of Edit-Automata. Bielova and Masacci [34] noticed that edit-automata generated by Ligatti et al. with the provided algorithm in [47] are of a restricted form. While EA have no restrictions on the order of enforcement operations they can perform, Bielova and Masacci noticed that EA generated by Ligatti's construction run their enforcement operations in such a way that, when they are input an incorrect execution sequence, they always output the longest correct prefix. Following this observation, [34] built a hierarchy of EA according to the enforcement ability they are endowed (*i.e.*, how enforcement operations can be performed). Delayed-Automata are constrained Edit-Automata that always output a prefix of their input. In other words, they can only insert previously suppressed actions. All-or-Nothing automata are a more constrained form of EA, *i.e.*, they are constrained Delayed-Automata. On each transition they can only either output all suspended events or suppress the current event. The kind of automata actually synthesized by Ligatti are named Ligatti's Automata by the authors of [34]. These automata are All-or-Nothing automata that always produce the longest correct prefix of the input.

Generic Enforcement Monitors. In [6, 46], independently from [34], Falcone et al. proposed the mechanism of Generic Enforcement Monitors (GEMs) as an alternative to EA. Contrary to EA, their memorization is realized through a specific memory mechanism completed with a set of operations. Moreover, the proposed automata differ in several points by offering novel features regarding enforcement monitoring. We recall some of them.

First, finding and encoding an enforcement mechanism using edit-automata is not an intuitive operation. As exposed in [34], synthesized automata using the transformation proposed in [48] may produce unexpected results for bad sequences. Second, compared to EA, GEMs propose a clear distinction between control states (used for property recognition) and the sequence memorization (when the current execution deviates from the property) in the memory device for potential replay (if the execution meets the property again). Edit-Automata use a potentially infinite number of control states for property recognition and

sequence memorization. Thus, even for a simple guarantee property *e.g.*, "eventually b" an edit-automaton needs an infinite number of states to memorize the potential incorrect sequence of events built on $\Sigma \setminus \{b\}$. Furthermore, one can notice that the size of an EA is hardly dependent on the vocabulary Σ under consideration. Hence such a mechanism is easier to implement, as they are given a restricted set of control states. Meanwhile, linking the proposed mechanism to their implementation is more compatible with formal reasoning. This provides more confidence in the implementation of such mechanisms.

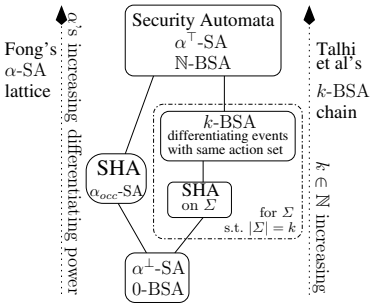
3.2 Models taking into account memory constraints

While previously presented models of enforcement monitors provide good basis for the design of enforcement mechanisms, they are supposed to be able to memorize an unbounded number of events: through a potentially infinite number of control states for EA and its derivatives, with an unbounded-size memory in GEMs. In order to get more insights on the suitability of such mechanisms for practical purposes, several models were derived.

Shallow-History automata. Fong [49] studied the effect of restraining the capacity of the runtime execution monitor using an information-based approach. Shallow History Automata (SHA) keep as history a set of access events the underlying program made and do not keep any information about the order of their arrival. Then, Fong generalized the result by using abstraction by an homomorphism α on a variant of Schneider's automata. Fong defined the notion of α-SA that intuitively abstracts previously accepted events at each transition it performs. It raised up an information-based lattice of enforcement mechanisms. At the top of this lattice are the $\alpha^{\top}$-SA keeping history of all events ($\alpha^{\top}$ distinguishes all elements of Σ^*). At the bottom of this lattice are the "memory-less" $\alpha^{\perp}$-SA, not tracking the history ($\alpha^{\perp}$ does not distinguish any sequence of Σ^*, *i.e.*, they are one-state mechanisms that prohibit a given set of events). Furthermore, the class of SA built using the abstraction function α_{occ} that captures the occurrence of events in an execution sequence corresponds to the class of SHA.

Bounded History Automata. In [50], Talhi et al. proposed Bounded Security Automata (BSA) and Bounded Edit-Automata (BEA) as restricted versions of SA and EA. These models manipulate a bounded space to record a limited history. Given a fixed size to track histories, their states represent a bounded history of valid execution execution sequences. At each performed step, the transition function of a Bounded Security Automaton abstracts the current history (state) along with the read event in order to produce the next history (state). In Bounded Edit-Automata, states are refined into pairs distinguishing the accepted prefix and the suppressed suffix of the input sequence. Thus, the transition function abstracts the concatenation of the current accepted prefix with the supressed suffix along with the read event in order to produce the new state. A BSA (resp. BEA) whose the maximum size of a history is k is said to be a k-BSA (resp. k-BEA). As expected, enforcement power of Bounded History Automata raises up with the available memory (the maximum size of a history), *i.e.*, for $k, k' \in \mathbb{N}$, when $k < k'$: k'-BSA (resp. k'-BEA) are more powerful than k-BSA (resp. k-BEA).

Summary. We report comparisons [50] related to runtime enforcement mechanisms taking into account memory limitation in the figure below.



Classes of enforcement mechanisms are represented in a hierarchical manner. For any BSA, one can find an α-SA enforcing the same property. Moreover, for any α-SA, there exists a k-BSA s.t. k is the size used to encode the results of α. Moreover, for a given alphabet Σ of size k, k-BSA are more powerful than SHA. However, note that those limited-memory models assume an infinite number of states.

4 Enforcement abilities of enforcement monitors

4.1 Power of general runtime enforcement mechanisms

Security Automata and decidable safety properties: Schneider announced that the set of precisely enforceable properties with SA is the set of safety properties. Then in [5], Hamlen et al. refined the set of enforceable properties and showed that these SA were in fact restrained by some computational limits. Indeed, as Viswanathan noticed in [51], the class of enforceable properties is impacted by the computational power of enforcement monitors. As the enforcement mechanism can implement no more than computable functions, the enforceable properties are included in the decidable ones. Hence, authors of [5] showed that the set of safety properties is a strict upper limit of the power of enforcement monitors defined as SA (the unsatisfiable safety property is also not enforceable [45]).

Edit-Automata and infinite renewal properties: The properties effectively enforced wrt. the equality by edit-automata are called *infinite renewal* properties. They have been defined, in the Safety-Liveness classification, as the properties for which every infinite valid sequence has an infinite number of valid prefixes [4]. The set of renewal properties is a super set of safety properties and contains some liveness properties (but not all). Moreover, Ligatti et al. showed that insertion and suppression automata can enforce two different proper subsets of the set of enforceable properties by Edit-Automata.

Finite edit-automata and memory-bounded properties. In [52], Beauquier et al. studied the effective enforcement ability wrt. equality of finite-state edit automata. Focusing on regular languages, they proved that enforceable properties are *memory-bounded* properties. Furthermore, they provided a syntactic criterion on generalized Muller automata [53] to determine if the property recognized by a given automaton is memory-bounded and thus enforceable; this criterion is checkable in time $O(n^2)$, where n is the number of states in the automaton.

Generic enforcement monitors and response properties. In [46], Falcone et al. showed that GEMs, instantiated with a set of enforcement operations similar to insertion and suppression, can delayed-precisely enforce the set of response properties within the Safety-Progress classification of properties. Moreover, they proved that the set of response properties is the upper-bound for any enforcement mechanism with a finite number of states (but with an unbounded memory).

4.2 Power of memory-limited runtime enforcement mechanisms

Shallow History Automata and an information-based lattice of enforceable policies. Fong showed in [49] that these automata can precisely enforce a set of properties strictly contained in the set of properties enforceable by SA. Regarding the lattice of enforcement monitors defined as α-SA, Fong showed that they give raise to a lattice on the space of all congruence relations over Σ^* which is ordered by the tracked information. Fong's classification has a practical interest by studying the effect of a practical programming constraint (limited memory) from an information point of view. It also shows that some classical security policies remain enforceable using such Shallow History Automata.

Bounded-History Automata and locally testable properties. In [50], Talhi et al. showed that there exists a taxonomy of effective enforceable properties wrt. equality based on the size limitation affecting the memory. As expected, for both BSA and BEA enforcement ability raises with the available space. Moreover, they related the enforcement ability of BHA to *locally testable properties*. Intuitively, a property is said k-locally testable if it can be recognized by an automaton with a finite memory and examining a sequence chunk of a fixed size k. According to which part of the sequence the chunk represents, several classes of locally testable properties can be defined. Intuitively, a property is prefix-testable (resp. suffix testable, prefix-suffix testable, strongly locally testable) if it is recognizable by examining a prefix (resp. suffix, both prefix and suffix, a factor) of limited size. Locally testable properties are linked to Bounded History Automata as follows:

- For BSA: prefix-closed k-prefix locally testable properties and k-strongly locally testable properties are enforceable with a memory of size k; suffix testable and prefix-suffix testable properties are not enforceable.
- For BEA: k-prefix locally testable properties are enforceable with a memory of size k; k-strongly locally testable are enforceable (no bound is given on the memory); suffix testable and prefix-suffix testable properties are not enforceable.

5 Synthesis of runtime enforcement monitors

In [54], Martinelli and Matteucci tackle the synthesis of enforcement mechanisms as defined by Ligatti. More generally, the authors consider security automata and edit-automata. The monitor is modeled by an algebraic operator expressed in CCS. The program under scrutiny is then a term $Y \rhd_K X$ where X is the target program, Y the controller program and $\rhd_K$ the operator modeling the monitor where K is the kind of monitor (truncation, insertion, suppression or edit). The desired property for the underlying system is formalized using μ-calculus. In [55], Matteucci extends the approach in the context of real-time systems.

In [48], Ligatti et al. announced a construction of Edit-Automata from finitary properties defined using a predicate on finite sequences for effective enforcement. However, as shown by [34], this construction actually affords a Ligatti automaton that delayed-precise enforce the finitary property.

In [56, 46], Falcone et al. defined class-specific transformations for the classes of enforceable properties within the Safety-Progress classification of properties.

The monitor synthesis procedures were defined from Streett automata. Besides, due to the connections between the views in the classification, their transformation indirectly provides enforcement monitor synthesis from LTL formula and properties defined using language-based operators. In [27], the authors generalized the class-specific transformations in an independent one.

In [52], Beauquier et al. defined translation of generalized pruned Muller automata [53] (for memory-bounded properties) to finite edit-automata (*i.e.*, edit-automata whose set of states is finite).

In [57], Chabot et al. synthesize Schneider's security automata from properties expressed by Rabin automata [53]. Authors provide a construction from safety properties in the general case, and for more than safety when leveraged with static information gathered from the program. However, full expressiveness of Rabin automata is left aside for non prefix-closed properties.

6 Practical problems and future challenges

We now describe some future challenges for runtime enforcement. Advances in runtime verification (see [1, 2]) will also surely benefit to runtime enforcement.

6.1 Theoretical open questions

An open question is how static information on the program can leverage runtime enforcement. As suggested in [57], having a specification of the program under scrutiny allows to slightly increase the space of enforceable properties. However, the study has been conducted only for safety properties and security automata. Thus, it remains to study how static information on the program would leverage enforcement, for others classes than safety and using more powerful mechanisms.

As exposed in Section 4, effective enforcement abilities of Generic Enforcement Monitors and Edit-Automata is unknown. Moreover, as exposed in [34], provably correct synthesis of enforcement monitors, beyond safety properties, is only effective for delayed-precise enforcement, both for Edit-Automata and Generic Enforcement Monitors. A working direction is, in this respect, to find more expressive formalisms, and associated monitor synthesis techniques.

Another working direction would be to adapt runtime verification frameworks dedicated to detection of errors on multi-threaded programs and use the principle of runtime enforcement so as to provably prevent those errors.

Soundness and transparency along with precise and delayed-precise enforcement indicate exactly how good and bad sequences should be processed by an enforcement monitor. By contrast, effective enforcement leaves the monitor free to act on bad sequences. For this purpose, one should find relevant remedial actions to be taken, *e.g.*, completion of bad sequences into good ones.

6.2 Practical challenges

A current working direction is to make the runtime enforcement technique more able to cope with practical limitations in order to deal with largescale examples. In particular it is likely that not all events produced by an underlying program can be freely *observed* and/or *controlled* by the enforcement mechanisms. Moreover, regarding the objective of limiting the resources consumed by the monitor,

it might be interesting to study how to store in memory only an *abstraction* of the observed sequence of events for effective enforcement and a suitable equivalence relation. From a theoretical point of view, this means to define enforcement up to some *abstraction preserving trace equivalence relations*.

Similarly, it would be of interest to study the notion of enforcement when weakening the transparency constraint. In this case, the most general form of edit-automata and our generic EMs could be used. Their complete enforcement potentials remain to be studied. This perspective would involve to define other relations between the input and the output sequences; and thus define other enforcement primitives so as to enforce properties automatically. It seems to us that such alternative constraints should be motivated by practical needs.

Finally, most of the practical and effective approaches to runtime enforcement have been performed using security automata. Proposing a framework solving practical implementation issues and staging the most expressive forms of runtime enforcement mechanisms would certainly be an achievement.

Acknowledgement. This tutorial is partially built on previously published material by the author and his colleagues J.-C. Fernandez and L. Mounier. Also, the author is much in debt to K. Havelund for many enriching discussions on runtime verification. Moreover, the author would like to thank H. Marchand, C. Morvan, and S. Pinchinat for their comments on an early version of this tutorial.

References

1. Havelund, K., Goldberg, A.: Verify your runs. Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005. Revised Selected Papers and Discussions (2008) 374–383
2. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming **78** (2008) 293–303
3. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000)
4. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transaction Information System Security. **12** (2009)
5. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Programming Lang. and Syst. **28** (2006) 175–205
6. Falcone, Y., Fernandez, J.C., Mounier, L.: Enforcement monitoring wrt. the safety-progress classification of properties. In: SAC '09: Proceedings of the ACM symposium on Applied Computing. (2009) 593–600
7. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Security monitor inlining for multithreaded java. In: Genoa: Proceedings of the 23^{rd} European Conference on ECOOP — Object-Oriented Programming. (2009) 546–569
8. Aktug, I., Dam, M., Gurov, D.: Provably correct runtime monitoring. In: FM '08: Proceedings of the 15^{th} int. symposium on Formal Methods. (2008) 262–277
9. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: NSPW '99: workhop on New security paradigms. (2000) 87–95
10. Cirstea, H., Moreau, P.E., de Oliveira, A.S.: Rewrite based specification of access control policies. Electron. Notes Theor. Comput. Sci. **234** (2009) 37–54
11. de Oliveira, A.S., Wang, E.K., Kirchner, C., Kirchner, H.: Weaving rewrite-based access control policies. In: FMSE'07: Proceedings of the ACM workshop on Formal Methods in Security Engineering. (2007) 71–80

12. Havelund, K., Rosu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods in System Design **24** (2003)
13. Havelund, K.: Runtime verification of C programs. In: TestCom'08: 20^{th} IFIP int. conf. on Testing of Software and Communicating Systems. (2008) 7–22
14. Drusinsky, D.: The Temporal Rover and the ATG rover. In: 7^{th} Int. SPIN Workshop on SPIN Model Checking and Software Verification. (2000) 323–330
15. Chen, F., Roşu, G.: MOP: An Efficient and Generic Runtime Verification Framework. In: OOPSLA'07: Object-Oriented Programming, Systems, Languages and Applications. (2007) 569–588
16. Chen, F., Rosu, G.: Parametric trace slicing and monitoring. In: TACAS'09: 15^{th} International Conference Tools and Algorithms for the Construction and Analysis of Systems. (2009) 246–261
17. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). (1990) 995–1072
18. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering **3** (1977) 125–143
19. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters **21** (1985) 181–185
20. Manna, Z., Pnueli, A.: Adequate proof principles for invariance and liveness properties of concurrent programs. Sci. Comput. Program. **4** (1984) 257–289
21. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Transaction Programming Languages and Systems **4** (1982) 455–495
22. Sistla, A.P.: On characterization of safety and liveness properties in temporal logic. In: PODC '85: Proceedings of the 4^{th} annual ACM symposium on Principles of distributed computing. (1985) 39–48
23. Manna, Z., Pnueli, A.: A hierarchy of temporal properties (invited paper, 1989). In: PODC '90: Proceedings of the 9^{th} annual ACM symposium on Principles of distributed computing. (1990) 377–410
24. Chang, E.Y., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Automata, Languages and Programming. (1992) 474–486
25. Chang, E., Manna, Z., Pnueli, A.: The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science (1992)
26. Streett, R.S.: Propositional Dynamic Logic of looping and converse. In: STOC '81: Proceedings of the 13^{th} Symp. on Theory Of computing, ACM (1981) 375–383
27. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime ? Software Tools for Technology Transfer, special issue on Runtime Verification (2010) Invited Paper, under review. Preprint as Verimag TR-2010-5.
28. Runtime Verification. http://www.runtime-verification.org (2001-2009)
29. Colin, S., Mariani, L.: Run-time verification. In: Model-based Testing of Reactive Systems. Volume 3472 of LNCS. (2005) 525–556
30. Chen, F., Şerbănuţă, T.F., Roşu, G.: jPredictor: a predictive runtime analysis tool for Java. In: ICSE'08: Proceedings of the 30^{th} International Conference on Software Engineering. (2008) 221–230
31. Bodden, E., Havelund, K.: Racer: Effective race detection using AspectJ. IEEE Transactions on Software Engineering (2009)
32. Bensalem, S., Havelund, K.: Dynamic deadlock analysis of multi-threaded programs. In: Hardware and Software Verification and Testing, 1^{st} International Haifa Verification Conference. Revised Selected Papers. (2005) 208–223
33. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL '04: Proceedings of the 31^{st} ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (2004) 256–267
34. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? In: FAST'08: 5^{th} International Workshop on Formal Aspects in Security and Trust. Revised Selected Papers. (2008) 287–301

35. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP. (1997) 220–242
36. The Apache Jakarta Project: Byte Code Engineering Library. http://jakarta.apache.org/bcel/ (2009)
37. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. on Software Engineering **30** (2004) 859–872
38. Viswanathan, M., Kim, M.: Foundations for the run-time monitoring of reactive systems - fundamentals of the MaC language. In: ICTAC'04: 1^{st} Int. Colloquium on Theoretical Aspects of Computing. Revised Selected Papers. (2004) 543–556
39. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: FM'06: Proceedings of Formal Methods. (2006) 573–586
40. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation (2009)
41. Falcone, Y., Fernandez, J.C., Mounier, L.: Runtime verification of safety-progress properties. In: RV'09: Proceedings of the 9^{th} Workshop on Runtime Verification. Revised selected Papers. (2009) 40–59
42. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Technische Universität München (2007)
43. d'Amorim, M., Roşu, G.: Efficient monitoring of ω-languages. In: Proceedings of 17^{th} Int. Conference on Computer-aided Verification (CAV'05). (2005) 364 – 378
44. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. In: RV'07: 7^{th} International Workshop on Runtime Verification. Revised Selected Papers. (2007) 111–125
45. Ligatti, J.A.: Policy Enforcement via Program Monitoring. PhD thesis, Princeton University (2006)
46. Falcone, Y., Mounier, L., Fernandez, J.C., Richier, J.L.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities (2010) under revision at Formal Methods in System Design. Preprint as Verimag TR 2008-7.
47. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: ESORICS'05 Proceedings of the 10^{th} European Symposium on Research in Computer Security. (2005) 355–373
48. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Int. Journal of Information Security **4** (2005) 2–16
49. Fong, P.W.L.: Access control by tracking shallow execution history. In: Proceedings of the 2004 IEEE Symposium on Security and Privacy. (2004) 43–55
50. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement for limited-memory systems. In: PST'06: Proceedings of the International Conference on Privacy, Security and Trust. (2006) 1–12
51. Viswanathan, M.: Foundations for the run-time analysis of software systems. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA (2000)
52. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite edit automata. Electr. Notes Theor. Comput. Sci. **229** (2009) 19–35
53. Perrin, D., Pin, J.E.: Infinite Words, Automata, Semigroups, Logic and Games. Elsevier (2004)
54. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. Electronic Notes in Theoritical Compututer Science **179** (2007) 31–46
55. Matteucci, I.: Automated synthesis of enforcing mechanisms for security properties in a timed setting. Elec. Notes in Theoritical Comp. Science **186** (2007) 101–120
56. Falcone, Y., Fernandez, J.C., Mounier, L.: Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In: ICISS'08: Proceedings of the 4^{th} International Conference on Information Systems Security. (2008) 41–55
57. Chabot, H., Khoury, R., Tawbi, N.: Generating in-line monitors for Rabin automata. In: NordSec'09: 14^{th} Nordic Conf. on Secure IT Systems. (2009) 287–301