On the Runtime Enforcement of Timed Properties [⋆]

Yliès Falcone[1] and Srinivas Pinisetty[2]

[1] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr
[2] School of Electrical Sciences, IIT Bhubaneswar, Bhubaneswar, India
spinisetty@iitbbs.ac.in

Abstract. Runtime enforcement refers to the theories, techniques, and tools for enforcing correct behavior of systems at runtime. We are interested in such behaviors described by specifications that feature timing constraints formalized in what is generally referred to as timed properties. This tutorial presents a gentle introduction to runtime enforcement (of timed properties). First, we present a taxonomy of the main principles and concepts involved in runtime enforcement. Then, we give a brief overview of a line of research on theoretical runtime enforcement where timed properties are described by timed automata and feature uncontrollable events. Then, we mention some tools capable of runtime enforcement, and we present the TiPEX tool dedicated to timed properties. Finally, we present some open challenges and avenues for future work.

Runtime Enforcement (RE) is a discipline of computer science concerned with enforcing the expected behavior of a system at runtime. Runtime enforcement extends the traditional runtime verification [12–14, 42, 43] problem by dealing with the situations where the system deviates from its expected behavior. While runtime verification monitors are execution observers, runtime enforcers are execution modifiers.

Foundations for runtime enforcement were pioneered by Schneider in [98] and by Rinard in [95] for the specific case of real-time systems. There are several tutorials and overviews on runtime enforcement for untimed systems [39, 47, 59], but none on the enforcement of timed properties (for real-time systems).

In this tutorial, we focus on runtime enforcing behavior described by a timed property. Timed properties account for physical time. They allow expressing constraints on the time that should elapse between (sequences of) events, which is useful for real-time systems when specifying timing constraints between statements, their scheduling policies, the completion of tasks, etc [5, 7, 88, 101, 102].

This tutorial comprises four stages:

1. the presentation of a taxonomy of concepts and principles in RE (Sec. 1);
2. the presentation of a framework for the RE of timed properties where specifications are described by timed automata (preliminary concepts are recalled in Sec. 2, the framework is overviewed in Sec. 3, and presented in more details in Sec. 4);
3. the demonstration of the TiPEX [82] tool implementing the framework (Sec. 5);
4. the description of some avenues for future work (Sec. 6).

⋆ This work is supported by the French national program "Programme Investissements d'Avenir IRT Nanoelec" (ANR-10-AIRT-05).

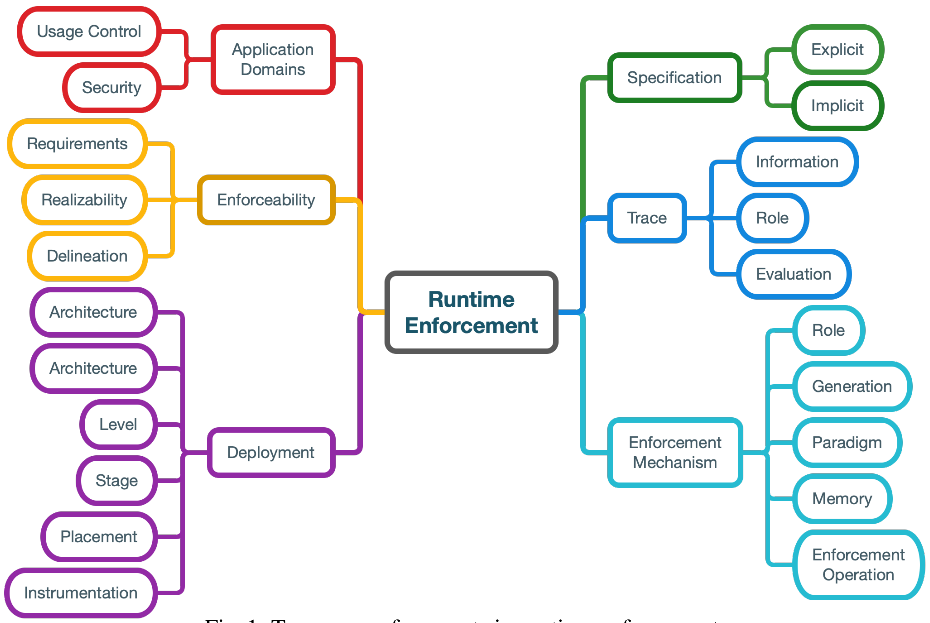

Fig. 1: Taxonomy of concepts in runtime enforcement.

1 Principles and Concepts in Runtime Enforcement

In the first stage of the tutorial, we discuss a *taxonomy* of the main concepts and principles in runtime enforcement (see Fig. 1). We refer to this taxonomy as the *RE taxonomy*. The RE taxonomy builds upon, specializes, and extends the taxonomy of runtime verification [45] (RV taxonomy). In particular, the RE taxonomy shares the notions of **specification**, **trace**, and **deployment** with the RV taxonomy. We briefly review and customize these for runtime enforcement in the following for the sake of completeness. The RE taxonomy considers the additional **enforceability** and **enforcement mechanism** parts. We also present some **application domains** where the RE principles were used.

1.1 Specification

A specification (Fig. 2) describes (some of) the intended system behavior to be enforced. It usually relies on some abstraction of the actual and detailed system behavior. A specification can be categorized as being explicit or implicit. An **explicit** specification makes the functional or non-functional requirements of the target system explicit. An explicit specification is expressed by the user using a specification language (e.g., some variant of temporal logic or extension of automata). Such specification language relies on an operational or denotational paradigm to express the intended behavior. The specification language offers modalities which allow referring to the past, present, or future of the execution. Other dimensions of a specification are related to the features allowing expressing the expected behavior with more or less details. The *time* dimension refers to the underlying model of time, being either a logical time or the actual physical time. The *data* dimension refers to whether the specification allows reasoning about any form of data involved in the program (values of variables or function parameters). An **implicit** specification is related to the semantics of the programming language of the target application system, or its programming or memory models. Implicit specifications generally capture a collection of errors that should not appear at runtime because they could lead to unpredictable behavior. Implicit specifications include security concerns (see also Sec. 1.6) such as memory safety [10, 105] where some form of memory access errors should be avoided (e.g., use after free, null pointer dereference, overflow) and the integrity of the execution (of the data, data flow, or control flow). They also include absence of concurrency errors [69] such as deadlocks, data races, and atomicity violations.

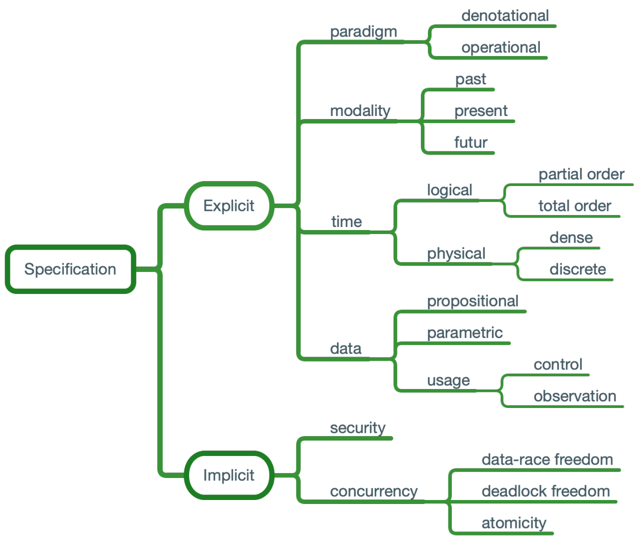

Fig. 2: Taxonomy - specification.

1.2 Trace

Depending on the target system and specification being enforced, the considered notion of trace can contain several sorts of **information** (Fig. 3): input/output from the system, events or sample states from the system, or signals. The notion of trace can play up to three different **roles**: it can be the mathematical *model* of a specification (when a set of traces defines the specification), the sequence of pieces of information from the system which is *input* to the enforcement mechanism, or the sequence of pieces of information enforced on the system which is *output* from the enforcement mechanism. In the two latter cases, the observation and imposition of the trace is influenced by the sampling on the system state, which can be triggered according to events or time. Moreover, the trace can contain more or less precise information depending on the points of control and observation provided by instrumentation. Such information can be gathered by **evaluating/abstracting** the system state at points of intervals of physical time. We refer to [74, 90] for more details on the concept of trace.

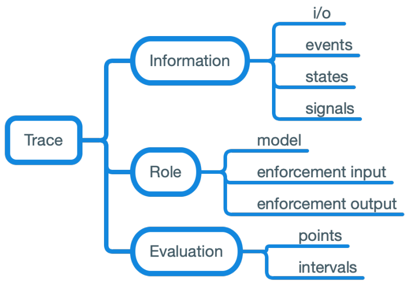

Fig. 3: Taxonomy - trace.

1.3 Enforcement Mechanism

An enforcement mechanism (EM, Fig. 4) is a mechanism in charge of enforcing the desired specification, be it either a mathematical model of the expected-behavior transformation or its realization by algorithms or routines. It is referred to by several names in the literature, e.g., enforcement monitor, reference monitor, enforcer, and enforcement mechanism. Several models of enforcement mechanisms were proposed: security automata [98], edit-automata [68] (and its variants [19]), generalized enforcement monitors [48], iteration suppression automata [21], delayers [83], delayers with suppression [44], sanitizers [104], shields [63], safety shields [107], shields for burst errors [107], and safety guards [108]. An EM reads a trace produced by the target system and produces a new trace where the specification is enforced. It acts like a "filter" on traces. This conceptualization of an EM as a(n) (input/output) filter abstracts away from its actual **role**, which can be an input sanitizer (filtering out the inputs to the target system), an output sanitizer (filtering out the outputs of the target system), or a reference monitor (granting or denying permission to the action that the system executes). An EM can be **generated** automatically from the specification (it is said to be synthesized) or programmed manually. Automatically generating an EM provides more confidence and binds it to the specification used to generate it, whereas manual generation permits programming an EM and makes room for customization. There exist several **paradigms** for describing the behavior of an EM: denotational, when an EM is seen as a mathematical function with the set of traces as domain and codomain; or operational, when the computation steps for an EM are detailed (e.g., rewriting rules, automaton, labelled transition system - LTS, or algorithm). To transform the trace, an EM can use some internal **memory** to store information from the execution. Such memory can be assumed infinite, finite, or shallow when it cannot record multiple occurrences of the same piece of information. Moreover, using data from the input trace and this memory, **enforcement operations** in an EM may transform the trace. Examples of enforcement operations include terminating the underlying target system, preventing an action from executing or altering it, executing new actions, etc.

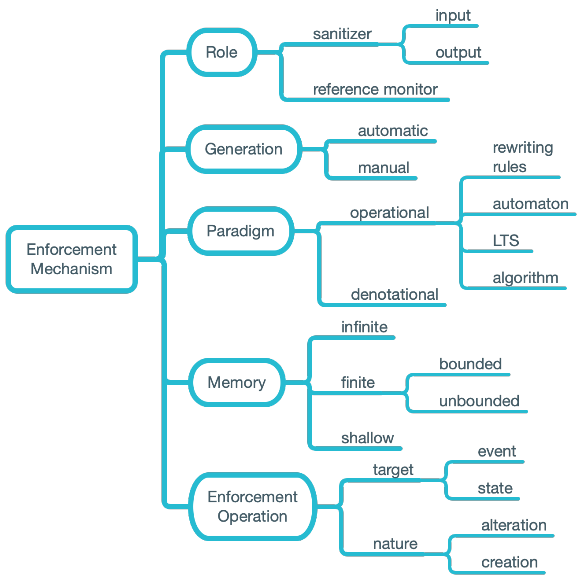

Fig. 4: Taxonomy - enforcement mechanism.

1.4 Deployment

Like deployment in runtime verification, deployment in runtime enforcement (Fig. 5) refers to how an EM is integrated within the system: its implementation, organization, and how and when it collects and operates on the trace. One of the first things to consider is the **architecture** of the system, which can be monolithic, multi-threaded or distributed. One can use a centralized EM (that operates on the whole system) or a decentralized one which adapts to the system architecture and possibly use existing communication mediums for decentralized EMs to communicate. An EM itself can be deployed at several **levels**: software, operating system or virtual machine, or hardware. The higher the level (in terms of abstraction), the more the mechanism has access to semantic information about the target system, while lower-level deployment provides the enforcement device with finer-grain observation and control capabilities on the target system. The **stage** refers to *when* an EM operates, either *offline* (after the execution) or *online* (during the execution, synchronously or asynchronously). Offline runtime enforcement (and verification) is conceptually simpler since an EM has access to the complete trace (in e.g., a log) and can thus perform arbitrary enforcement operation. On the contrary, in online enforcement, an EM only knows the execution history and decisions have to be made while considering all possible future behaviors. The **placement** refers to *where* an EM operates, either *inline* or *outline*, within or outside the existing address space of the initial system. The deployment parameters are constrained by the **instrumentation** (technique) used to augment the initial system to include an EM. Instrumentation can be software-based or hardware-based depending on the implementation of the target system. In the case of software-based instrumentation, it can operate at the level of the source (language) of the application system, its intermediate representation (IR), the binary, or using library interposition. Hardware-based instrumentation [8, 9, 75, 99] can be for instance realized by observing and controlling the system through a JTAG port and using dedicated hardware (e.g., an FPGA).

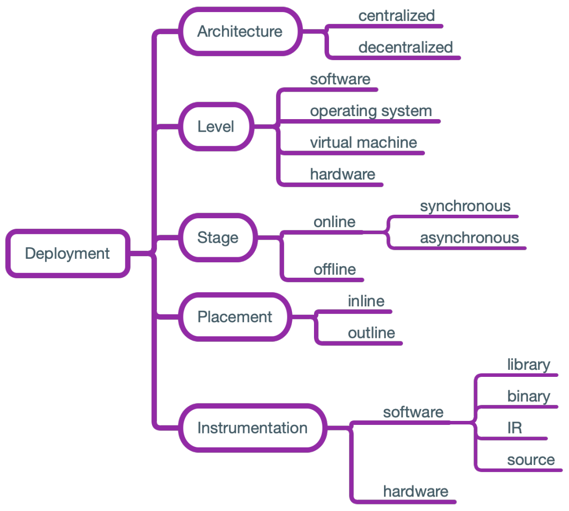

Fig. 5: Taxonomy - deployment.

Deployment challenges. Deploying an EM with the appropriate parameters raises several challenges and issues. From a bird-eye view, the challenges revolve around ensuring that an EM does not “conflict with the initial system”. We discuss this around two questions. First, *how to implement the “reference logic” where the enforcement mechanism takes important decisions regarding the system execution?* In the case where an EM is a sanitizer, deployment should ensure that all the relevant inputs or outputs go through the enforcement device. In the case where an EM is a reference monitor, the reference logic should ensure that the application actions get executed only if the monitor authorizes it. In security-sensitive or safety-critical applications, users of an enforcement framework may demand formal guarantees. There are some approaches to certify runtime verification mechanisms [3, 23, 33] and some to verify edit-automata [94], but more research endeavors are needed in these directions. Second, *how to preserve* *the integrity of the application?* As an EM modifies the behavior of the application, it should not alter its functioning by avoiding crashes, preserving its semantics, and not deteriorating its performance. For instance, consider the case where an EM intervenes by forbidding the access to some resource or an action to execute (denying it or postponing it). In case of online monitoring, an EM should be aware of the application semantics and more particularly of its control and data flows. In case of outline monitoring, there should be some signaling mechanism already planned in the application or added through instrumentation.

1.5 Enforceability

Enforceability (Fig. 6) refers to the concept of determining the specification behavior that can effectively be enforced on systems. EMs should follow some **requirements** on how they correct the behavior of systems. For instance, *soundness* refers to the fact that what is output by an EM should comply with the specification while *transparency* refers to the fact that the modification to the initial system behavior should be minimal. Additional constraints such as *optimality* can be introduced to several possible modifications that a monitor can make, according to some desired quality of service. Additionally, distances or pre-orders can be defined over valid traces for the same purpose [20, 58]. When it is possible to obtain an EM that enforces a property while complying with the requirements, the property is said to be *enforceable*, and *non-enforceable* otherwise. Enforceability of a specification is also influenced by the **realizability** of EMs. For this, assumptions are made on the feasibility of some operations of an EM. An example is when an EM memorizes input events from the target system, it should not prevent the system from functioning. Another example is when enforcing a timed specification, as the time that elapses between events matters for the satisfaction of the specification, there are assumptions to be made or guarantees to be ensured on the computation time performed by EMs (e.g., the computation time of an EM should be negligible) or on the system communication (e.g., communication overhead or reliability). Moreover, the amount of memory that an EM disposes influences how much from the execution history it can record or events it can store, and thus the enforceable properties [50, 106]. Furthermore, importantly in a timed context, physical constraints should be taken into consideration: in the online enforcement of a specification, events cannot be released by an EM before being received. The realizability of EMs can benefit from knowledge on the possible system behavior. Such knowledge can come from a (possibly partial) model of the system or static analysis. Knowledge permits upgrading an EM with predictive abilities [86] and it can thus enforce more specifications (see also [11, 85, 110] for predictive runtime verification frameworks). Another concern with enforceability is to **delineate** the sets of enforceable and non-enforceable specifications. Characterizing the set of enforceable specifications allows identifying the (possibly

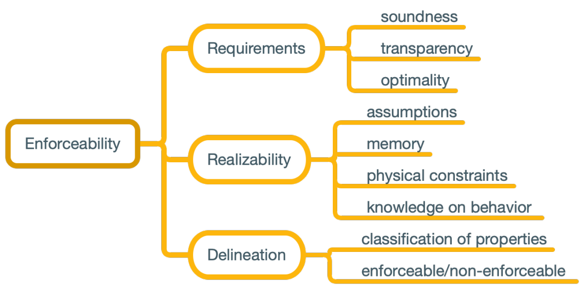

Fig. 6: Taxonomy - enforceability.

syntactically characterized) fragments of a specification language that can be enforced. For this, one can rely on existing classical classifications of properties, such as the safety-liveness "dichotomy" [4, 66, 103] or the safety-progress hierarchy [27, 71] classifications. There exist several delineations of enforceable/non-enforceable properties based on different assumptions and EMs; see e.g., [26, 48, 58, 68, 98].

1.6 Application Domains

Application domains (Fig. 7) refers to the domains where the principles of runtime enforcement are applied. We briefly refer to some applications of runtime enforcement in categories: usage control and security/privacy, and memory safety. We do not further elaborate the taxonomy for application domains since classifying security domains is subject to interpretation and most implementations of EMs for security address several flavors of security. Regarding applications for **usage control**, runtime enforcement was applied to enforce usage control policies in [73], enforcement of the usage of the Android library in [41], disabling Android advertisements in [36]. Regarding applications in the domain of **security**, runtime enforcement was applied to enforce the opacity of secrets in [46, 55, 109], access control policies in [76–78], confidentiality in [28, 53], information-flow policies [28, 49, 64, 64], security and authorization policies in [22, 38], privacy policies in [28, 56, 65], control-flow integrity in [2, 34, 52, 57, 62], and memory safety in [24, 25, 35, 100].

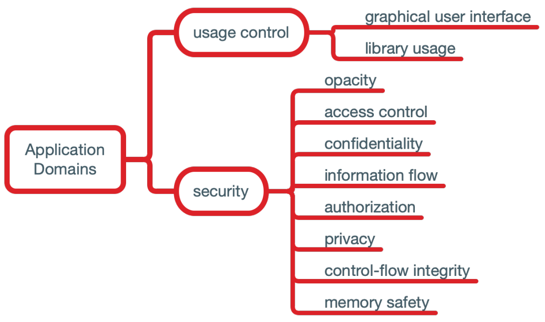

Fig. 7: Taxonomy - application domains.

2 Real-time Systems and Specifications with Time Constraints

The correctness of real-time systems depends not only on the logical result of the computation but also on the time at which the results are produced. Such systems are specified with requirements with precise constraints on the time that should elapse between actions and events. Formalization of a requirement with time constraints is referred to as a timed property. Timed automata is a formal model used to define timed properties. A timed automaton [6] is a finite automaton extended with a finite set of real valued clocks. It is one of the most studied models for modeling and verifying real-time systems with many algorithms and tools. In this section, we present the preliminaries required to formally define timed requirements and executions (traces) of a system.

2.1 Preliminaries and Notations

Untimed concepts. Let Σ denote a finite alphabet. A (finite) word over Σ is a finite sequence of elements of Σ. The *length* of a word w is the number of elements in it and is denoted by $|w|$. The empty word over Σ is denoted by ϵ_Σ, or ϵ when clear from the context. The set of all (resp. non-empty) words over Σ is denoted by Σ^* (respectively Σ^+). The *concatenation* of two words w and w' is denoted by $w \cdot w'$. A word w' is a *prefix* of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$, and $w' \prec w$ if additionally $w' \neq w$; conversely w is said to be an *extension* of w'.

A *language* over Σ is a subset of Σ^*. The *set of prefixes* of a word w is denoted by $\mathrm{pref}(w)$. For a language $\mathscr{L}$, $\mathrm{pref}(\mathscr{L}) \stackrel{\text{def}}{=} \bigcup_{w \in \mathscr{L}} \mathrm{pref}(w)$ is the set of prefixes of words in $\mathscr{L}$. A language $\mathscr{L}$ is *prefix-closed* if $\mathrm{pref}(\mathscr{L}) = \mathscr{L}$ and *extension-closed* if $\mathscr{L} \cdot \Sigma^* = \mathscr{L}$.

Timed words and timed languages. In a timed setting, we consider the occurrence time of actions. Input and output streams of enforcement mechanisms are seen as sequences of events composed of a date and an action, where the date is interpreted as the absolute time when the action is received by the enforcement mechanism.

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers, and Σ a finite alphabet of *actions*. An *event* is a pair (t,a), where $\mathrm{date}((t,a)) \stackrel{\text{def}}{=} t \in \mathbb{R}_{\geq 0}$ is the absolute time at which the action $\mathrm{act}((t,a)) \stackrel{\text{def}}{=} a \in \Sigma$ occurs.

A timed word over the finite alphabet Σ is a finite sequence of events $\sigma = (t_1,a_1) \cdot (t_2,a_2) \cdots (t_n,a_n)$, for some $n \in \mathbb{N}$, where $(t_i)_{i \in [1,n]}$ is a non-decreasing sequence in $\mathbb{R}_{\geq 0}$.

The set of timed words over Σ is denoted by $\mathrm{tw}(\Sigma)$. A *timed language* is any set $\mathscr{L} \subseteq \mathrm{tw}(\Sigma)$. Even though the alphabet $(\mathbb{R}_{\geq 0} \times \Sigma)$ is infinite in this case, previous untimed notions and notations (related to length, prefix etc) extend to timed words.

When concatenating two timed words, one should ensure that the result is a timed word, i.e., dates should be non-decreasing. This is ensured if the ending date of the first timed word does not exceed the starting date of the second one. Formally, let $\sigma = (t_1,a_1) \cdots (t_n,a_n)$ and $\sigma' = (t'_1,a'_1) \cdots (t'_m,a'_m)$ be two timed words with $\mathrm{end}(\sigma) \leq \mathrm{start}(\sigma')$, their concatenation is $\sigma \cdot \sigma' \stackrel{\text{def}}{=} (t_1,a_1) \cdots (t_n,a_n) \cdot (t'_1,a'_1) \cdots (t'_m,a'_m)$. By convention $\sigma \cdot \varepsilon \stackrel{\text{def}}{=} \varepsilon \cdot \sigma \stackrel{\text{def}}{=} \sigma$. Concatenation is undefined otherwise.

2.2 Timed Automata

A timed automaton [6] (TA) is a finite automaton extended with a finite set of real-valued clocks. Intuitively, a clock is a variable whose value evolves with the passing of physical time. Let $X = \{x_1, \ldots, x_k\}$ be a finite set of *clocks*. A *clock valuation* for X is an element of $\mathbb{R}^X_{\geq 0}$, that is a function from X to $\mathbb{R}_{\geq 0}$. For $\chi \in \mathbb{R}^X_{\geq 0}$ and $\delta \in \mathbb{R}_{\geq 0}$, $\chi + \delta$ is the valuation assigning $\chi(x) + \delta$ to each clock x of X. Given a set of clocks $X' \subseteq X$, $\chi[X' \leftarrow 0]$ is the clock valuation χ where all clocks in X' are assigned to 0. $\mathscr{G}(X)$ denotes the set of *guards*, i.e., clock constraints defined as Boolean combinations of simple constraints of the form $x \bowtie c$ with $x \in X$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathscr{G}(X)$ and $\chi \in \mathbb{R}^X_{\geq 0}$, we write $\chi \models g$ when g holds according to χ. A (semantic) state is a pair composed of a location and a clock valuation.

Instead of presenting the formal definitions, we introduce TAs on an example. The timed automaton in Fig. 8 formalizes the requirement *"In every 10 time units (tu), there cannot be more than 1 alloc action"*. The set of locations is $L = \{l_0, l_1, l_2\}$, l_0 is the initial location, l_0 and l_1 are accepting locations, and l_2 is a non-accepting location. The set of actions is $\Sigma = \{alloc, rel\}$. There are transitions between locations upon actions. A finite set of

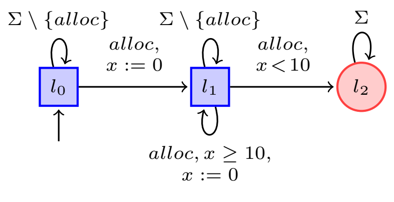
$\Sigma \setminus \{alloc\}$
$\Sigma \setminus \{alloc\}$
Σ
$alloc,$ $x := 0$
$alloc,$ $x < 10$
l_0
l_1
l_2
$alloc, x \geq 10,$ $x := 0$

Fig. 8: Example of TA.

real-valued clocks is used to model realtime behavior, set $X = \{x\}$ in the example. On the transitions, there are i) guards with constraints on clock values (such as $x < 10$ on the transition between l_1 and l_2 in the example), and ii) assignment to clocks. Upon the first occurrence of action *alloc*, the automaton moves from l_0 to l_1, and 0 is is assigned to clock x. In location l_1, if action *alloc* is received, and if the value of x is greater than or equal to 10, then the automaton remains in l_1, resetting the value of clock x to 0. It moves to location l_2 otherwise.

2.3 Partitioning the States of a Timed Automaton

Given a TA with semantic states Q and accepting semantic states Q_F, following [42], we can define a partition of Q with four subsets *good* (G), *currently good* (G^c), *currently bad* (B^c) and *bad* (B), based on whether a state is accepting or not, and whether accepting or non-accepting states are reachable or not. This partitioning is useful for runtime verification and enforcement. An enforcement device makes decisions by checking the reachable subsets. For example, if all the reachable states belong to the subset B, then it is impossible to correct the input sequence anymore (in the future). If the current state belongs to the subset G, then any sequence will lead to a state belonging to the same subset and thus the enforcement device can be turned off. This partition is also useful to classify timed properties and for the synthesis of enforcement devices.

Formally, Q is partitioned into $Q = G^c \cup G \cup B^c \cup B$, where $Q_F = G^c \cup G$ and $Q \setminus Q_F = B^c \cup B$, and:

- $G^c = Q_F \cap \text{pre}^*(Q \setminus Q_F)$ is the set of *currently good* states, that is the subset of accepting states from which non-accepting states are reachable;
- $G = Q_F \setminus G^c = Q_F \setminus \text{pre}^*(Q \setminus Q_F)$ is the set of *good* states, that is the subset of accepting states from which only accepting states are reachable;
- $B^c = (Q \setminus Q_F) \cap \text{pre}^*(Q_F)$ is the set of *currently bad* states, that is the subset of non-accepting states from which accepting states are reachable;
- $B = (Q \setminus Q_F) \setminus \text{pre}^*(Q_F)$ is the set of *bad* states, that is the subset of non-accepting states from which only non-accepting states are reachable.

where, for a subset P of Q, $\text{pre}^*(P)$ denotes the set of states from which set P is reachable.

It is well known that reachability of a set of locations is decidable using the classical zone (or region) symbolic representation (see [18]). As Q_F corresponds to all states with location in F, the partition can then be symbolically computed on the zone graph.

2.4 Classification of Timed Properties

A timed property is defined by a timed language $\varphi \subseteq \text{tw}(\Sigma)$ that can be recognized by a timed automaton. That is, the set of regular timed properties are considered. Given a timed word $\sigma \in \text{tw}(\Sigma)$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$.

Definition 1 (Regular, safety, and co-safety properties).

- *Regular timed properties are the properties that can be defined by languages accepted by a TA.*
- *Safety timed properties are the non-empty prefix-closed regular timed properties.*
- *Co-safety timed properties are the non-universal*[3] *extension-closed regular timed properties.*

[3] The universal property over $\mathbb{R}_{\geq 0} \times \Sigma$ is $\text{tw}(\Sigma)$.

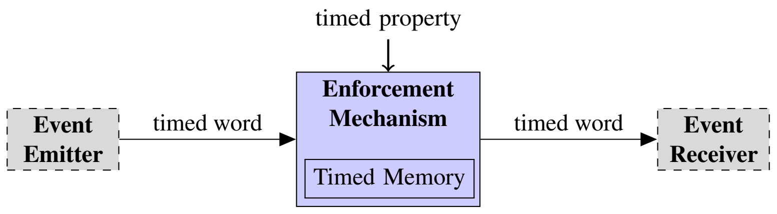

Fig. 9: RE of a timed property. The enforcement mechanism (EM) is synthesized from a timed property. At runtime, the EM is placed between an event emitter (EE) and event receiver (ER); it receives as input a timed word from the EE and produces as output a timed word for the ER.

As in the untimed case, safety (resp. co-safety) properties state that “nothing bad should ever happen” (resp. “something good should happen within a finite amount of time”).

3 Overview of RE Approaches for Timed Properties

In this section, we overview some formal approaches [44, 80, 81, 83, 84, 91–93] to the runtime enforcement of timed properties described by timed automata (TA). Properties can feature uncontrollable events which can be only seen by the enforcement mechanism (EM) and cannot be acted upon. The runtime enforcement problem is conceptualized as illustrated in Fig. 9: an EM reads as input a timed word and should transform and output it so that it complies with a timed property used to obtain the EM, using a timed memory which accounts for the physical time during which elements have been stored. In all the following frameworks, EMs are described with two paradigms: a denotational one where EMs are seen as functions through their input/output behavior, and two operational ones: input/output labeled transition systems and algorithms. These approaches differ either in the supported classes of properties for which EMs can be synthesized and the enforcement operations of the enforcement mechanism.

Runtime enforcement of timed properties [84] (for safety and co-safety properties). In [84] the first steps to runtime enforcement of (continuous) timed safety and co-safety properties was introduced. EMs were endowed only with an enforcement operation allowing to delaying events to satisfy the required property. For this purpose, the EM stores some actions for a certain time period computed when they are received. Requirements over the EMs ensured that their outputs not only satisfy the required property, but also with the shortest delay according to the current satisfaction of the property.

*Runtime enforcement of regular timed properties [81, 83].*The approach in [81, 83] generalizes [84] and synthesizes EMs for any regular timed property. It allows considering interesting properties of systems belonging to a larger class specifying some form of transactional behavior. The difficulty that arises is that the EMs should consider the alternation between currently satisfying and not satisfying the property[4]. The unique enforcement operation is still delaying events as in [84].

[4] Indeed, in safety (resp. co-safety) (timed) automaton, there are only good, currently good, and bad states (resp. bad, currently bad, and good states), and thus the strategies for the EM is simpler: avoiding the bad states (resp. reaching a good state) [42].

Runtime enforcement of regular timed properties by suppressing and delaying events [44]. The approach in [44] considers events composed of actions with absolute occurrence dates, and allows increasing the dates (while allowing reducing delays between events in memory). Moreover, suppressing events is also introduced. An event is suppressed if it is not possible to satisfy the property by delaying, whatever are the future continuations of the input sequence (i.e., the underlying TA can only reach non-accepting states from which no accepting state can be reached). In Sec. 4, we overview this framework.

*Runtime enforcement of parametric timed properties with practical applications [80].*The framework in [80] makes one step towards practical runtime enforcement by considering event-based specifications where i) time between events matters and ii) events carry data values ([54]) from the monitored system. It defines how to enforce parametric timed specifications which are useful to model requirements from some application domains such as network security which have constraints both on time and data. For this, it introduces the model of Parametrized Timed Automata with Variables (PTAVs). PTAVs extend TAs with session parameters, internal and external variables. The framework presents how to synthesize EMs as in [44, 83] from PTAVs and shows the usefulness of enforcing such expressive specifications on application scenarios.

*Enforcement of timed properties with uncontrollable events [91, 92].*The approach in [91, 92] presents a framework for enforcing regular untimed and timed properties with uncontrollable events. An EM cannot delay nor intercept an uncontrollable event. To cope with uncontrollable events, the notion of transparency should be weakened to the so-called notion of compliance. Informally, compliance means that the order of controllable events should be maintained by the EM, while uncontrollable events should be released as output soon after they are received.

*Runtime enforcement of cyber-physical systems [87].*In synchronous reactive systems, terminating the system or delaying the reaction is not feasible. Thus, the approaches in [44,80,81,83,91,92] are not suitable for such systems. The approach in [87] introduces a framework for synchronous reactive systems with bidirectional synchronous EMs. While the framework considers similar notions of soundness, and transparency, it also introduces the so-called additional requirements of causality and instantaneity which are specific to synchronous executions. Moreover, the framework considers properties expressed using a variant of Discrete Timed Automata (DTA).

4 A framework for the Runtime Enforcement of Timed Properties

In this section, we present a framework for the runtime enforcement of timed properties described by timed automata [44]. Most of the material comes from [44, 79].

4.1 Overview

Given some timed property φ and an input timed word σ, the EM outputs a timed word o that satisfies φ. The considered EMs are *time retardants*, i.e., their main enforcement operation consists in delaying the received events[5]. In addition to introducing additional delays (increasing dates), for the EM and system to continue executing, the

[5] Several application domains have requirements, where the required timing constraints can be satisfied by increasing dates of some actions [67].

EM can suppress events when no delaying is appropriate. However, it can not change the order of events. The EM may also reduce delays between events stored in its memory.

To ease the design and implementation of EMs in a timed context, they are described at three levels of abstraction: *enforcement functions*, *enforcement monitors*, and *enforcement algorithms*; all of which can be deployed to operate online. EMs should abide to some requirements, namely the physical constraint, soundness, transparency.

- The *physical constraint* says that the output produced for an extension σ' of an input word σ extends the output produced for σ. This stems from the fact that, over time the enforcement function outputs a continuously growing sequence of events. The output for a given input can only be modified by appending new events (with greater dates).
- *Soundness* says that the output either satisfies property φ, or is empty. This allows to output nothing if there is no way to satisfy φ. Note that, together with the physical constraint, this implies that no event can be appended to the output before being sure that the property will be eventually satisfied with subsequent output events.
- *Transparency* says that the output is a delayed subsequence of the input σ; that is with increased dates, preserved order, and possibly suppressed events.

Notice that for any input σ, releasing ε as output would satisfy soundness, transparency, and the physical constraint. We want to suppress an event or to introduce additional delay only when necessary. Additionally, EMs should also respect optimality requirements:

- *Streaming behavior and deciding to output as soon as possible.* Since an EM does not know the entire input sequence, for efficiency reasons, the output should be built incrementally in a streaming fashion. EMs should take decision to release input events as soon as possible. The EM should wait to receive more events, only when there is no possibility to correct the input.
- *Optimal suppression.* Suppressing events should occur only when necessary, i.e., when, upon the reception of a new event, there is no possibility to satisfy the property, whatever is the continuation of the input.
- *Optimal dates.* Choosing/increasing dates should be done in way that dates are optimal with respect to the current situation, releasing here as output as soon as possible.

The enforcement function $E_\varphi : \mathrm{tw}(\Sigma) \to \mathrm{tw}(\Sigma)$ for a property φ defines how an input stream σ is transformed into an output stream. An enforcement monitor (see [44]) is a more concrete view and defines the operational behavior of the EM over time as a timed labelled transition system. An enforcement algorithm realises enforcement monitor in pseudo code with two concurrent processes and a shared buffer At an abstract level, one process stores the received events in the shared buffer and computes their releasing date. The other process scrutinizes the shared buffer and releases the event at their releasing dates . In [44], we formally prove that enforcement functions respect the requirements, that enforcement monitors realizes enforcement functions, that enforcement algorithms implements enforcement monitors, and that all description of EMs can be optimized for the particular case of timed safety properties.

4.2 Intuition on an Example

We provide some intuition on the expected behavior of EMs. Consider two processes that access to and operate on a shared resource. Each process i (with $i \in \{1,2\}$) has three interactions with the resource: acquisition (acq_i), release (rel_i), and a specific operation

(op_i). Both processes can also execute a common action op. System initialization is denoted by action $init$. In the following, variable t keeps track of the evolution of time.

Consider one specification, referred to as S_1, of the shared ressource: "*Operations op_1 and op_2 should execute in a transactional manner. Both actions should be executed, in any order, and any transaction should contain one occurrence of op_1 and op_2. Each transaction should complete within 10 tu. Between operations op_1 and op_2, occurrences of operation op can occur. There is at least 2 tu between any two occurrences of any operation.*"

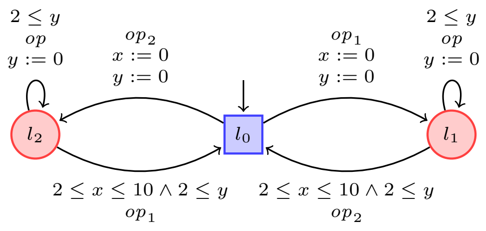

Fig. 10: TA defining property S_1.

Figure 11 illustrates the behavior of an EM and how it transforms an input timed word (red) to a correct output timed word (blue) satisfying S_1; actions are in abscissa and occurrence dates in ordinate. Note, the satisfaction of the property is not represented in the figure. The input sequence is $\sigma = (2, op_1) \cdot (3, op_1) \cdot (3.5, op) \cdot (6, op_2)$. At $t = 2$, the EM can not output action op_1 because this action alone does not satisfy the specification (and the EM does not yet know the next events i.e., actions and dates). If the next action was op_2, then, at the date of its reception, the EM could output action op_1 followed by op_2, as it could choose dates for both actions in order to satisfy the timing constraints. At $t = 3$ the EM receives a second op_1 action. Clearly, there is no possible date for these two op_1 actions to satisfy the specification, and no continuation could solve the situation. The EM thus suppresses the second op_1 action, since this action is the one that prevents satisfiability in the future. At $t = 3.5$, when the EM receives action op, the input sequence still does not satisfy the specification, but there exists an appropriate delaying of such action so that with future events, the specification can be satisfied. At $t = 6$, the EM receives action op_2, it can decide that action op_1 followed by op and op_2 can be released as output with appropriate delaying. Thus, the date associated with the first op_1 action is set to 6 (the earliest possible date, since this decision is taken at $t = 6$), 8 for action op (since 2 is the minimal delay between those actions satisfying the timing constraint), and 10 for action op_2. Henceforth, as shown in the figure, the output of the EM for σ is $(6, op_1) \cdot (8, op) \cdot (10, op_2)$.

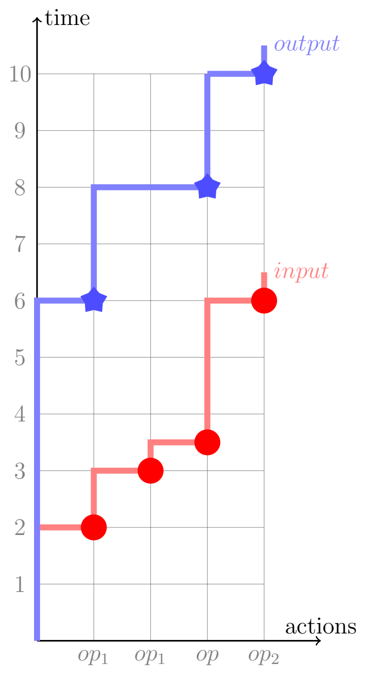

Fig. 11: Illustration of the behavior of an EM enforcing S_1.

5 Tool Implementations

Any tool for runtime verification [13] can perform basic enforcement by terminating the execution of programs violating their specification. There are, however, several runtime verification tools that go further and feature their own enforcement operations, for instance Java-MOP and EnforceMOP [70] with handlers, LARVA [31] with compensations [32]. There are also numerous tools in the security domain enforcing (implicit) specifications related to the security of the monitored applications (memory safety, control-flow integrity, etc); see [104] for a recent overview.

To the best of our knowledge, there are two tools dedicated to the runtime enforcement of timed properties: TiPEX [82] and GREP [93]. TiPEX implements the framework presented in Sec. 4 and provides additional features to synthesize timed automata and check the class of a timed automaton as per Def. 1. A detailed description of the TiPEX tool with some examples is provided in [82]. TiPEX can be downloaded from [97]. GREP [93] follows the same objectives as TiPEX but is based on game theory to synthesize the enforcement mechanisms. GREP also handles uncontrollable events.

6 Open Challenges and Avenues for Future Work

We conclude this tutorial by proposing some future research directions.

Enforcement monitoring for systems with limited memory. An enforcement mechanism basically acts as a filter storing the input events in its memory, until it is certain that the underlying property will be satisfied. As was the case with untimed properties [16, 17, 50, 106], defining enforcement mechanisms and new enforcement strategies should be defined when there are bounds on memory usage or limited resources. Delineating the subset of enforceable timed properties for which effective enforcement mechanisms can be obtained is also a subject for future work.

Predictive runtime enforcement. Predictive runtime enforcement considers the case where knowledge about the event emitter is available [86]. When the enforcement mechanism knows the set of input sequences that it may receive, then it may anticipate decisions of releasing events as output without storing them in memory nor waiting for future events, e.g., when it knows that all the possible continuations of the input it has observed will not violate the property. Predictive runtime enforcement of timed properties poses several difficulties and challenges that have to be further explored [85, 86].

Realizing requirements automatically. In current research efforts, enforcement mechanisms are seen as modules outside the system, which take as input a stream of events (output of the system being monitored) and verify or correct this stream according to the property. For a better adoption of runtime enforcement theories and tools, one direction is to define methods and instrumentation techniques so that enforcement mechanisms can realize the requirements (which could not be integrated in the initial application. For this, one can imagine enforcement monitors (realizing some requirements) integrated as another layer on top of the core functionality or with libraries, inspiring from aspect-oriented programming [60, 61] and acceptability-oriented computing [95].

Decentralized runtime enforcement. Decentralized runtime verification approaches [15, 29, 37, 40] allow decentralizing the monitoring mechanism on the components of a system (see [51] for an overview). Such approaches deal with the situations where it is not desired to impose a central observation point in the system. Frameworks for decentralized runtime enforcement (of timed properties) have yet to be defined and will permit enforcing properties on distributed systems. For this purpose, one can inspire from from generalized consensus to help enforcement mechanisms forge a collective decisions when applying enforcement operations to the system.

Acknowledgment The authors thank Frédéric Desprez, Antoine El-Hokayem, Raphaël Jakse, Ali Kassem, and the reviewers for their comments on a preliminary version of this tutorial. The framework for runtime enforcement of timed properties reported in Sec. 2 to Sec. 5 is based upon joint research efforts with colleagues and friends: Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, Laurent Mounier, Omer Nguena-Timo, Matthieu Renard, and Antoine Rollet.

References

1. Proceedings of the 5th Annual Symp. on Logic in Computer Science (LICS ’90). IEEE Computer Society (1990)
2. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13(1), 4:1–4:40 (2009)
3. Aktug, I., Dam, M., Gurov, D.: Provably correct runtime monitoring. J. Log. Algebr. Program. 78(5), 304–339 (2009)
4. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. 21(4), 181–185 (1985)
5. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for real-time systems. In: Proc. of the 5th Annual Symp. on Logic in Computer Science (LICS ’90) [1], pp. 414–425
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
7. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. In: Proc. of the Fifth Annual Symp. on Logic in Computer Science (LICS ’90) [1], pp. 390–401
8. Amiar, A., Delahaye, M., Falcone, Y., du Bousquet, L.: Compressing microcontroller execution traces to assist system analysis. In: Schirner, G., Götz, M., Rettberg, A., Zanella, M.C., Rammig, F.J. (eds.) Embedded Systems: Design, Analysis and Verification - 4th IFIP TC 10 Int. Embedded Systems Symp., IESS 2013. IFIP Advances in Information and Communication Technology, vol. 403, pp. 139–150. Springer (2013)
9. Amiar, A., Delahaye, M., Falcone, Y., du Bousquet, L.: Fault localization in embedded software based on a single cyclic trace. In: IEEE 24th Int. Symp. on Software Reliability Engineering, ISSRE 2013. pp. 148–157. IEEE Computer Society (2013)
10. de Amorim, A.A., Hritcu, C., Pierce, B.C.: The meaning of memory safety. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust - 7th Int. Conf., POST 2018, Held as Part of the European Joint Conf.s on Theory and Practice of Software, ETAPS 2018, Proc. LNCS, vol. 10804, pp. 79–105. Springer (2018)
11. Babaee, R., Gurfinkel, A., Fischmeister, S.: Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In: Colombo and Leucker [30], pp. 187–204
12. Bartocci, E., Falcone, Y. (eds.): Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS, vol. 10457. Springer (2018)
13. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First int. competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. STTT 21(1), 31–70 (2019)

14. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci and Falcone [12], pp. 1–33
15. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Form. Meth. in Syst. Design 48(1-2), 46–93 (2016)
16. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite edit automata. Electr. Notes Theor. Comput. Sci. 229(3), 19–35 (2009)
17. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite and pushdown edit automata. Int. J. Inf. Sec. 12(4), 319–336 (2013)
18. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Proc. of the 4th Advanced Course on Petri Nets - Lecture Notes on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer (2003)
19. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? In: Degano, P., Guttman, J.D., Martinelli, F. (eds.) Formal Aspects in Security and Trust, 5th Int. Workshop, FAST 2008. LNCS, vol. 5491, pp. 287–301. Springer (2008)
20. Bielova, N., Massacci, F.: Predictability of enforcement. In: Erlingsson, Ú., Wieringa, R.J., Zannone, N. (eds.) Engineering Secure Software and Systems - Third Int. Symp., ESSoS 2011s. LNCS, vol. 6542, pp. 73–86. Springer (2011)
21. Bielova, N., Massacci, F.: Iterative enforcement by suppression: Towards practical enforcement theories. J. of Computer Security 20(1), 51–79 (2012)
22. Birgisson, A., Dhawan, M., Erlingsson, Ú., Ganapathy, V., Iftode, L.: Enforcing authorization policies using transactional memory introspection. In: Ning, P., Syverson, P.F., Jha, S. (eds.) Proc. of the 2008 ACM Conf. on Computer and Communications Security, CCS 2008. pp. 223–234. ACM (2008)
23. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) Formal Methods and Software Engineering - 14th Int. Conference on Formal Engineering Methods, ICFEM 2012. Lecture Notes in Computer Science, vol. 7635, pp. 494–509. Springer (2012)
24. Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: Proc. of the CGO 2011, The 9th Int. Symp. on Code Generation and Optimization. pp. 213–223. IEEE Computer Society (2011)
25. Bruening, D., Zhao, Q.: Using Dr. Fuzz, Dr. Memory, and custom dynamic tools for secure development. In: IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016. p. 158. IEEE Computer Society (2016)
26. Chabot, H., Khoury, R., Tawbi, N.: Extending the enforcement power of truncation monitors using static analysis. Computers & Security 30(4), 194–207 (2011)
27. Chang, E.Y., Manna, Z., Pnueli, A.: Characterization of temporal property classes. In: Kuich, W. (ed.) Automata, Languages and Programming, 19th Int. Colloquium, ICALP92. LNCS, vol. 623, pp. 474–486. Springer (1992)
28. Chong, S., Vikram, K., Myers, A.C.: SIF: enforcing confidentiality and integrity in web applications. In: Provos, N. (ed.) Proc. of the 16th USENIX Security Symp. USENIX Association (2007)
29. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Form. Meth. in Syst. Design 49(1-2), 109–158 (2016)
30. Colombo, C., Leucker, M. (eds.): Runtime Verification - 18th Int. Conf., RV 2018, LNCS, vol. 11237. Springer (2018)
31. Colombo, C., Pace, G.: Runtime verification using LARVA. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. An Int. Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Kalpa Publications in Computing, vol. 3, pp. 55–63. EasyChair (2017)
32. Colombo, C., Pace, G.J.: Recovery within long-running transactions. ACM Comput. Surv. 45(3), 28:1–28:35 (2013)

33. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Provably correct inline monitoring for multithreaded java-like programs. Journal of Computer Security 18(1), 37–59 (2010)
34. Davi, L., Sadeghi, A., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Cheung, B.S.N., Hui, L.C.K., Sandhu, R.S., Wong, D.S. (eds.) Proc. of the 6th ACM Symp. on Information, Computer and Communications Security, ASIACCS 2011. pp. 40–51. ACM (2011)
35. Duck, G.J., Yap, R.H.C., Cavallaro, L.: Stack bounds protection with low fat pointers. In: 24th Annual Network and Distributed System Security Symp., NDSS 2017. The Internet Society (2017)
36. El-Harake, K., Falcone, Y., Jerad, W., Langet, M., Mamlouk, M.: Blocking advertisements on Android devices using monitoring techniques. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation - 6th Int. Symp., ISoLA 2014, Part II. LNCS, vol. 8803, pp. 239–253. Springer (2014)
37. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan, T., Sen, K. (eds.) Proc. of the 26th ACM SIGSOFT Int. Symp. on Software Testing and Analysis. pp. 372–375. ACM (2017)
38. Erlingsson, Ú., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Kienzle, D.M., Zurko, M.E., Greenwald, S.J., Serbau, C. (eds.) Proc. of the 1999 Workshop on New Security Paradigms. pp. 87–95. ACM (1999)
39. Falcone, Y.: You should better enforce than verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First Int. Conf., RV 2010. LNCS, vol. 6418, pp. 89–105. Springer (2010)
40. Falcone, Y., Cornebize, T., Fernandez, J.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 Int. Conf., FORTE 2014, Held as Part of the 9th Int. Federated Conf. on Distributed Computing Techniques, DisCoTec 2014. LNCS, vol. 8461, pp. 66–83. Springer (2014)
41. Falcone, Y., Currea, S., Jaber, M.: Runtime verification and enforcement for Android applications with RV-Droid. In: Qadeer and Tasiran [89], pp. 88–95
42. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? STTT 14(3), 349–382 (2012)
43. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
44. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Sci. Comput. Program. 123, 2–41 (2016)
45. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In: Colombo and Leucker [30], pp. 241–262
46. Falcone, Y., Marchand, H.: Enforcement and validation (at runtime) of various notions of opacity. Discrete Event Dynamic Systems 25(4), 531–570 (2015)
47. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci and Falcone [12], pp. 103–134
48. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Form. Meth. in Syst. Design 38(3), 223–262 (2011)
49. Ferraiuolo, A., Zhao, M., Myers, A.C., Suh, G.E.: Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security, CCS 2018. pp. 1583–1600. ACM (2018)

50. Fong, P.W.L.: Access control by tracking shallow execution history. In: 2004 IEEE Symp. on Security and Privacy (S&P 2004). pp. 43–55. IEEE Computer Society (2004)
51. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci and Falcone [12], pp. 176–210
52. Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: 2014 IEEE Symp. on Security and Privacy, SP 2014. pp. 575–589. IEEE Computer Society (2014)
53. Hallé, S., Khoury, R., Betti, Q., El-Hokayem, A., Falcone, Y.: Decentralized enforcement of document lifecycle constraints. Inf. Syst. 74(Part), 117–135 (2018)
54. Havelund, K., Reger, G., Thoma, D., Zalinescu, E.: Monitoring events that carry data. In: Bartocci and Falcone [12], pp. 61–102
55. Ji, Y., Wu, Y., Lafortune, S.: Enforcement of opacity by public and private insertion functions. Automatica 93, 369–378 (2018)
56. Johansen, H.D., Birrell, E., van Renesse, R., Schneider, F.B., Stenhaug, M., Johansen, D.: Enforcing privacy policies with meta-code. In: Kono, K., Shinagawa, T. (eds.) Proc. of the 6th Asia-Pacific Workshop on Systems, APSys 2015. pp. 16:1–16:7. ACM (2015), https://doi.org/10.1145/2797022
57. Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N.B., Ponomarev, D.: Branch regulation: Low-overhead protection from code reuse attacks. In: 39th Int. Symp. on Computer Architecture (ISCA 2012). pp. 94–105. IEEE Computer Society (2012)
58. Khoury, R., Tawbi, N.: Corrective enforcement: A new paradigm of security policy enforcement by monitors. ACM Trans. Inf. Syst. Secur. 15(2), 10:1–10:27 (2012)
59. Khoury, R., Tawbi, N.: Which security policies are enforceable by runtime monitors? A survey. Computer Science Review 6(1), 27–45 (2012)
60. Kiczales, G.: Aspect-oriented programming. In: Roman et al. [96], p. 730
61. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Roman et al. [96], pp. 49–58
62. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Boneh, D. (ed.) Proc. of the 11th USENIX Security Symp. pp. 191–206. USENIX (2002)
63. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L.R., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. Form. Meth. in Syst. Design 51(2), 332–361 (2017)
64. Kozyri, E., Arden, O., Myers, A.C., Schneider, F.B.: JRIF: reactive information flow control for java. In: Guttman, J.D., Landwehr, C.E., Meseguer, J., Pavlovic, D. (eds.) Foundations of Security, Protocols, and Equational Reasoning - Essays Dedicated to Catherine A. Meadows. LNCS, vol. 11565, pp. 70–88. Springer (2019)
65. Kumar, A., Ligatti, J., Tu, Y.: Query monitoring and analysis for database privacy - A security automata model approach. In: Wang, J., Cellary, W., Wang, D., Wang, H., Chen, S., Li, T., Zhang, Y. (eds.) Web Information Systems Engineering - WISE 2015 - 16th Int. Conf., Part II. LNCS, vol. 9419, pp. 458–472. Springer (2015)
66. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. 3(2), 125–143 (1977)
67. Lesage, J., Faure, J., Cury, J.E.R., Lennartson, B. (eds.): 12th Int. Workshop on Discrete Event Systems, WODES 2014. Int. Federation of Automatic Control (2014)
68. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12(3), 19:1–19:41 (2009)
69. Lourenço, J.M., Fiedor, J., Krena, B., Vojnar, T.: Discovering concurrency errors. In: Bartocci and Falcone [12], pp. 34–60
70. Luo, Q., Rosu, G.: Enforcemop: a runtime property enforcement system for multithreaded programs. In: Pezzè, M., Harman, M. (eds.) Int. Symp. on Software Testing and Analysis, ISSTA. pp. 156–166. ACM (2013)

71. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems - specification. Springer (1992)
72. Margaria, T., Steffen, B. (eds.): Leveraging Applications of Formal Methods, Verification and Validation - 7th Int. Symp., ISoLA 2016, Part II, LNCS, vol. 9953 (2016)
73. Martinelli, F., Matteucci, I., Mori, P., Saracino, A.: Enforcement of U-XACML history-based usage control policy. In: Barthe, G., Markatos, E.P., Samarati, P. (eds.) Security and Trust Management - 12th Int. Workshop, STM 2016. LNCS, vol. 9871, pp. 64–81. Springer (2016)
74. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14(3), 249–289 (2012)
75. Nguyen, T., Bartocci, E., Nickovic, D., Grosu, R., Jaksic, S., Selyunin, K.: The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In: Margaria and Steffen [72], pp. 371–379
76. Pavlich-Mariscal, J.A., Demurjian, S.A., Michel, L.D.: A framework of composable access control definition, enforcement and assurance. In: Bastarrica, M.C., Solar, M. (eds.) XXVII Int. Conf. of the Chilean Computer Science Society (SCCC 2008). pp. 13–22. IEEE Computer Society (2008)
77. Pavlich-Mariscal, J.A., Demurjian, S.A., Michel, L.D.: A framework for security assurance of access control enforcement code. Computers & Security 29(7), 770–784 (2010)
78. Pavlich-Mariscal, J.A., Michel, L., Demurjian, S.A.: A formal enforcement framework for role-based access control using aspect-oriented programming. In: Briand, L.C., Williams, C. (eds.) Model Driven Engineering Languages and Systems, 8th Int. Conf., MoDELS 2005. LNCS, vol. 3713, pp. 537–552. Springer (2005)
79. Pinisetty, S.: Runtime enforcement of timed properties. (Enforcement à l'éxécution de propriétés temporisées). Ph.D. thesis, University of Rennes 1, France (2015)
80. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Runtime enforcement of parametric timed properties with practical applications. In: Lesage et al. [67], pp. 420–427
81. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Runtime enforcement of regular timed properties. In: Cho, Y., Shin, S.Y., Kim, S., Hung, C., Hong, J. (eds.) Symp. on Applied Computing, SAC 2014. pp. 1279–1286. ACM (2014)
82. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Tipex: A tool chain for timed property enforcement during execution. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th Int. Conf., RV 2015. LNCS, vol. 9333, pp. 306–320. Springer (2015)
83. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.: Runtime enforcement of timed properties revisited. Form. Meth. in Syst. Design 45(3), 381–422 (2014)
84. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.L.: Runtime enforcement of timed properties. In: Qadeer and Tasiran [89], pp. 229–244
85. Pinisetty, S., Jéron, T., Tripakis, S., Falcone, Y., Marchand, H., Preoteasa, V.: Predictive runtime verification of timed properties. J. of Systems and Software 132, 353–365 (2017)
86. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Form. Meth. in Syst. Design 51(1), 154–199 (2017)
87. Pinisetty, S., Roop, P.S., Smyth, S., Allen, N., Tripakis, S., Hanxleden, R.V.: Runtime enforcement of cyber-physical systems. ACM Trans. Embed. Comput. Syst. 16(5s), 178:1–178:25 (Sep 2017)
88. Pnueli, A.: Embedded systems: Challenges in specification and verification. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) Embedded Software, Second International Conf. Lecture Notes in Computer Science, vol. 2491, pp. 1–14. Springer (2002)
89. Qadeer, S., Tasiran, S. (eds.): Runtime Verification, Third Int. Conf., RV 2012, LNCS, vol. 7687. Springer (2013)
90. Reger, G., Havelund, K.: What is a trace? A runtime verification perspective. In: Margaria and Steffen [72], pp. 339–355

91. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal enforcement of (timed) properties with uncontrollable events. Mathematical Structures in Computer Science 29(1), 169–214 (2019)
92. Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H.: Enforcement of (timed) properties with uncontrollable events. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th Int. Colloquium. LNCS, vol. 9399, pp. 542–560. Springer (2015)
93. Renard, M., Rollet, A., Falcone, Y.: Runtime enforcement using büchi games. In: Erdogmus, H., Havelund, K. (eds.) Proc. of the 24th ACM SIGSOFT Int. SPIN Symp. on Model Checking of Software. pp. 70–79. ACM (2017)
94. Riganelli, O., Micucci, D., Mariani, L., Falcone, Y.: Verifying policy enforcers. In: Lahiri, S.K., Reger, G. (eds.) Runtime Verification - 17th Int. Conference, RV 2017. Lecture Notes in Computer Science, vol. 10548, pp. 241–258. Springer (2017)
95. Rinard, M.C.: Acceptability-oriented computing. In: Crocker, R., Jr., G.L.S. (eds.) Companion of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003. pp. 221–239. ACM (2003)
96. Roman, G., Griswold, W.G., Nuseibeh, B. (eds.): 27th Int. Conf. on Software Engineering (ICSE 2005). ACM (2005)
97. S. Pinisetty et al.: TiPEX website. https://srinivaspinisetty.github.io/Timed-Enforcement-Tools/ (2015)
98. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (2000)
99. Selyunin, K., Nguyen, T., Bartocci, E., Nickovic, D., Grosu, R.: Monitoring of MTL specifications with ibm's spiking-neuron model. In: Fanucci, L., Teich, J. (eds.) 2016 Design, Automation & Test in Europe Conf. & Exhibition, DATE 2016. pp. 924–929. IEEE (2016)
100. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: Proc. of the 2005 USENIX Annual Technical Conf. pp. 17–30. USENIX (2005)
101. Sifakis, J.: Modeling real-time systems. In: Proc. of the 25th IEEE Real-Time Systems Symp. (RTSS 2004). pp. 5–6. IEEE Computer Society (2004)
102. Sifakis, J., Tripakis, S., Yovine, S.: Building models of real-time systems from application software. Proc. of the IEEE 91(1), 100–111 (2003)
103. Sistla, A.P.: Safety, liveness and fairness in temporal logic. Formal Asp. Comput. 6(5), 495–512 (1994)
104. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: Sanitizing for security. CoRR abs/1806.04355 (2018)
105. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symp. on Security and Privacy, SP 2013. pp. 48–62. IEEE Computer Society (2013)
106. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Inf. Comput. 206(2-4), 158–184 (2008)
107. Wu, M., Zeng, H., Wang, C.: Synthesizing runtime enforcer of safety properties under burst error. In: Rayadurgam, S., Tkachuk, O. (eds.) NASA Formal Methods - 8th Int. Symp. LNCS, vol. 9690, pp. 65–81. Springer (2016)
108. Wu, M., Zeng, H., Wang, C., Yu, H.: Safety guard: Runtime enforcement for safety-critical cyber-physical systems: Invited. In: Proc. of the 54th Annual Design Automation Conf. pp. 84:1–84:6. ACM (2017)
109. Yin, X., Lafortune, S.: A new approach for synthesizing opacity-enforcing supervisors for partially-observed discrete-event systems. In: American Control Conf., ACC 2015. pp. 377–383. IEEE (2015)
110. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th Int. Symp., NFM 2012. LNCS, vol. 7226, pp. 418–432. Springer (2012)