Runtime Enforcement with Reordering, Healing, and Suppression

Yliès Falcone, Gwen Salaün

Runtime Enforcement with Reordering, Healing, and Suppression

Yliès Falcone and Gwen Salaün

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, F-38000 Grenoble France
firstname.lastname@univ-grenoble-alpes.fr

Abstract. Runtime enforcement analyses an execution trace, detects when this execution deviates from its expected behaviour with respect to a given property, and corrects the trace to make it satisfy the property. In this paper, we present new enforcement techniques that reorder actions when necessary, inject actions to the application to ensure progress of the property, and discard actions to avoid storing too many unnecessary actions. At any step of the enforcement, we provide a verdict, called enforcement trend in this work, which takes its value in a 4-valued truth domain. Our approach has been implemented in a tool and validated on several application examples. Experimental results show that our techniques better preserve the application actions, hence ensuring better service continuity.

1 Introduction

Runtime verification [1, 10, 18, 22, 25] is an alternative to traditional formal verification techniques, such as model checking, and avoids their complexity by analysing execution traces. Therefore, runtime verification has the advantage of scaling up very well without requiring a comprehensive model of the application, but at the expense of lower coverage. Runtime verification aims at verifying whether an execution trace satisfies a given correctness property. Runtime enforcement [13,20,21,24] goes beyond classic runtime verification by correcting the execution that deviates from its expected behaviour to ensure the satisfaction of a given property. To do so, a so-called *enforcement monitor* (or *enforcer* in short) accepts as input a sequence of actions and generates as output a sequence of actions respecting the property.

Existing enforcement techniques suffer from several issues. First, messages or actions involved in an execution trace may arrive to the enforcer in a different order, thus violating the property. This can occur in untimed distributed systems for instance, where it is impossible to guarantee the exact order of issued actions. Most enforcement techniques do not provide reordering strategy and discard many input actions in order to preserve the property validity. Second, the progress of the property might be prevented by the absence in the input execution trace of some specific action. A solution to ensure the property progress is to inject some expected actions to the application. Third, there is also a need of some removal strategies in order to decide when some actions need to be

stored because they are useful in the future or deleted to avoid storing too many (unnecessary) actions.

In this paper, we present new enforcement techniques that combine reordering, healing, and suppression. Such combination presents several advantages. The enforcer first avoids any sequence of output actions that invalidates the property. It also outputs as many actions received as input as possible, and ensures progress in the property thanks to healing techniques. Finally, suppression techniques avoid to store unnecessary actions.

More precisely, the enforcement techniques take as input a sequence of actions and a property in an automata-based or logic-based formalism, and ensure that the property will not be violated. Reordering is ensured by using a bag to store the actions that do not arrive in the correct order. Healing techniques allow the addition of actions and thus ensure progress of the application and of the property. Suppression mechanisms are used in two different situations: (i) to systematically remove actions that can cause the violation of the property, and (ii) to avoid the storage of too many actions in the bag used for reordering purposes. At any moment, we provide a verdict, called enforcement trend in this work, taken from a 4-valued truth domain (forever positive, currently positive, possibly positive, possibly negative). The enforcement trend becomes possibly negative if we have to make intensive use of reordering and healing techniques for avoiding property violation. Note that forever false is always avoided by our enforcement techniques. Our approach has been implemented in a tool and validated on several examples of applications and properties. Our enforcement techniques take several parameters as input that govern the triggering of healing and suppression techniques, which can be used to tune its behaviour.

Let us illustrate on a real-world example. Imagine a dispatcher receives parcels and can move them to three conveyor belts. The initial behaviour is random or arbitrary in the sense that the dispatcher moves the parcel to any belt. However, we want the dispatcher to be fairer by moving a parcel to belts one after the other in a specific order. Reordering is helpful because it can be used to store temporarily a parcel when it is not respecting the ordering strategy of the new scheduler. However, reordering is not enough, because in some cases one may wait for a parcel aimed to a given belt for too long. In that situation, healing techniques are interesting to decide to move a parcel to a specific belt even if it was not originally planned for that one. Last but not least, suppose that the original dispatcher moves parcels to a specific belt more often than to the other belts, say the first one for instance. In that case, many actions to move parcels to that line will be issued by the dispatcher and stored in the buffer. To avoid filling the buffer with these unnecessary actions we can decide to remove some of them from the buffer.

The paper is organised as follows. Section 2 defines execution traces and the formalism used in this work for specifying properties. Section 3 presents the enforcement techniques including the possible enforcement trends and the characteristics of the enforcer. Section 4 illustrates the approach on a case study.

Section 5 introduces the tool support and different experiments we carried out to validate our solution. Section 6 discusses related work. Section 7 concludes.

2 Models

We introduce the required notions of execution traces, properties, and bags.

Execution traces. We consider a finite set of actions A corresponding to the operations that can be executed by a program or application. An execution trace t is a sequence of actions over A. The concatenation of two actions α_1 and α_2 is denoted by $\alpha_1.\alpha_2$. The empty sequence is denoted by ϵ. Concatenation is extended to traces in the usual way. A trace σ is a *prefix* of a trace σ', noted $\sigma \preceq \sigma'$, if there exists a trace σ'' such that $\sigma' = \sigma.\sigma''$.

Properties. A property denotes a subset of (valid) execution traces in A^*. Considering a finite execution trace t and a property P, when $t \in P$, we say that t satisfies P. We model and define properties using finite-state automata.

Definition 1 (Property automaton). *A property automaton PA is a tuple* (S, s^0, Σ, T, va) *where:*

- S *is a (finite) set of* states *and* $s^0 \in S$ *is the* initial state*;*
- $\Sigma \subseteq A$ *is a finite set of actions called* alphabet*;*
- $T : S \times \Sigma \to S$ *is the total* transition function*;*
- $va : S \to \{green, violet, red\}$ *is the verdict function.*

Moreover, the verdict function is defined such that:

- $va(s^0) \neq red$,
- $\forall (s, \alpha, s') \in T : va(s) = green \implies va(s') = green$,
- $\forall (s, \alpha, s') \in T : va(s) = red \implies va(s') = red$.

A transition $(s_1, \alpha, s_2) \in T$ (also noted $s_1 \xrightarrow{\alpha}_T s_2$) indicates that the automaton can move from state s_1 to state s_2 by performing action α. States are associated with colors that are used to specify the satisfaction of the property: green states mean acceptance, violet states mean undetermined, and red states mean violation. We assume that the color of the initial state is not red. Property automata can be automatically generated from LTL properties according to existing monitor-synthesis techniques providing a finite-trace semantics to LTL. For instance, following [4, 8], the property automaton is in a green (red, resp.) state whenever the current trace satisfies (does not satisfy, resp.) the property and all possible extensions do (do not, resp.) satisfy the property. Colors can also be assigned by the user. In such a case, we require that the marking of states is consistent with the 3-valued semantics defined in [4]. In particular, there is no transition from red states to green nor violet states, nor from green states to violet nor red states. Moreover, a property automaton is *deterministic* and

complete. For $s \in S$, $Reach(s)$ is the set of states reachable from s with sequences over Σ, that is, the states related to s through the transitive closure of T. Moreover, for $s \in S$ and $\sigma \in \Sigma^*$, $Reach(s, \sigma)$ is the state reached from s following the transition function while reading σ. We associate action sequences with verdicts given by a property automaton: a sequence of actions $\sigma \in \Sigma^*$ is associated with the verdict of the state reached by reading σ on the property automaton: $[PA](\sigma) = va(Reach(s^0, \sigma))$.

Bags. The notion of *bag* is used by the runtime enforcer for storing actions. A bag is used to store possibly multiple occurrences of actions from a certain alphabet Σ without any order. We note $\mathbb{B}_\Sigma$ the set of bags over Σ. Function $add : \Sigma \times \mathbb{B}_\Sigma \to \mathbb{B}_\Sigma$ adds an action to a bag. Function $remove : \Sigma \times \mathbb{B}_\Sigma \to \mathbb{B}_\Sigma$ removes an instance of an action from a bag. Function $remove : 2^\Sigma \times \mathbb{B}_\Sigma \to \mathbb{B}_\Sigma$ is overloaded to define the removal of a set of actions from a buffer. Function $count : \mathbb{B}_\Sigma \to \mathbb{N}$ returns the number of actions in a bag. Function $actions : \mathbb{B}_\Sigma \to 2^\Sigma$ returns the actions stored in a bag, that is, the domain of the input bag. Predicate *empty* indicates whether a bag is empty.

3 Enforcement Techniques

In this section, we present successively the main ideas behind the enforcement techniques proposed in this paper, the details of how the enforcer works, and the formal characteristics ensured by the enforcer.

3.1 Overview

An enforcer takes as input an execution trace generated by a program or application in the form of a sequence of actions, as well as a temporal property described as a property automaton. The enforcer produces as output a sequence of actions that satisfies the property by avoiding red states, as is the case with standard enforcers as in e.g., [12,23,26]. In addition, the enforcer re-uses (previous) input actions as much as possible independently from their reception order. Any input action that is not part of the property alphabet is immediately returned as output. If the input action is part of the property alphabet: (i) if it makes the property automaton progress (there is one transition from the current state holding that action as label), the action is immediately returned as output, (ii) otherwise, the monitor needs to modify the input sequence of actions by using buffering or healing techniques. To do so, the enforcement techniques rely on three bags used by the monitor:

- a *buffer* is used to store temporarily actions that are not immediately required for making the property automaton progress;
- a *healer* stores actions that are injected to the output execution trace to ensure progress of the property automaton;
- a *well* is used to store and keep track of input actions that can only lead the property automaton to red states, and thus invalidate the property.

Moreover, the monitor takes three parameters as input. The first one (k_{heal}) corresponds to the number of actions stored in the buffer from which healing techniques are triggered. Basically, the underlying idea is to first determine whether we can make progress in the property automaton by reordering actions (and thus temporarily storing them in the buffer). However, if the buffer grows too much, we start healing by adding new actions to the system. Note that this parameter is not a bound of the buffer size. Indeed, healing ensures progress of the property automaton, but it does not ensure consumption of the actions stored in the buffer. Said differently, this parameter can be seen as a way to minimise the deviation with respect to the expected trace specified by the property. If this parameter is small, healing techniques will be triggered earlier, but at the price of injecting possibly more new actions as output.

The second parameter (k_{purge}) is optional and triggers the removal of actions in the buffer. It is a natural number corresponding to the number of action occurrences in the buffer from which we start suppressing part of them (half by default). For instance, if this parameter is fixed to 20, and at some point, 20 occurrences of some action are present in the buffer, we remove 10 of them. This parameter is useful to avoid storing too many actions in the buffer which may not be consumed.

The third parameter (k_{verd}) is related to the computation of verdicts that are called in this work *enforcement trends* since these are not definitive verdicts. At any moment of the monitoring, a truth value can be returned according to the current trace generated as output and to the current states of the different bags. We rely on a 4-valued truth domain in this work: forever positive, currently positive, possibly positive, and possibly negative. The property is forever true if a green state has been reached. The property is currently positive if the current state of the property automaton is violet, and the enforcer has not made use (yet) of reordering or healing techniques. The enforcement trend is possibly positive if the current state of the property automaton is violet, and the number of actions in the buffer and healer is below a threshold, which is the third parameter. The enforcement trend is possibly negative if the current state of the property automaton is violet, and the number of actions in the buffer and healer goes beyond the threshold.

To sum up, these three parameters are used by the enforcement mechanisms for different purposes:

- k_{heal} ($k_{\text{heal}} \geq 0$) corresponds to the number of actions stored in the buffer from which healing techniques are triggered. If $k_{\text{heal}} = 0$, healing techniques are activated from the beginning;
- k_{purge} ($k_{\text{purge}} \geq 0$) is the number of a certain occurrence of an action appearing in the buffer from which purging techniques are triggered. If $k_{\text{purge}} = 0$, purging techniques are not used at all;
- k_{verd} ($k_{\text{verd}} > 0$) is the number of actions stored in both the buffer and healer, which makes the enforcement trend go from possibly positive to possibly negative. If $k_{\text{verd}} = 1$, the enforcement trend becomes possibly negative as soon as we have one action in the buffer or in the healer.

In this work, k_{heal} and k_{verd} are computed automatically, by using the size of the property automaton alphabet or by statically analysing the property automaton. More precisely, we use for k_{heal} the length of the longest sequence or the length of the longest cycle (one iteration) in the property automaton. If there are more than k_{heal} actions in the buffer, it means we need to supplement buffering. As for k_{verd}, we choose a multiple of the size of the alphabet. For instance, if there are twice the number of elements of the alphabet in the buffer and healer, we consider we had to change quite significantly the input trace, and the enforcement trend changes from possibly positive to possibly negative. Another solution is to rely on machine learning techniques for computing these bounds by using the history of the execution trace and the decisions of the enforcer.

3.2 Enforcement Monitor

We detail how the enforcement techniques proposed in this paper work. The behaviour of the enforcer is described by a *transition system*, and requires as input three parameters k_{heal}, k_{purge} and k_{verd} as well as a property automaton $PA = (S, s^0, \Sigma, T, va)$. Let us start by defining the configurations of the enforcement monitor.

Definition 2 (Enforcement monitor configurations). *The set of configurations of the enforcement monitor is defined as* $Conf = S \times \mathbb{B}_\Sigma \times \mathbb{B}_\Sigma \times \mathbb{B}_\Sigma$. *A configuration of the enforcement monitor is a tuple* (s, b, h, w) *where* s *is a state of PA, and* b *(buffer),* h *(healer), and* w *(well) are three bags to store elements of* Σ.

The enforcement monitor takes one action from the execution trace as input, and generates none, one or several actions (at the same time) as output. More precisely, the monitor can react to an input action in different ways (see Def. 3 for a formal definition):

- *stop* if the monitor has reached a green state;
- *execute* the action as output if the action does not belong to Σ, or if the action belongs to Σ, the property automaton can execute this action in its current state, and there is no such action in the healer bag;
- *add to buffer* if the action belongs to Σ, leads to a red state, but can be used later in the property automaton (without going to a red state);
- *heal* by generating as output an action that can be executed from the current state in the property automaton. Healing techniques are triggered when adding a new action to the buffer, which makes k_{heal} to be reached;
- *remove from healer bag* if the action belongs to Σ, and there is such an action in the healer bag;
- *add to well* if the action belongs to Σ, leads to a red state, but cannot be used elsewhere in the property automaton;
- *purge buffer* when one specific action makes the buffer reach the k_{purge} bound. The monitor removes from the buffer a certain number of these actions (half by default).

Table 1. Transition rules of the enforcement monitor given an input action α.

$$\frac{\alpha \notin \Sigma}{(s,b,h,w) \overset{\alpha/\alpha}{\rightsquigarrow} (s,b,h,w)} \text{ (execute1)}$$

$$\frac{\alpha \in \Sigma \setminus h \quad s \xrightarrow{\alpha}_T s' \quad va(s') \neq red}{(s,b,h,w) \overset{\alpha/\alpha}{\rightsquigarrow} (s',b,h,w)} \text{ (execute2)}$$

$$\frac{\alpha \in \Sigma \qquad s \xrightarrow{\alpha}_T s' \qquad va(s') = red \qquad \exists s'', s''' \in Q : s'' \in Reach(s) \wedge \ va(s''') \neq red \wedge \ s'' \xrightarrow{\alpha}_T s'''}{(s,b,h,w) \overset{\alpha/\epsilon}{\rightsquigarrow} (s, add(\alpha,b), h, w)} \text{ (addtobuffer)}$$

$$\frac{\alpha \in \Sigma \cap h}{(s,b,h,w) \overset{\alpha/\epsilon}{\rightsquigarrow} (s,b,remove(\alpha,h),w)} \text{ (removefromhealer)}$$

$$\frac{\alpha \in \Sigma \quad \forall s' \in Reach(s) : s' \xrightarrow{\alpha}_T s'' \implies va(s'') = red}{(s,b,h,w) \overset{\alpha/\epsilon}{\rightsquigarrow} (s,b,h,add(\alpha,w))} \text{ (addtowell)}$$

In the following definition, we formally define the different sorts of transitions of the monitor. Note that for any input action, we apply only one of the following behaviours: stop, execute, add to buffer and eventually heal, remove from healer bag, or add to well. Correct termination is not present in this definition, because it makes the whole monitor stops in a green state. Healing techniques and buffer purge are presented aside because they do not need any input action to be executed. Triggering these two rules is possible every time an action is added to the buffer with the rule (addtobuffer). As far as healing techniques are concerned, they generate a single action as output, but this is systematically followed by a check to see whether actions from the buffer can be consumed and thus added to the output execution trace. If there are several actions that can be taken out from the buffer, the monitor always maximises this number, by executing the longest sequence of actions (random if there are several longest ones). Note that we could decide to apply several times the healing rule for ensuring a progress of several actions as ouput. However, we do not want to inject more actions as output than those executed by the system as input.

Definition 3 (Enforcement monitor). *The set of transitions of the enforcement monitor is the smallest subset of* $Conf \times \Sigma \times \Sigma^* \times Conf$ *abiding to the rules described in Table 1 and Table 2. A transition* $(s,b,h,w) \overset{\alpha/o}{\rightsquigarrow} (s',b',h',w')$ *indicates a move from configuration* (s,b,h,w) *to configuration* (s',b',h',w') *while inputting* $\alpha \in \Sigma$ *and outputting* $o \in \Sigma^*$.

Table 2. Transition rules of the enforcement monitor after adding to buffer.

$$\frac{count(b) > k_{\text{heal}} \qquad \forall i : \alpha_i \in b \qquad \beta = gen(s, PA) \qquad Reach(s, \beta.\alpha_1 \dots \alpha_n) = s'}{(s, b, h, w) \overset{\epsilon/\beta.\alpha_1.\dots.\alpha_n}{\rightsquigarrow} (s', remove(\{\alpha_1, \dots, \alpha_n\}, b), add(\beta, h), w)} \text{ (heal)}$$

$$\frac{b(\alpha) > k_{\text{purge}}}{(s, b, h, w) \overset{\epsilon/\epsilon}{\rightsquigarrow} (s, b, h, purge(\alpha, w))} \text{ (purgebuffer)}$$

Rules in Table 1 apply when a new action is received as input. Rules in Table 2 apply when when an action has been added to the buffer.

Function *gen*, used in the former definition (Table 2), generates an action that makes the property automaton progress, and thus possibly leads to the execution of additional actions as output (by taking them from the buffer). The healer never adds an action leading to a red state because our enforcement techniques systematically avoid red states. The healer does not add an action leading to a green state either, because we want the property to become true thanks to an input action. Therefore, the healer can output an action leading to a violet state or does not output anything (this is the case when outgoing transitions can lead to red and green states only).

Definition 4 (Healing). *Given a (current) state s of PA, the healer returns an action or the empty word as follows:*

$$gen(s, PA) = \begin{cases} \alpha \text{ if } s \xrightarrow{\alpha}_T s' \text{ and } va(s') = violet; \\ \epsilon \text{ otherwise.} \end{cases}$$

3.3 Enforcement Trend

An *enforcement trend* can be associated with the current configuration of the enforcement monitor. An enforcement trend has four possible values: forever positive, currently positive, possibly positive, and possibly negative. Recall that the enforcement monitor avoids that the output sequence makes the input property automaton reach red states.

Definition 5 (Enforcement trend). *The enforcement trend associated with a configuration (s, b, h, w) of the enforcement monitor is defined as:*

- *forever positive if $va(s) = green$;*
- *currently positive if $va(s) = violet$, $empty(b)$ and $empty(h)$;*
- *possibly positive if $va(s) = violet$ and $count(b) + count(h) < k_{\text{verd}}$;*
- *possibly negative if $va(s) = violet$ and $count(b) + count(h) \geq k_{\text{verd}}$.*

Note that parameter k_{verd} can be fixed in different ways: arbitrarily, using the size of the alphabet (e.g., twice or thrice the size of the alphabet) or by analysing statically the structure of the property automaton (e.g., length of the longest path to green states or length of the longest cycle if there is no green states).

3.4 Characteristics

The enforcement techniques proposed in this paper are online, untimed, and operational. Online means that the monitor takes as input a trace built from the running monitored system (as opposed to an offline postmortem trace). Untimed means that the enforcement monitor does not account from the physical time that elapses between these actions. Operational means that the provided definition describes *how* the enforcement monitor executes and can thus directly be used as a guide for the implementation.

In the rest of this section, we focus more particularly on the non-functional properties of enforcement monitors with healing. We revisit the classical characteristics of soundness, monotonicity, and transparency, taking into account the healer and the well. Moreover, we introduce two properties that stem from the addition of a healer to enforcement monitors, namely *progress* and *healing as a last resort*.

For this, we see and reason on an enforcement monitor as an enforcement function $E^{PA} : \Sigma^* \to \Sigma^* \times \mathbb{B}_\Sigma \times \mathbb{B}_\Sigma \times \mathbb{B}_\Sigma$, dedicated to a property automaton PA. The enforcement function describes the enforcement monitor as a relation between the input and the corresponding output, content of the buffer, content of the healer and content of the well. When $E^{PA}(in) = (o, b, h, w)$, it means that when the enforcement monitor inputs in (one action after the other), the overall produced output is o and the contents of the buffer, the healer, and the well are respectively b, h, and w.

In the following, we shall use the dot notation to refer to the elements in a configuration of the enforcement monitor and for $E^{PA}(in) = (o, b, h, w)$, we note $E^{PA}(in).\text{out} = o$, $E^{PA}(in).\text{buff} = b$, $E^{PA}(in).\text{heal} = h$, and $E^{PA}(in).\text{well} = w$.

An enforcement monitor is *sound*, meaning that for any input sequence, the property is not violated by the output sequence produced by the enforcement monitor.

Proposition 1 (Soundness). $\forall in \in \Sigma^* : [\![PA]\!](E^{PA}(in).\text{out}) \neq \mathit{red}$

Proof (Sketch). Soundness holds because the enforcement monitor never produces an action as output if it leads to a red state starting from the state stored in its configuration. Moreover, the state stored in its configuration is the state reached by executing the output in the property automaton.

An enforcement monitor is *monotone*, meaning that it respects the following physical constraints: the produced output cannot be undone and the actions discarded in the well are definitely lost.

Proposition 2 states that the output (sequence) of the enforcer is a growing function of the input sequence.

Proposition 2 (Monotonicity when outputting actions).
$\forall in, in' \in \Sigma^* : in \preceq in' \implies E^{PA}(in).\text{out} \preceq E^{PA}(in').\text{out}$

Proof (Sketch). Monotonicity of the output is a straightforward consequence of the fact that the output sequence of the enforcement monitor is formed by concatenating the output actions produced while reading the input sequence.

Proposition 3 similarly states that the well of the enforcer is a set where one can only add new elements.

Proposition 3 (Monotonicity when discarding actions).
$\forall in, in' \in \Sigma^* : in \preceq in' \implies E^{PA}(in).\text{well} \subseteq E^{PA}(in').\text{well}$

Proof (Sketch). Monotonicity when discarding actions is a direct consequence of the fact that actions are only discarded with rule (addtowell), which accumulates actions in the well.

An enforcement monitor is *transparent*, meaning that (i) it intervenes (by making the output sequence differ from the input) only when the input sequence violates the property, and (ii) the input actions are found either in output, in the buffer or in the well, and only the healer can generate additional actions.

Proposition 4 (Transparency).

- $\forall in \in \Sigma^* : [\![PA]\!](in) \neq red \implies E(in).\text{out} = in$
- $\forall in \in \Sigma^* :$
  $\text{actions}(E^{PA}(in).\text{out}) \cup \text{actions}(E^{PA}(in).\text{buff}) \cup \text{actions}(E^{PA}(in).\text{well})$
  $\setminus \text{actions}(E^{PA}(in).\text{heal}) = \text{actions}(in).$

Proof (Sketch). The first part of transparency holds because if the input sequence does not violate the property, then it means that it leads to a violet or a green state in the underlying property automaton. Henceforth, only rules (execute1) and (execute2) have been applied when inputting in.

The second part of transparency holds because input actions either go to output, to the buffer or to the well. Additional actions are exactly those in the healer part of the configurations. Actions created by the healer are later removed whenever they appear as input.

An enforcement monitor ensures progress, meaning that the produced output sequence keeps growing when the healer can ($|E(in).\text{buff}| \geq k_{\text{heal}}$) and should ($[\![PA]\!](E(in).\alpha) = red$) intervene.

Proposition 5 (Progress).
$\forall in \in \Sigma^*, \forall \alpha \in \Sigma :$
$|E^{PA}(in).\text{buff}| \geq k_{\text{heal}} \wedge [\![PA]\!]\left((E^{PA}(in).\text{out}).\alpha\right) = red$
$\implies |E^{PA}(in.\alpha)| > |E^{PA}(in)|$

Proof (Sketch). Progress holds because when the buffer contains more than k_{heal} actions and $[\![PA]\!]\left((E^{PA}(in).\text{out}).\alpha\right)$, only rule (heal) can apply. Function *gen* returns some action that lead to a violet state from the state reached in the property automaton after outputting $[\![PA]\!](E^{PA}(in)).\text{out}$. Such a state is necessarily violet and the action thus necessarily exists because of the constraints on state colors.

An enforcement monitor *heals as a last resort*, meaning that whenever the healer intervenes (which is witnessed by its bag being non-empty), it means that the new input action cannot be produced as output and that the healing threshold has been reached for the buffer size.

Proposition 6 (Healing as last resort).
$\forall in \in \Sigma^*, \forall \alpha \in \Sigma :$
$E^{PA}(in).\text{heal} = \epsilon \wedge E^{PA}(in \cdot \alpha).\text{heal} \neq \epsilon$
$\implies [PA](E^{PA}(in).\alpha) = red \wedge |E^{PA}(in).\text{buff}| \geq k_{\text{heal}}$

Proof (Sketch). Healing as a last resort holds because healing can (only) happen by applying rule (heal), which requires that the buffer contains more than k_{heal} actions. Moreover, the input action causing healing to happen must have been put into the buffer by previously applying rule (addtobuffer) and this led to the buffer size to exceed k_{heal}.

4 Case Study

We illustrate our approach with the example mentioned in the introduction. Suppose that a delivery company uses a dispatcher, which receives parcels from an input conveyor belt and moves them to three output conveyor belts (we could extend this example to as many belts as required). The initial behaviour of the dispatcher is to move a parcel to any belt. Assume now we want the dispatcher to be fairer by moving a parcel to the belts one after the other in a specific order. To do so, we first need to model this specification using the property automaton given in Figure 1, where we define the repetition of a fair repartition among three conveyor belts. When these actions arrive in a different order, this corresponds to an incorrect behaviour (red state). The automaton also exhibits a case of correct termination with the STOP transition going to a green state. The alphabet of the property is {BELT1, BELT2, BELT3, STOP} and the overall alphabet also consists of two other actions: PARCEL (arrival of a parcel) and PAUSE (pause of the input belt for five seconds for instance). We decide to start healing techniques when there are three actions in the buffer ($k_{\text{heal}} = 3$), which corresponds to the length of the longest sequence of actions in the automaton (without passing twice through the same state). We choose $k_{\text{verd}} = 8$, which is twice the number of elements in the alphabet of the property automaton.

Let us illustrate how our approach works in practice by using an excerpt of the application of the enforcement techniques on this example (Fig. 2), where we replace BELT1, BELT2 and BELT3 with B1, B2 and B3, respectively, for the sake of readability. Each row in the result shows an input action, the resulting output action(s), the states of buffer, healer, well if not empty, and finally the enforcement trend. At the beginning of this trace, all bags are empty and the trend is (currently) positive. The last output action was BELT2 (l.85), so we expect BELT3 to be the next action of the property automaton. However, the enforcer receives BELT1 (l.88) and BELT2 twice (l.92 and l.95), so it moves

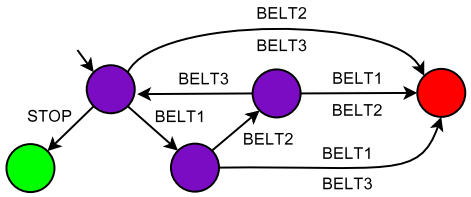


Fig. 1. Parcel Dispatcher Property Automaton

these three actions to the buffer. Then it receives BELT3 (l.99) and it can output BELT3, BELT1 and BELT2. This illustrates the use of reordering techniques.

We now look at an example of healing. At l.109, the buffer contains three actions (BELT2, BELT2, BELT3). The last output action was BELT3 (l.104), so BELT1 is required now. However, this is BELT2 that is received as input (l.110), which is a fourth action to add to the buffer, thus triggering healing techniques. The enforcement monitor decides to move a parcel to BELT1. This action is added to the healer and appears as output. This step forward in the property automaton also allows the consumption from the buffer of actions BELT2 and BELT3, explaining why we have three actions appearing as output on l.110.

Last but not least, looking at l.124 for instance, we see that there are two BELT1 in the healer, and the input action is BELT1 as well. In that situation, since the enforcer owes somehow two BELT1 actions to the application, it does not generate anything as output but just removes one BELT1 from the healer.

5 Tool Support and Experiments

In this section, we first present the implementation of the enforcement techniques. Second, we introduce different experiments carried out to evaluate our approach in terms of configuration size and enforcement trend. Finally, we compare our solution with another approach based only on reordering.

5.1 Tool Support

The enforcement techniques presented in this paper have been implemented in a prototype tool written in Python. Note that an option of the tool allows the user to make use of reordering techniques only or reordering and healing techniques together. The tool mainly consists of three modules: one for representing and manipulating property automata, one implementing several strategies for generating input actions, and one implementing the enforcement techniques.

5.2 Experiments

In our experiments, we applied our approach to several examples, from short execution traces (hundreds of actions as input) to very long traces consisting

85 - B3 → B3 B1 B2 - trend: currently positive
86 - PARCEL → PARCEL - trend: currently positive
87 - PAUSE → PAUSE - trend: currently positive
88 - B1 → - buffer: ['B1'] - trend: possibly positive
89 - PARCEL → PARCEL - buffer: ['B1'] - trend: possibly positive
90 - PAUSE → PAUSE - buffer: ['B1'] - trend: possibly positive
91 - PARCEL → PARCEL - buffer: ['B1'] - trend: possibly positive
92 - B2 → - buffer: ['B1', 'B2'] - trend: possibly positive
93 - PARCEL → PARCEL - buffer: ['B1', 'B2'] - trend: possibly positive
94 - PAUSE → PAUSE - buffer: ['B1', 'B2'] - trend: possibly positive
95 - B2 → - buffer: ['B1', 'B2', 'B2'] - trend: possibly positive
96 - PARCEL → PARCEL - buffer: ['B1', 'B2', 'B2'] - trend: possibly positive
97 - PARCEL → PARCEL - buffer: ['B1', 'B2', 'B2'] - trend: possibly positive
98 - PARCEL → PARCEL - buffer: ['B1', 'B2', 'B2'] - trend: possibly positive
99 - B3 → B3 B1 B2 - buffer: ['B2'] - trend: possibly positive
100 - PARCEL → PARCEL - buffer: ['B2'] - trend: possibly positive
101 - B2 → - buffer: ['B2', 'B2'] - trend: possibly positive
102 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
103 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
104 - B3 → B3 - buffer: ['B2', 'B2'] - trend: possibly positive
105 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
106 - PAUSE → PAUSE - buffer: ['B2', 'B2'] - trend: possibly positive
107 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
108 - B3 → - buffer: ['B2', 'B2', 'B3'] - trend: possibly positive
109 - PARCEL → PARCEL - buffer: ['B2', 'B2', 'B3'] - trend: possibly positive
110 - B2 → B1 B2 B3 - buffer: ['B2', 'B2'] - healer: ['B1'] - trend: possibly positive
111 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - healer: ['B1'] - trend: possibly positive
112 - B3 → - buffer: ['B2', 'B2', 'B3'] - healer: ['B1'] - trend: possibly positive
113 - PAUSE → PAUSE - buffer: ['B2', 'B2', 'B3'] - healer: ['B1'] - trend: possibly positive
114 - B2 → B1 B2 B3 - buffer: ['B2', 'B2'] - healer: ['B1', 'B1'] - trend: possibly positive
115 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - healer: ['B1', 'B1'] - trend: possibly positive
116 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - healer: ['B1', 'B1'] - trend: possibly positive
117 - B3 → - buffer: ['B2', 'B2', 'B3'] - healer: ['B1', 'B1'] - trend: possibly positive
118 - PARCEL → PARCEL - buffer: ['B2', 'B2', 'B3'] - healer: ['B1', 'B1'] - trend: possibly positive
119 - PARCEL → PARCEL - buffer: ['B2', 'B2', 'B3'] - healer: ['B1', 'B1'] - trend: possibly positive
120 - B3 → B1 B2 B3 - buffer: ['B2', 'B3'] - healer: ['B1', 'B1', 'B1'] - trend: possibly positive
121 - PARCEL → PARCEL - buffer: ['B2', 'B3'] - healer: ['B1', 'B1', 'B1'] - trend: possibly positive
122 - PAUSE → PAUSE - buffer: ['B2', 'B3'] - healer: ['B1', 'B1', 'B1'] - trend: possibly positive
123 - PAUSE → PAUSE - buffer: ['B2', 'B3'] - healer: ['B1', 'B1', 'B1'] - trend: possibly positive
124 - B1 → - buffer: ['B2', 'B3'] - healer: ['B1', 'B1'] - trend: possibly positive
125 - PARCEL → PARCEL - buffer: ['B2', 'B3'] - healer: ['B1', 'B1'] - trend: possibly positive
126 - B1 → - buffer: ['B2', 'B3'] - healer: ['B1'] - trend: possibly positive
127 - PARCEL → PARCEL - buffer: ['B2', 'B3'] - healer: ['B1'] - trend: possibly positive
128 - B2 → - buffer: ['B2', 'B3', 'B2'] - healer: ['B1'] - trend: possibly positive
129 - PARCEL → PARCEL - buffer: ['B2', 'B3', 'B2'] - healer: ['B1'] - trend: possibly positive
130 - PAUSE → PAUSE - buffer: ['B2', 'B3', 'B2'] - healer: ['B1'] - trend: possibly positive
131 - B1 → - buffer: ['B2', 'B3', 'B2'] - trend: possibly positive
132 - PARCEL → PARCEL - buffer: ['B2', 'B3', 'B2'] - trend: possibly positive
133 - PARCEL → PARCEL - buffer: ['B2', 'B3', 'B2'] - trend: possibly positive
134 - B2 → B1 B2 B3 - buffer: ['B2', 'B2'] - healer: ['B1'] - trend: possibly positive
135 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - healer: ['B1'] - trend: possibly positive
136 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - healer: ['B1'] - trend: possibly positive
137 - B1 → - buffer: ['B2', 'B2'] - trend: possibly positive
138 - PAUSE → PAUSE - buffer: ['B2', 'B2'] - trend: possibly positive
139 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
140 - PARCEL → PARCEL - buffer: ['B2', 'B2'] - trend: possibly positive
141 - B3 → - buffer: ['B2', 'B2', 'B3'] - trend: possibly positive
142 - PARCEL → PARCEL - buffer: ['B2', 'B2', 'B3'] - trend: possibly positive
143 - PARCEL → PARCEL - buffer: ['B2', 'B2', 'B3'] - trend: possibly positive
144 - B1 → B1 B2 B3 - buffer: ['B2'] - trend: possibly positive
145 - PARCEL → PARCEL - buffer: ['B2'] - trend: possibly positive
146 - PARCEL → PARCEL - buffer: ['B2'] - trend: possibly positive
147 - B1 → B1 B2 - trend: currently positive
148 - PAUSE → PAUSE - trend: currently positive
149 - PARCEL → PARCEL - trend: currently positive

Fig. 2. Illustration of the Enforcement Techniques on the Parcel Dispatcher

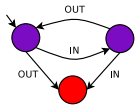
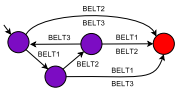
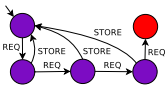
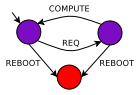
of hundreds of thousands of actions as input. The goal of these experiments is to show the configuration size and the final enforcement trend, which is maintained as much as possible in the possibly positive value. Table 3 presents the results of some representative experiments. The first five columns describe the example with a short textual description, the property automaton, the length of the simulation in terms of input actions, and the k_{heal} and k_{verd} parameters. In these experiments, k_{heal} is equal to twice the longest sequence in the property automaton and k_{verd} is equal to three times the size of the property automaton alphabet. The last four columns present the result by giving the average size of the three bags (buffer, healer, well) as well as the enforcement trend as a percentage. This trend shows for how many actions as input, the enforcement trend was currently or possibly positive. As an example, if the execution run consists of 1000 actions, and the enforcement trend is currently or possibly positive for 800 of these actions, the result is 80%. During these experiments, we chose to not purge the buffer, which corresponds to a value of 0 for k_{purge}. Each line of the table was computed by repeating 100 times the simulation. Since input traces are always different, this repetition allows us to compute more accurrate output values. Note that we do not use property automata with green states, because green states make our simulation stop, and we prefer to run it for a specific length here (third column).

First of all, we can see in the table that the enforcement trend is mostly currently or possibly positive. We will show in the next subsection how our approach compares in that aspect with similar approaches. The variability of the enforcement trend in the table (70%, 80% or 90%) is due to the variability of the actions used as input. Similarly to the enforcement trend, we can see that the average number of actions in the buffer or generated by the healer remain rather stable when the length of the trace increases. This shows that, for these examples, the approach succeeds in maintaining a positive trend without making intensive use of reordering and healing, even when the size of the execution trace increases. Note also that the number of actions in the buffer remains on average below the k_{heal} parameter, even though the buffer contents can go above k_{heal}, which is not a bound of the buffer. This happens because we never prevent adding values to the buffer, even when healing is triggered. The final row of the table (no untimely reboot) shows an example where the well is used to store reboot actions that can only lead to red states.

5.3 Comparison

In the final part of this section, we compare the results obtained with our enforcement techniques with respect to an existing enforcement approach based on action reordering and suppression [9]. We chose that work because it provides solutions similar to those proposed in this paper. More precisely, we compare three techniques: (i) our approach where reordering is used but not healing, (ii) our approach where we use both reordering and healing, and (iii) the approach presented in [9]. In a first set of experiments, we chose the two first examples in Table 3 and we apply the three techniques to these two examples. Table 4 shows

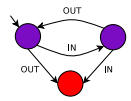
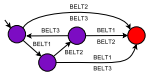
Table 3. Experimental Results

Example	PA	|Simul.|	k_h.	k_v.	|B|	|H|	|W|	Positive?
Alternating in/out		1,000	4	6	2.07	1.04	0	85.82%
		5,000			2.37	1.66	0	78.57%
		10,000			2.51	2.07	0	74.67%
		50,000			2.48	1.96	0	75.29%
		100,000			2.52	1.99	0	75.20%
		500,000			2.52	2.08	0	75.09%
Belts dispatcher		1,000	6	9	4.05	0.85	0	90.75%
		5,000			5.34	1.55	0	76.93%
		10,000			5.65	1.73	0	73.84%
		50,000			5.85	1.78	0	71.88%
		100,000			5.82	1.76	0	72.36%
No 3 req. without store		1,000	8	6	1.56	0.03	0	95.06%
		5,000			1.88	0.16	0	91.74%
		10,000			1.95	0.13	0	91.67%
		50,000			2.01	0.21	0	90.45%
		100,000			1.96	0.18	0	91.20%
No untimely reboot		1,000	4	9	2.12	0.92	25.29	94.28%
		5,000			2.43	1.94	128.12	87.13%
		10,000			2.40	1.36	257.40	91.51%
		50,000			2.55	2.24	1281.91	84.91%
		100,000			2.47	1.84	2562.46	87.77%

the results where we fixed the simulation length at 1,000 actions. The two last columns of this table shows the two main differences between these approaches, namely the number of actions appearing as output and the enforcement trend. In our approach, we do not discard any action as input (we do not reach 1,000 because there are a few actions remaining in the buffer when we stop the simulation) and use all of them while trying to maintain a positive verdict for the property automaton. In contrast, [9] favors the preservation of a correct verdict by discarding valid input actions, those that do arrive in the right order and make the verdict become false (about 10% of input actions in Table 4).

In a second set of experiments carried out on the example presented in Section 4 (belts dispatcher), we compare our two approaches (reordering only vs. reordering + healing) using execution runs of different lengths (500, 1,000, and then every 1,000 up to 10,000). For each length, we repeated the simulation 100 times to have accurrate results (one simulation instance means that we use the same trace as input for both approaches). As a result, we computed for how many actions as input, the enforcement trend was currently or possibly positive. Figure 3 shows the resulting curves for these two techniques with percentage of truth value over execution run (vertical axis) and execution run length (horizontal axis). We can see that reordering combined with healing techniques obtain

Table 4. Comparison with [9]

Approach	PA	|Simul.|	k_h.	k_v.	|Out. act.|	Positive?
Reordering		1,000	4	6	996.1	77.06%
Reordering + healing					998.3	85.82%
Pinisetty *et al.* [9]					910.4	100%
Reordering		1,000	6	9	994.2	80.12%
Reordering + healing					996.7	90.75%
Pinisetty *et al.* [9]					902.8	100%

the best results by maintaining a positive enforcement trend for a larger number of inputs (about 70-90%), whereas results for reordering only provide lower positive results (about 60-80%).

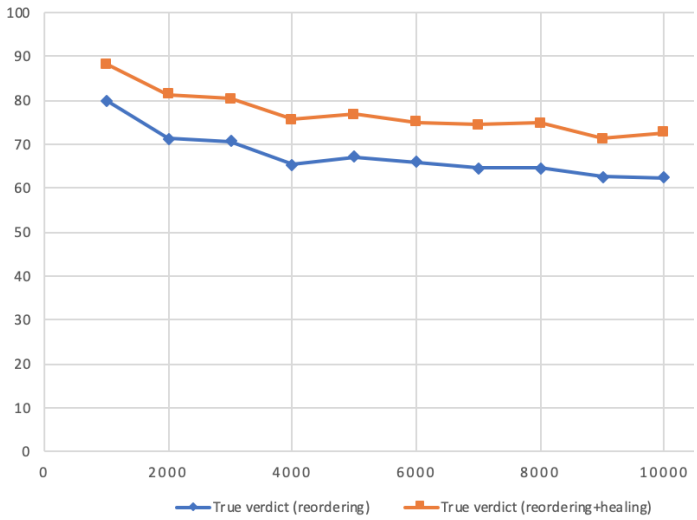


Fig. 3. Reordering Techniques only vs. Reordering + Healing Techniques (percentage of truth value on the vertical axis and execution run length on the horizontal axis)

6 Related Work

Several enforcement techniques have been proposed in the literature considering different kinds of inputs, architectures, or models. In this section, we compare our approach with closest related work. The interested reader can find comprehensive overviews of the related techniques in [11, 15].

The approach in [5] monitors distributed systems with respect to LTL properties using an alternative to the orchestration and migration approaches. Their solution relies on a choreography-based architecture where monitors are organised as a tree across the distributed system. The choreography-based decentralised monitoring is formalised and shows how to synthesise a network from an LTL formula, resulting in an algorithm working on top of an LTL network.

Regarding runtime enforcement of untimed properties, several models and frameworks have been proposed. We can mention security automata [26]; which can stop the underlying system upon property violation, edit-automata [23] and generic enforcement monitors [12] which can insert or suppress actions. Regarding runtime enforcement of timed properties, [13] provides a recent overview of related work. As an example, [9] considers runtime enforcement for timed specifications modelled as timed automata. These enforcement mechanisms work by delaying actions to match timing constraints, and suppressing actions when no delaying is appropriate, thus possibly allowing for longer executions. Enforcement mechanisms are formalised at several levels of abstraction (enforcement function, monitor, and algorithms), which facilitates the design and implementation of these mechanisms.

The approach in [2] verifies distributed systems at runtime where components communicate with monitors over unreliable channels, meaning that messages can be delayed, reordered, or even lost. The authors propose an extension of the real-time logic MTL, which provides a new three-valued semantics that is well suited for runtime verification as it accounts for partial knowledge about a system's behaviour. They also present online algorithms that reason soundly and completely about streams where actions can occur out of order.

As proposed in some existing works, e.g. [2], we tackle the reordering problem by storing and delaying messages when necessary. In addition, we complement reordering with healing that injects some new actions to ensure progress of the application while satisfying the property being analysed. Removing actions is also supported in our solution as it is in some other work, e.g., [9]. The novelty of our work resides in the combination of these three techniques (reordering, healing, suppression).

7 Concluding Remarks

We have presented new enforcement techniques, which accept as input a sequence of actions and a property automaton, and generate as output a sequence of actions that satisfies the given property. These techniques rely on three mechanisms that reorder input actions when necessary, inject new actions to the application for ensuring property progress (healing), and remove actions if they are not required or if they risk to congest the buffering system. The enforcement techniques were implemented and validated on several examples.

A first perspective of this work is to support *distributed enforcement*. This entails considering actions from several components and several input execution traces. For this, we can take inspiration from the distributed and decentralized

runtime verification approaches [3, 7, 15]. Distributed mechanisms would require to rely on synchronization mechanisms, similar to those used in choreography-based development [16, 17], to take consistent distributed decisions. Another perspective is to compute automatically system-specific or domain-specific values for the parameters k_{heal} and k_{verd} by relying on the trace history and by using machine learning techniques. Finally, we plan to apply our enforcement approach to concrete application areas. One idea in that direction is to enforce properties during the execution of BPMN processes [19], particularly for optimization purposes [6, 14].

Acknowledgements. The authors would like to thank the anonymous reviewers for their useful comments. This work was supported by the Région Auvergne-Rhône-Alpes within the "*Pack Ambition Recherche*" programme, the H2020-ECSEL-2018-IA call Grant Agreement number 826276 (CPS4EU), the French ANR project ANR-20-CE39-0009 (SEVERITAS), and the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

References

1. E. Bartocci, Y. Falcone, A. Francalanza, and G. Reger. Introduction to runtime verification. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2018.
2. D. A. Basin, F. Klaedtke, and E. Zalinescu. Runtime Verification over Out-of-order Streams. *ACM Trans. Comput. Log.*, 21(1):5:1–5:43, 2020.
3. A. Bauer and Y. Falcone. Decentralised LTL monitoring. *Formal Methods Syst. Des.*, 48(1-2):46–93, 2016.
4. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.
5. C. Colombo and Y. Falcone. Organising LTL Monitors over Distributed Systems with a Global Clock. *Formal Methods Syst. Des.*, 49(1-2):109–158, 2016.
6. F. Durán, C. Rocha, and G. Salaün. Analysis of the runtime resource provisioning of BPMN processes using maude. In S. Escobar and N. Martí-Oliet, editors, *Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers*, volume 12328 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2020.
7. A. El-Hokayem and Y. Falcone. On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. *ACM Trans. Softw. Eng. Methodol.*, 29(1):1:1–1:57, 2020.
8. Y. Falcone, J. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *Int. J. Softw. Tools Technol. Transf.*, 14(3):349–382, 2012.
9. Y. Falcone, T. Jéron, H. Marchand, and S. Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Sci. Comput. Program.*, 123:2–41, 2016.
10. Y. Falcone, S. Krstic, G. Reger, and D. Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021.

11. Y. Falcone, L. Mariani, A. Rollet, and S. Saha. Runtime failure prevention and reaction. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 103–134. Springer, 2018.
12. 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.
13. Y. Falcone and S. Pinisetty. On the Runtime Enforcement of Timed Properties. In B. Finkbeiner and L. Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 48–69. Springer, 2019.
14. Y. Falcone, G. Salaün, and A. Zuo. Semi-automated Modelling of Optimized BPMN Processes. In *Proc. of SCC'21*. IEEE, 2021.
15. A. Francalanza, J. A. Pérez, and C. Sánchez. Runtime Verification for Decentralised and Distributed Systems. In E. Bartocci and Y. Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 176–210. Springer, 2018.
16. M. Güdemann, P. Poizat, G. Salaün, and L. Ye. VerChor: A Framework for the Design and Verification of Choreographies. *IEEE Trans. Services Computing*, 9(4):647–660, 2016.
17. M. Güdemann, G. Salaün, and M. Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 238–253. Springer, 2012.
18. K. Havelund and A. Goldberg. Verify your runs. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.
19. ISO/IEC. International Standard 19510, Information technology – Business Process Model and Notation. 2013.
20. R. Khoury and S. Hallé. Runtime enforcement with partial control. In J. García-Alfaro, E. Kranakis, and G. Bonfante, editors, *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, volume 9482 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2015.
21. R. Khoury and N. Tawbi. Which security policies are enforceable by runtime monitors? A survey. *Comput. Sci. Rev.*, 6(1):27–45, 2012.
22. M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebraic Methods Program.*, 78(5):293–303, 2009.
23. J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1):2–16, 2005.
24. J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In D. Gritzalis, B. Preneel, and M. Theoharidou, editors, *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, volume 6345 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 2010.

25. C. Sánchez, G. Schneider, W. Ahrendt, E. Bartocci, D. Bianculli, C. Colombo, Y. Falcone, A. Francalanza, S. Krstic, J. M. Lourenço, D. Nickovic, G. J. Pace, J. Rufino, J. Signoles, D. Traytel, and A. Weiss. A survey of challenges for runtime verification from advanced application domains (beyond software). *Formal Methods Syst. Des.*, 54(3):279–335, 2019.
26. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.