Runtime Failure Prevention and Reaction

Yliès Falcone[1], Leonardo Mariani[2], Antoine Rollet[3], and Saikat Saha[2]

[1] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr
[2] Univ. of Milano Bicocca, 20126 Milano, Italy, {mariani,saha}@disco.unimib.it
[3] LaBRI, Bordeaux INP, University of Bordeaux, Bordeaux, France,
antoine.rollet@labri.fr

Abstract. This chapter describes how to use in-the-field runtime techniques to improve the dependability of software systems. In particular, we first present an overall vision of the problem of ensuring highly-dependable behaviours at runtime based on the concept of autonomic monitor, and then we present the two families of relevant approaches for this purpose. First, we present techniques related to *runtime enforcement* that can prevent the system producing bad behaviours. Second, we describe *healing* techniques that can detect if the system has produced a bad behaviour and react to the situation accordingly (e.g., moving the system back to a correct state).

Keywords: runtime enforcement, prevention of failures, reaction to failures, self-healing, autonomic computing

1 Introduction

Fully assessing the quality of software systems in-house is infeasible for several well-known reasons. For instance, the space of the behaviours and the configurations that must be validated in-house and their combination might be intractable; many real usage scenarios might be impossible to reproduce and validate in-house; and the context of execution of a system might be only partially known, such as for the many software applications that can be extended directly by their end-users through the installation of plug-ins, which makes the problem of verifying software in-house extremely hard.

To improve the dependability of software systems, the software running in the field can be equipped with solutions to prevent, detect, and react to failures. These solutions attempt to handle the faults that have not been revealed in-house directly in the field, once they produce observable effects. The range of solutions to cope with failures at runtime is quite broad. It spans from fault tolerance techniques, which exploit various forms of redundancy to overcome the impact of failures, to self-healing approaches, which can automatically heal executions before they produce an observable effect.

In this chapter, we discuss approaches concerning two complementary, although related, aspects: *runtime enforcement* techniques, which can prevent a monitored program from misbehaviouring by enforcing the program to run according to its specification, and *healing* techniques, which can react to misbehaviours and failures to restore the

normal execution of the monitored program, possibly completely masking any observed failure.

Runtime enforcement and healing techniques look at the same problem from the opposite sides. The former affects executions with the objective of preventing failures, for instance preventing that a wrong result is ultimately generated by a program. The latter affects executions with the objective of restoring normal executions once a failure has been observed, possibly masking the failure to any external observer (e.g., the users of a system).

Runtime enforcement typically requires a specification of a system, for instance the specification of a property that must be satisfied by an application, to properly steer executions. When such a specification is available, it can be extremely effective in preventing failures. However, its effectiveness is limited by the scope and availability of the specifications. On the other hand, healing techniques often exploit source of information alternative to ad-hoc specifications (e.g., program versions, redundancy, and failure patterns) to be able to remedy to the observed problems. The two classes of solutions together represent a relevant range of options to deal with failures at runtime.

Of course, the boundaries between enforcement and healing are not always sharp, and some approaches in one category may have some characteristics present also in the approaches in the other category, and vice versa. In this chapter, we do not aim to exhaustively discuss the approaches in the two areas or claim that the distinction between enforcement and healing is effective in all the cases, but rather we aim to give a general and coherent vision of these techniques and to provide an initial set of references for the readers interested in more details. The discussion is mostly informal, and specific details are provided only when needed.

The chapter is organised as follows. Section 2 presents the concept of autonomic monitoring, which is exploited to discuss as part of the same conceptual framework both runtime enforcement and healing techniques. Section 3 discusses techniques to prevent failures by enforcing the correct behaviours. Section 4 presents techniques to react to failures by restoring the correct behaviour. Section 5 discusses some open challenges, and finally Section 6 provides final remarks.

2 Autonomic Monitors

Runtime enforcement and healing techniques have to deal with faults, anomalous behaviours and failures. In this chapter, a *failure* is the inability of a system or component to perform its required functions within previously specified limits, a *fault* is an incorrect step, process or data definition, and an *anomalous behaviour* (or a *bad behaviour*) is anything observed in the operation of software that deviates from expectations based on previously verified software products, reference documents, or other sources of indicative behaviour [53]. A fault present in a software may cause anomalous behaviours and even worse failures.

We present solutions for runtime enforcement and healing referring to the same high-level software architecture. Since both runtime enforcement and healing represent specific cases of autonomic computing technologies, we adapt the architecture of a general autonomic manager proposed by IBM [56] to the case of an *Autonomic Mon-*

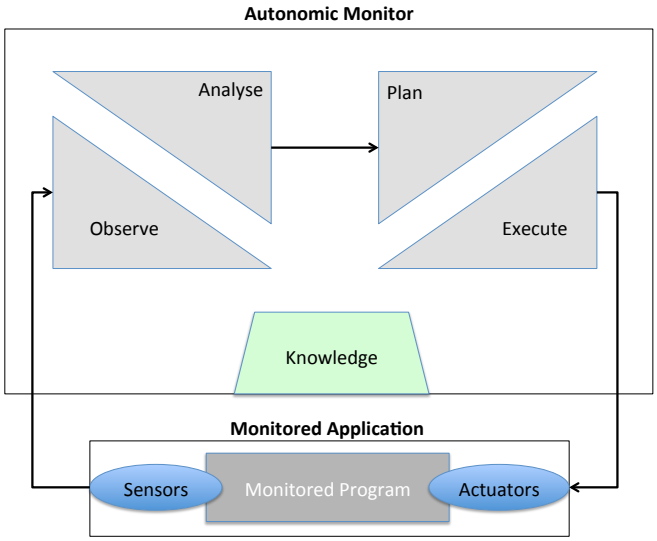


Fig. 1: General architecture for enforcement and healing at runtime.

itor that can perform enforcement and healing at runtime. The resulting architecture is shown in Figure 1.

The two main components of the architecture are the *Monitored Program* and the *Autonomic Monitor*. The Monitored Program is coupled with the Autonomic Monitor, which adds behaviour enforcement and healing capabilities to the monitored program. The interaction between these two components is possible through *Sensors*, which are probes or gauges that collect information about the monitored program, and *Effectors*, which are handles that can be used to change the behaviour of the monitored program according to the decisions taken by the Autonomic Monitor. We consider here the *Monitored Program* in a general sense, meaning that for instance its configuration or its environment is included in this definition.

The behaviour of the Autonomic Monitor is determined by a feedback loop that comprises four phases: *Observe*, *Analyse*, *Plan*, and *Execute*. These four phases exploit some *Knowledge* about the monitored program to work effectively. In particular, the *Observe* phase collects information and data from the monitored program using sensors, and filters the collected data until events that need to be analysed are generated and passed to the analysis phase. The Observe phase can also update the Knowledge based on the collected information. The Analyse phase performs data analysis depending on the knowledge and the events that have been produced. Should an action need to be taken, the control is passed to the Plan phase, which identifies the appropriate procedures to enforce a given behaviour or to heal the monitored program. The Execute phase actuates the changes to the behaviour of the monitored program based on the decision taken by the Plan. The Execute phase also performs the preparation tasks,

such as locking resources, that might be necessary before the monitored program can be affected. When the monitored program is modified, the Knowledge can be updated accordingly.

In this chapter, we describe the enforcement and healing techniques based on:

- the *requirements* on the monitored program, and on the sensors and effectors that must be introduced into the monitored program;
- the *behaviour of the four phases* Observe, Analyse, Plan and Execute that characterise an autonomic monitor, note that some phases might be extremely simple for some techniques;
- the *knowledge* about the system that must be provided and updated to let the autonomic monitor work properly.

The following two sections organise the approaches distinguishing between *runtime enforcement* and *healing* techniques.

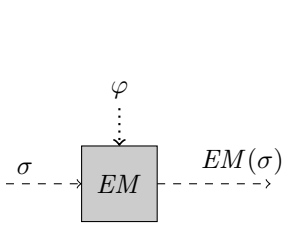
3 Enforce the Correct Behaviour

We overview some of the research efforts in the domain of *runtime enforcement* [35,59]. Runtime enforcement is a “branch" of runtime verification focusing on preventing and reacting to misbehaviours and failures. While runtime verification generally focuses on the oracle problem, namely assigning verdicts to a system execution, runtime enforcement focuses on ensuring the correctness of the sequence of events by possibly modifying the system execution.

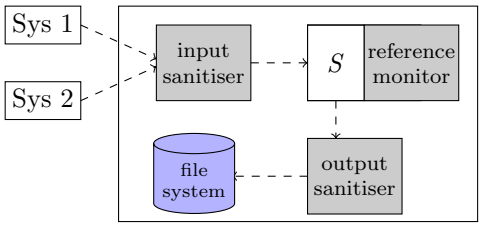
Structure of this section. The rest of this section is organised as follows. Section 3.1 introduces runtime enforcement and presents how it contributes to the runtime quality assurance and fits into the general software architecture presented in Section 2. Section 3.2 overviews the main existing models of enforcement mechanisms. Section 3.3 focuses on the notion of enforceability of a specification, namely the conditions under which a specification can be enforced. Section 3.4 presents work related to the synthesis of enforcement mechanisms. Section 3.5 discusses some implementation issues and solutions, and presents some tool implementations of runtime enforcement frameworks.

3.1 Introduction and Definitions

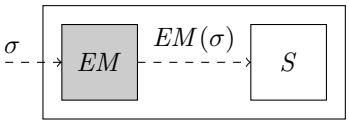
Research efforts in runtime enforcement generally abstract away from implementation details and more precisely on how the specification is effectively enforced on the system. That is, in regard of Figure 1, one generally assumes that sensors and effectors are available by means of instrumentation and one focuses on the Analyse and Plan phases instead of the Observe and Execute ones. Moreover, runtime enforcement problems revolve mainly on defining *input-output relationships* on sequences of events (see Figure 2a). That is, the actual execution, made available through the Observe module, is abstracted into a sequence of events of interest (according to the specification). More precisely, a runtime enforcement framework shall describe how to transform a (possibly incorrect according to the specification) input sequence of events into an output



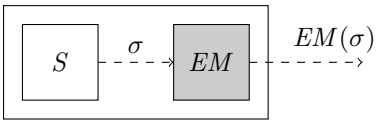
(a) Conceptual view of the runtime enforcement problem



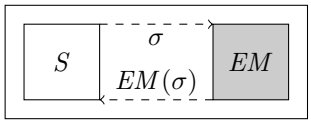
(b) Scenario illustrating the possible usages of an enforcement mechanism



(c) EM as input sanitiser



(d) EM as output sanitiser



(e) EM as reference monitor

Fig. 2: Illustration of the runtime enforcement problem: an enforcement mechanism EM transforms an input σ to an output $EM(\sigma)$ according to property φ and for a system S.

sequence of events by means of a so-called *enforcement mechanism*.[4] The transformation is performed according to the given specification which is used to synthesise the enforcement mechanism.

Before elaborating on the different ways an enforcement mechanism can transform the input sequence and how the enforcement mechanism can be synthesised (in Section 3.2 and Section 3.4 respectively), we relate the implicit assumptions made in runtime enforcement endeavours to the architecture of a general autonomic manager (Figure 1). In particular, one should note that the conceptual presentation of the runtime enforcement problem in Figure 2a abstracts away several architectural setups. We present some examples of more concrete architectural setups and illustrate them on a scenario in Example 1. First, an enforcement mechanism can be used for *input sanitisation* (see Figure 2c). In such a case, the mechanism is used to "protect" the system from its (untrusted) environment. All inputs to the system shall enter first the enforce-

[4] We follow the terminology of [40] which generalises previous terminologies used in runtime enforcement. We use the term enforcement mechanism to encompass definitions of mechanisms dedicated to enforcement described at different abstraction levels. Moreover, using the term enforcement mechanism allows us to abstract away the architecture of the autonomic monitor and its placement w.r.t. the monitored system.

ment mechanism which filters out those that could harm the system or ensure that all the necessary inputs are provided to the system. Examples of such situations include using the enforcement mechanism as a firewall (to discard or alter some inputs) or using it to ensure that the pre-conditions required to use the system are met when, for instance, the system is supposed to receive inputs from two external parties. Second, an enforcement mechanism can be used for *output sanitisation* (see Figure 2d). All outputs of the system shall enter first the mechanism which filters or transforms them. Examples of such situations include using the enforcement mechanism to prevent leaking of sensitive information or a transformation of the trace produced by the system. Third, an enforcement mechanism can be used as *reference monitor* (see Figure 2e). This architecture is close to the one of the autonomic monitor presented in Figure 1. There is a closed loop between the system and the enforcement mechanism. All actions of interest or relevant state changes are first submitted to the enforcement mechanism which then grants, denies or alters state changes. Examples of such situations include using the enforcement mechanism to grant access to sensitive primitives or system operations.

Example 1 (Using enforcement mechanisms). Consider the example system S depicted in Figure 2b where enforcement mechanisms are used to enforce the correct behaviour and ensure quality at runtime. Let us assume that S is purposed to realise some behaviour based on services provided by external systems Sys1 and Sys2. Actions of S are driven by some users (not depicted in Figure 2b) and the actions should be logged to a file system. The input sanitiser is used to forward to S information only when both Sys1 and Sys2 provide the expected service, possibly discard or reformat some information from the users. The reference monitor is used to monitor the important actions of S by for instance rescheduling the actions or not letting S execute some actions when these are not allowed. The output sanitiser is used to ensure that actions are logged properly by enforcing a pre-defined log format, anonymising user sensitive information, or discarding irrelevant information.

The input-output relationship realised by the enforcement mechanism should fulfill the following constraints.

- *Soundness:* the output sequence should be correct w.r.t. the specification.
- *Transparency:* a correct input sequence should not be modified, if possible.[5]

Remark 1 (Runtime enforcement vs supervisory control theory). Runtime enforcement share the same objectives with supervisory control theory, which was introduced by Ramadge and Wonham [81,82]. In supervisory control theory, one uses an automaton modelling the system to synthesise a *supervisor* and a list of forbidden states. Events of the system are partitioned into the so-called controllable and non-controllable events. Intuitively, the supervisor is composed with an automaton model of the system (synchronous product) and ensures the most permissive behaviour of the initial system while preventing bad behaviour (rejected by the automaton). Should the system try to execute

[5] This is the notion of transparency adopted in a majority of papers on runtime enforcement. Some research efforts notice that this notion of transparency only constrains correct execution sequences; and they advocate that constraints should be placed on how an enforcement mechanism transforms incorrect execution sequences [11,12,58].

References	Models of enforcement mechanisms	Specification formalisms used for synthesis
[88]	security automata	Büchi automata
[62]	edit-automata	deterministic finite-state automata
[37]	generalised enforcement monitors	Streett automata
[20]	edit automata	Rabin automata
[76]	delayers	timed automata
[40]	delayers with suppression	timed automata
[65]	security automata	μ-calculus formulae
[42]	generalised enforcement monitors	labelled transition systems
[39]	enforcement mechanisms with rollback	finite-state automata
[15]	safety shields	safety automata
[92]	shields for burst errors	temporal logic (safety)
[13]	iteration suppression automata	deterministic finite-state automata

Table 1: Summary of existing models of enforcement mechanisms with the specification formalism from which they can be synthesised.

an action that could lead the system to exhibit a bad behaviour, the supervisor disables this action which then cannot execute on the system anymore.

3.2 Models of Enforcement Mechanisms/Monitors

The first model of enforcement mechanism was *security automata* (SA) [88]. An SA is a finite-state machine that executes in parallel with the monitored program. Whenever the target programs want to execute an action in the scope of the enforced property, two cases arise. Either the transition is defined and then the SA lets the target system execute the action, otherwise the target system is halted. We note the follow up work [50] which corrects and extends the results in [88] related to the enforcement abilities of security automata (see Section 3.3).

Ligatti et al. later extended the work of Schneider et al. by noticing that security automata are (only) sequence recognisers. They propose the model of *edit-automata* (EA) [62] which are sequence transformers. In addition of halting the target system, edit-automata can insert and suppress actions (originating from the target system or not). For instance, an EA can suppress and memorise an action of the target system for later replay. In an EA, the memorisation of actions is realised using the state-space. Several variants of edit-automata have been proposed [11].

Falcone et al. generalised edit automata with the so-called *generalised enforcement monitors* (GEMs) [43]. Contrarily to EAs, a GEM clearly separates sequence recognition from sequence transformation: GEMs are based on finite-state machines extended with generic enforcement operations that act on an internal memory. Separating sequence recognition from action memorisation has several advantages. First, GEMs are more amenable to implementation. Second, one can define easily formal composition operations on GEMs by computing the product state space and composing memory operations.

Bielova and Massacci proposed Iterative Suppression Automata (ISAs) [13], as a variant of EAs. They noticed that the usual requirements of soundness and transparency (and their implementation with EAs) do not distinguish what should happen when the input execution does not satisfy the specification. The underlying motivation is to be able to compare EAs in the manners they intervene on incorrect executions.

As noticed in [37], EAs and GEMs suffer from a practical limitation. Both of these models assume being able to freeze an unbounded number of actions to be replayed later. This amounts to assuming that an enforcement mechanism is able to predict the result of any action. To address this issue, Dolzhenko et al. introduce Mandatory Results Automata (MRAs) [64,30]. Upon the observation of any action, an MRA should return a result to the target application before seeing the next action. An MRA is placed between the untrusted target application and the executing system, and enforces the actions executed by the target as well as the results returned. Then, an MRA has to consider input and output events on traces.

In [24,39], Charafeddine et al. propose enforcement mechanism with k-step *roll-back* abilities. Such enforcement mechanism allows the system to deviate from the desired property up to k observable execution steps. Should the system not return to a correct state after k steps, the enforcement mechanism rolls the (non-deterministic) system back to the last correct state and forces it to explore alternative executions. An instantiation with 1 step of such general definition enforcement mechanisms is then implemented and integrated in component-based systems (cf. [6]).

Similar to the above models are the so-called *safety shields* [15] for reactive hardware systems, i.e. systems with Boolean signals as inputs and outputs. A shield is a Mealy machine which ensures soundness and minimum interference according to a notion of distance measuring the deviation between the output and the input of the shield. When a state where a property violation becomes unavoidable is reached, the shield enters in a *recovery* period, called *k-stabilisation*, and is allowed to deviate from its input for at most k consecutive steps. Bloem et al. assume here that the violation should be a rare event, and then the monitor keeps track of all possibilities assuming that it was an isolated error. If another violation arises during this recovery period, the shield enters in a *fail-safe* mode, where correctness is still ensured, but no minimal deviation. Note, a shield cannot buffer events. Wu et al. extends shields and propose enforcement mechanisms that respond immediately to violations and guarantees the safety under burst errors [92]. Similar to the model in [24], k-stabilising shields (which recover the system in a finite time) and admissible shields (which collaborate with the system) are introduced in [52].

Models with memory constraints. Most of the above models of enforcement mechanisms are endowed with an infinite memory as they allow the possible memorisation of an unbounded number of events. Several models have been proposed to account for practical memory limitations and bound the memory needed by enforcement mechanisms. Fong proposed Shallow History Automata (SHAs) [44] as security automata that do not keep track of the order of event arrival. Fong generalised SHA as α-SA which are SA endowed with a morphism α abstracting the current input sequence. Talhi et al. introduced Bounded Security Automata (BSAs) and Bounded Edit-Automata (BEAs) [91]. BSAs and BEAs are SAs and EAs with a bounded memory to memorise

the input sequence respectively. The previous models bound the size of the memory of the enforcement mechanism (with an integer). Beauquier et al. introduced finite EAs and deterministic context-free EAs, that is EAs with a finite set of states [10]. They prove that finite EAs are strictly less expressive than EAs and study the conditions under which a property can be enforced by a finite EA.

Models with real-time enforcement primitives. The previously described models of enforcement mechanisms feature untimed sequence recognition mechanisms and enforcement primitives. In particular, when they do not account for the time that elapses between the occurrence of two received events. Moreover, the amount of time during which an event remains in the memory of the enforcement mechanism is not taken into account. Models for enforcing timed properties have been defined as *delayers* in [77,76] to enforce timed properties. Such models account for the physical time elapsing during the reception of actions, storing and releasing actions in real-time. Later, the model of delayers has been extended into *delayers with suppression* [40] where actions are discarded from the memory when releasing such events would irremediably make the underlying property violated. Since physical time has consequences on the implementability of enforcement mechanisms, soundness and transparency need to be redefined and additional constraints such as optimality are required on how such enforcement mechanisms release actions.

Models supporting uncontrollable events. Closer to controllers in supervisory-control theory (see Remark 1), enforcement mechanisms accounting for uncontrollable actions (i.e., actions that cannot be affected by the enforcement mechanism) have been defined [85,57]. In addition to the current satisfaction of the output execution, such models take into account the possible reception of uncontrollable events. Uncontrollable actions as clock ticks were first introduced by Basin et al. in [5]. Unrestricted uncontrollable actions were later introduced in extensions of GEMs in [85,84,86] and of EAs in [57].

Predictive enforcement mechanisms. Inspired by the predictive semantics of runtime verification monitors [94], predictive enforcement mechanisms were proposed in [78,79]. Predictive enforcement mechanisms leverage some apriori knowledge of the system to output some events faster, instead of delaying them until more events are observed (or permanently).

3.3 Enforceable Specifications

We now turn our attention to the existing characterisations of the so-called *enforceable* specifications, i.e., specifications that can be enforced. Before elaborating on the existing characterisation, we first narrow down the term specification. As suggested by Schneider [88], one can distinguish properties from policies when specifying systems. A property (can be seen as a predicate that) partitions individual executions, while a policy (can be seen as a predicate that) partitions sets of executions. Hence, not all policies are properties. When observing a system execution, it is possible to determine

the membership to a property; while determining membership to a policy generally requires observing additional executions.[6] Examples of properties include deadlock and starvation freedom, fairness, access control constraints, formalised requirements over executions. The classical example of policy (which is not a property, i.e., it can not be expressed with predicates over single execution) is information-flow because it requires checking for potential correlation between executions.

Enforceability of a property depends on several factors:

- the formalism used to specify the property, and more particularly whether the formalism describes finite or infinite executions;[7]
- the enforcement primitives endowed to the monitors and how these enforcement primitives are mapped to actual system effectors;
- constraints stemming from the system in which enforcement monitors are to be integrated.

In the pioneering work of Schneider on security automata, safety properties were characterised as enforceable [88]. Since a security automaton can only either 1) let a system action execute or 2) halt permanently the system, its decisions are irremediable. Concurrently, Kim et al. noticed that any monitoring mechanism (evaluating the execution of a system against a property) should be able to determine if the current execution is outside the set of allowed executions [60]. Thus, properties should be also *co-recursively enumerable*, that is, the non-membership test should be computable. We note that the results in [88] were later refined in [50][8], with the insights given in [60].

Ligatti et al. proved that, compared to security automata, using the additional enforcement primitives, edit-automata can enforce the so-called *renewal* properties [8,62,63]. In the safety-liveness classification of properties [71], renewal properties form a superset of safety properties which contains some liveness properties. Intuitively, a property is a renewal if a) any infinite execution sequence in the property contains infinitely many prefixes in the property, and b) any infinite execution sequence not in the property contains only finitely many prefixes in the property. Falcone et al. proved that Generalised Enforcement Monitors instantiated with the *store* and *dump* operations, which respectively memorise and release events, can enforce the so-called *response* properties [38] in the Safety-Progress hierarchy of properties [21]. Response properties are properties

[6] We note that some ongoing research efforts study *hyper-properties* [26], which resemble policies. We also note ongoing work advocating monitoring hyper-properties [16].

[7] As was the case in runtime verification, early work on runtime enforcement considered infinite executions.

[8] Hamlen et al. [50] additionally introduce the notion of *RW-enforceable* policies (policies enforceable by enforcement mechanisms with Program Rewriting abilities), and use it to define a more precise characterisation of enforceable security policies. They model the untrusted programs as Türing machines with deterministic transition relations with three infinite-length tapes. They divide enforcement mechanisms into three categories: static analysers, reference monitors, and program rewriters. Static analysers operate strictly prior to running the untrusted program. Reference monitors intercept events or actions the program under scrutiny and intervene before occurrence of an event violating the policy, by terminating it or applying some other corrective action. Program rewriters modify in a finite time the program under scrutiny prior to execution.

for which some expected good behaviour should happen infinitely often. They can be intuitively understood as repeated transactions.

Moreover, we note that on finite sequences all properties are renewals. This observation is in line with the fact that (pure) response properties coincide with renewal properties, as noticed in [38].

Ligatti et al. proved that the MRA approach permits the enforcement of a new variant of properties, named *result-sanitization* or *monitor-centric* policies which are simpler and more expressive than usual definitions (*target-centric* ones). They also provide a hierarchical characterisation of the policies enforceable or not with MRAs. For instance, they show that MRAs precisely enforce a strict subset of safety properties, whereas Non-deterministic MRAs (NMRAs) precisely enforce a strict superset of safety properties. Depending on the definition chosen for non-safety properties or with additional assumptions, MRAs can also enforce some non-safety properties.

Falcone and Jaber [24,39] showed that stutter-free safety properties are enforceable on component-based systems with monitors that can roll the system back by one observable execution step. Stutter-invariance is required on properties because of constraints stemming from the nature of synchronisation of components. A hierarchy of enforceable properties according to the number of steps the enforcement mechanism can roll the system back (the so-called k-step enforceability) is defined [39]. While 1-step enforceable properties are characterised, a general characterisation of k-step enforceable properties is left open.

Basin et al. extend the characterisation given in [88,50] of enforceable properties by additionally considering a universe of possible (input) traces and a set of controllable actions [5]. A property is enforceable if it is a safety and is such that violations are not caused by uncontrollable actions, and the set of prefixes of sequences in the universe and the property is decidable.

3.4 Synthesising Enforcement Mechanisms

We now report on some of the existing techniques used to synthesise enforcement mechanisms from properties described in several specification languages/formalisms (Table 1, p. 7, gives the specification formalism from which each type of enforcement mechanisms can be synthesised). Schneider et al. synthesise SAs from Büchi automata [88]. Ligatti et al. synthesise EAs from deterministic finite-state automata describing renewal properties [62]. Falcone et al. synthesise GEMs from Streett automata [37]. Chabot et al. synthesise EAs from Rabin automata [20]. Pinisetty et al. synthesise delayers in [74,73,76] and Falcone et al. synthesise delayers with suppression in [40], from timed automata. Using partial model-checking techniques, Mateucci and Martinelli synthesise SAs from μ-calculus formulae [65]. Enforcement mechanisms are described as algebraic operators driven by controller programs. Falcone and Marchand synthesise GEMs from labelled transition system marked with secret states to enforce opacity properties [42]. Charafedine et al. transforms deterministic finite-state automata into enforcement mechanisms with 1-step roll-back abilities and integrate them into a component-based system [24,39]. Bloem et al. synthesise safety shields from safety automata by solving 2-player safety games [15]. Wu et al. synthesise shields that handle burst errors using a game-based algorithm [92]. Bielova and Masacci adapt the

construction of EAs to synthesise a variant called iteration suppression automata for iterative properties described by deterministic finite-state automata. Iterative properties are such that the good executions are formed of “iterations" that can repeat an arbitrary number of times.

3.5 Implementations and Applications

The principles of runtime enforcement have been implemented and applied to several domains. Most of these approaches are based on either SAs, EAs, or GEMs.

Tool implementations. While there is a plethora of tools for runtime verification [4], there are only a few tool implementations for the runtime enforcement of properties on systems: Polymer [9], Mobile [49], TiPeX [75], and more recently GREP [86,83]. Polymer is a language and system for the definition and composition of enforcement mechanisms for Java applications. Mobile is a language-support for verified enforcement on .NET. Whenever a Mobile program type-checks with respect to a security policy, it is guaranteed that the program respects the policy. TiPEX implements algorithms for enforcing timed properties described as timed automata. TiPEX enforcement mechanisms correct input sequences by delaying actions. GREP also implements algorithms for enforcing timed properties described as timed automata with the ability to handle uncontrollable events. These algorithms are based on game theory.

We note that runtime verification tools can perform for free basic form of runtime enforcement as in security automata by halting the target system whenever a violation of the property occurs. Java-MOP [25] is a tool for runtime verification which arguably provides some support for ad-hoc runtime enforcement. Java-MOP provides self-recovery mechanisms in case of violation in the form of *handlers*. Handlers are code snippets that can be integrated in the target program in order to handle the violation (or validation) of a property using contextual execution information retrieved using aspect-oriented programming.

Application domains. One of the first domains of application is the security domain; and enforcement mechanisms were initially defined as security devices. Runtime enforcement was applied to enforce security policies [34], availability requirements in [28], privacy policies [54] and [61], opacity properties in [41,42], role-based access control security policies in [72], usage-control policies [66], the confidentiality of artifacts in [48]. There is also a body of work applying runtime enforcement principles on mobile devices such as Android-based mobile phones [36,67,68,1,31,69].

Limitations of enforcement. Even if a system is equipped with an enforcement mechanism, it may reach a failing situation. This is due to several reasons. Firstly, the enforcement mechanism considers only the described property and then acts according to this latter only. Any other (maybe unexpected) event not taken into account by the property may lead to a failure. Moreover, an enforcement mechanism has a restricted enforcing power since it follows a specific set of rules. If the necessary correcting action is not included in this set, then a failure may arise. As shown by [85], there are some situations in case of uncontrollable events where it is not possible to avoid an incorrect situation.

Finally, there may be a gap between the abstract description of the enforcement mechanism and its real implementation. In this case, this is more a problem of instrumentation of the approach.

4 Healing Failures

In this section, we discuss techniques that can be used to heal executions after the failure has been observed by the autonomic monitor. In this context, a failure is defined as an execution that deviates from the intended semantics of the monitored program.

In order to automatically react to failures, it is first necessary to detect them. We can distinguish between domain independent failures, that is, failures that do not depend on the specific semantics of an application (e.g., crashes, uncaught exceptions, and deadlocks) and domain-dependent failures, that is, failures that depend on the specific semantics of the system (e.g., the generation of a wrong output).

Domain-independent failures can be trivially detected with an implicit oracle, that is, an oracle that can simply recognise the event that represents the failure (e.g., the application that quits abruptly, an exception reported in a log file, and the application that stops responding). Domain-dependent failures can be detected with program-specific oracles, that is, oracles obtained from a definition of the semantics of the program. These oracles detect failures by comparing the observed behaviour to the behaviour defined in the specification, for instance, an oracle might be obtained from a logical specification of the input/output behaviour of the system to detect incorrect outputs [3].

Detecting failures is a responsibility shared between the Observer and the Analyser components of the Autonomic Monitor architecture introduced in Section 2. The Observer is responsible for collecting events that can be processed by the Analyser to establish if the monitored program has failed. In the case of domain independent failures, the Observer has to simply detect the events that characterise failures and notify them to the Analyser, which reacts by triggering the healing process. In the case of domain dependent failures, the Observer collects the events relevant to the classes of failures that can be recognised, while the Analyser processes these events based on its Knowledge of the expected behaviour of the system. If a mismatch between the expected behaviour and the actual behaviour of the monitored program is observed, the Analyser triggers the healing process.

The healing process is driven by the Planner that activates appropriate mechanisms, based on its knowledge of the system and the available strategies, as described in the rest of this chapter. The Executor concretely actuates the plan elaborated by the Planner.

Since techniques that react to failures can affect a monitored program only after a failure has been observed, they need to incorporate mechanisms to either *rollback* the execution to a safe point before the failure happened, to successively influence the execution preventing any failure, or to *compensate* the effect of an observed failure moving the monitored system to the same state that would be observed if the failure has never happened.

Techniques for reacting to failures rely to one of the following three main sources of information: knowledge about the *redundant* elements of a system that can be exploited to workaround a failure, knowledge about the *actions* that can be taken to react to some

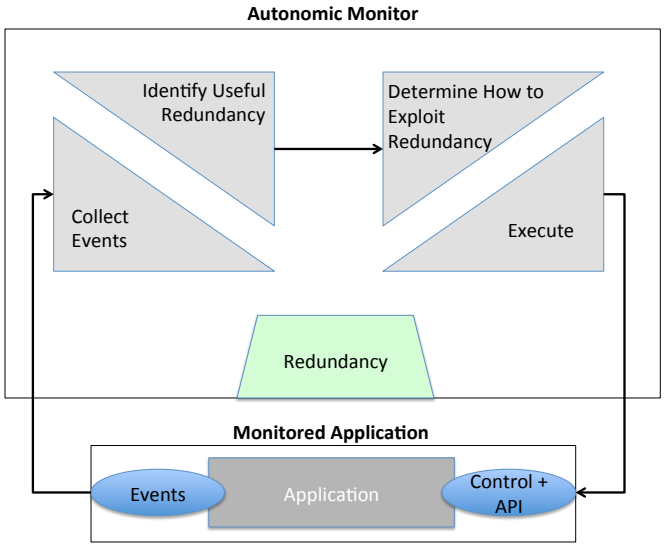


Fig. 3: General architecture for reacting to failures exploiting redundancy

specific types of failures, and knowledge about the *actions* that can be taken to explore program and configuration variants in response to an *unknown type of failure*.

The rest of this section presents the techniques for reacting to failures organised according to the knowledge that is exploited for the reaction: Section 4.1 presents techniques that exploit the knowledge of the redundant elements, Section 4.2 presents techniques that exploit the knowledge of specific failure types, and Section 4.3 presents techniques that exploit the knowledge of the existing program variants.

4.1 Techniques that Exploit Redundancy

We say that two processing units (e.g., two components or two code fragments) are functionally redundant if they produce the same outputs for the same inputs. Note that redundant units are allowed to show behavioural differences, for instance in their internal structure, or in their non-functional characteristics, such as performance and usability. A monitored program may include functionally redundant units, either introduced intentionally or incidentally. Techniques relying on redundancy may exploit both forms of redundancy.

Techniques based on *explicit redundancy* exploit the redundant elements intentionally introduced into the monitored system, such as the multiple redundant copies of a same fault-tolerant component, to workaround failures, while the techniques based on *intrinsic redundancy* exploit the redundant elements incidentally present in the monitored system to workaround failures. An example of incidentally redundant elements is the case of two different functionalities that, although not designed to be redundant,

might be used to achieve the same result in specific situations (e.g., for some specific inputs).

The general architecture of the techniques exploiting redundancy is shown in Figure 3. The sensor sends *events* from the monitored application to the Observer. The types of collected events might change depending on the specific technique and application, for instance they could be method invocations or http requests. The Observer *collect*s these events, maintains the relevant parts of the history of the execution and intercepts failure signals, such as crashes and uncaught exceptions.

When a failure is detected, the Analyser matches the collected sequence of events with its knowledge of the *redundancy* of the system to *identify the useful redundancy*, that is, the redundant units that might be exploited to avoid the failure. If some useful redundancy is present in the system (e.g., a redundant copy of a component or a functionality involved in the failure), the Planner has to *determine how to specifically exploit the redundancy* in the system to avoid the failure. For instance, the Planner may decide to transfer the execution to another component or to rollback the execution to a safe point and then execute a redundant copy of the operation that has failed. The Executor concretely *execute*s the plan, exploiting its knowledge of the implementation of the system. The Effectors are the elements that support the execution of the plan within the target application, that is, the Executor interacts with them to run the plan. They usually consist of the *API* of the monitored system, sometime suitably extended with mechanisms to *control* the execution, for instance to transfer the execution across components or to rollback executions.

In the rest of this section we discuss some techniques exploiting these two forms of redundancy.

Explicit Redundancy. A well-known way to tolerate failures is through the deployment of multiple redundant components into the same system. The general intuition is that if a component fails, the failure might be worked around by transferring the execution to a redundant copy of the same component. This solution has been extensively investigated in the context of fault-tolerant systems, especially in N-version programming [2].

In addition to classic fault-tolerance, there are other ways of taking advantage of the redundancy explicitly introduced into a monitored program. In particular, recent approaches investigate scenarios that might be less effort-demanding than N-version programming, which requires the independent implementation of multiple copies of the same component. An interesting approach is the one investigated by Hosek and Cadar [51], who exploited the multiple versions available for the same program to automatically react to failures caused by faulty software updates.

The key idea is to maintain alive both versions of a software system after an update. The two versions are then executed side by side and when a failure is experienced in one version, the other one is exploited to overcome the failure. To achieve this capability, the execution of the two versions must be monitored and synchronised. The monitor collects and compares the system calls performed by the monitored programs. When a diverging behaviour is observed, appropriate actions are taken. The execution is also synchronised, that is one version cannot proceed with the execution until the other version has produced the same system call. In this way, the execution might be timely switched from one version to the other.

A divergent behaviour might produce different reactions depending on the kind of divergence. If the two versions produce different system calls, the result produced by one version is simply preferred to the other, for instance the new version of the system might be preferred to the old one. If one program crashes, the approach performs a lightweight rollback to the last system call, executes the code in the other version until the next system call is produced, and then continues with the execution of the version that produced the failure. This strategy de facto reuses the code in the other version to avoid failures, and it might be effective to overcome bugs introduced with faulty upgrades. Note that this explicit form of redundancy does not require special effort to be generated because it is naturally introduced with the evolution of a software system.

Intrinsic Redundancy. Since intrinsic redundancy is not documented explicitly, discovering the intrinsically redundant operations might be hard and expensive, indeed it is undecidable in general. The effort required to discover these elements is compensated by the possibility to augment systems that have not been designed to react to failures with the capability to handle them.

Intrinsic redundancy can be extracted in various ways, for instance using testing and analysis techniques [45], and can be suitably integrated with mechanisms to either rollback executions or compensate the effect of failures to obtain systems with high reliability. When integrated with *rollback* mechanisms, failures can be handled by first bringing the execution back to a safe point and then running an intrinsically-redundant alternative operation with the one that has failed [18]. When integrated with *compensation mechanisms*, failures can be handled by first compensating their effects, if any, and then again executing an alternative operation intrinsically redundant with the one that has failed [19].

The knowledge of the intrinsically redundant operations can be encoded using rewriting rules, which associate a sequence of operations to another sequence of operations that has the same observable behaviour of the original sequence. As discussed in [19], examples of intrinsically redundant operations typically present in container classes are:

`addAll(a,b)` $\rightarrow$ `add(a); add(b)`

`add(a)` $\rightarrow$ `addAll(a,b); remove(b)`

The first rule indicates that adding the elements `a` and `b` using the `addAll` method produces the same effect as adding first `a` and then `b` using the `add` method. Alternatively, the second rule indicates that adding element `a` with the `add` method produces the same effect as adding the element `a` and an element `b` using the `addAll` method and then removing `b`.

When a failure is detected, the sequence of the operations performed by the monitored program is analysed, checking if any rewriting rule can be exploited to change the failing sequence into an alternative sequence. The intuition for exploiting intrinsically redundant operations is that a failing execution might be worked around by replaying the execution using some alternative but equivalent operations. For instance, if a failure has been observed when running the sequence of operations

```
newList(); addAll(a,b)
```

the sequence might be automatically replaced with the alternative sequence

```
newList(); add(a); add(b)
```

using the first rewriting rule.

Note that the rewriting rules above allow substituting a sequence of operations with alternative, but equivalent, sequences of operations that do not share any operation with the original sequence. Avoiding to reuse operations executed during the failure intuitively increases the probability to produce a new sequence that does not fail.

If the opportunity to workaround the failure is detected, the planner elaborates a suitable strategy, which could be based either on rollback or on compensation mechanisms. If multiple rules could be exploited, the plan may attempt to execute a sequence of rollback/compensation operations followed by the execution of a rewritten sequence until the failure is overcame or no more options are available. The order of application of the rules might be based on historical information, giving precedence to the rules that have been most successful in the past.

The choice of using rollback or compensation before executing a rewritten sequence of operations depends on the nature of the system that must react to errors. For instance, rollback has been used to overcome failures in container classes [18], while compensation has been exploited with Web applications where it is often sufficient to reload a Web page to cancel the effect of a failure [19].

In general, not all the systems can be addressed with rollback or compensation mechanisms. For instance, the state of a system might be too large, complex and difficult to observe and control to be rolled back. Similarly, the impact of a failure may have consequences that cannot be cancelled by any other system operation. However, when at least one of the two approaches can be feasibly applied to a software system, the system could be potentially extended with healing capabilities.

4.2 Failure-Specific Techniques

Failure-specific techniques exploit the knowledge about some specific classes of failures to effectively recognise and react to them. These techniques have a narrow applicability compared to techniques addressing broader classes of failures, such as the ones based on redundancy (see Section 4.1) and the ones exploring variants (see Section 4.3). However, when an observed failure is in their scope, they can be dramatically effective.

The general architecture of failure-specific techniques is shown in Figure 4. The sensor sends *events* from the monitored application to the Observer, which *collect*s these events, maintains the relevant parts of the history of the execution, and intercepts failure signals, such as crashes and uncaught exceptions. When a failure is detected, the Analyser matches the failure and the collected sequence of events with the known failure types. If the observed failure matches with some known failure types, the corresponding reactions are retrieved. The Planner is then responsible for defining a strategy to apply the selected reactions, contextualising them to the monitored program, if needed, and defining their order of application. The Executor concretely *execute*s the plan, exploiting its knowledge of the implementation of the system. The Effectors are the elements that support the execution of the plan, that is, the Executor interacts with the Effectors when running the plan. In this case, the Effectors usually consist of the *API* of the monitored system whose operations are invoked while applying a selected reaction.

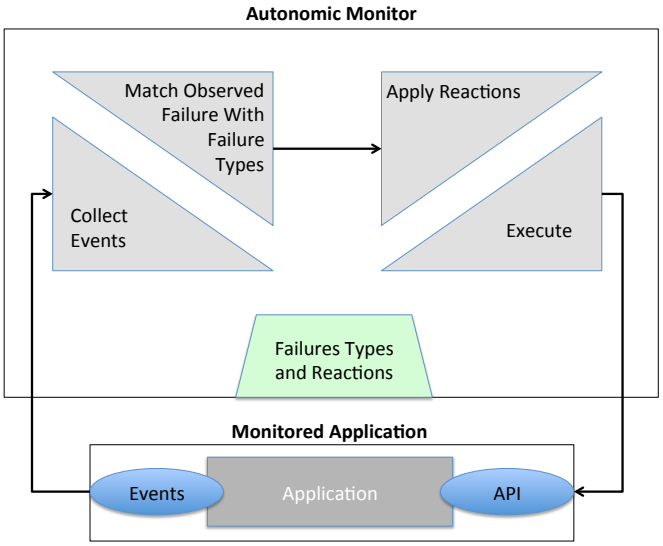


Fig. 4: General architecture for reacting to specific failures types

In the following, we present two failure-specific techniques, one addressing a pre-defined set of failure types, and another that can dynamically learn how to react to failures based on a set of samples.

Pre-defined Failure Types. Techniques addressing pre-defined failure types are techniques designed to handle specific situations in specific systems. A notable example is the case of healing connectors [23,22], which are connectors that can be deployed on a component-based system to react to failures caused by incorrect interactions between components.

Healing connectors implement reaction mechanisms that are activated when a component throws exceptions that should not be raised. To react correctly and efficiently, they exploit the *knowledge* of how the interaction between components may fail due to some specific classes of integration problems that may result in some exceptions. When an exception is caught, healing connectors check if the cause of the exception is a known problem, and if it is the case, they apply the pre-defined reaction. The reactions may follow four patterns [23]: parameter translation, component preparation, alternative operation, and environmental changes.

Parameter translation can be used to react to the failures caused by the use of a wrong parameter value in the invocation of an operation. It reacts by replaying the failed interaction while replacing the incorrect value with the correct one. For instance, parameter translation can be used to automatically fix a wrongly encoded URI stored in a parameter.

Component preparation can be used to react to the failures caused by the components that produce exceptions because they are in a state that is not suitable to accept

a given request. It reacts by replaying the failed interaction after having modified the state of the component. For instance, component preparation can be used to initialise an uninitialised component.

An alternative operation can be used to react to the failures caused by the use of a specific faulty operation. It reacts by replaying the failed interaction while replacing the faulty operation with an alternative operation, similarly to methods exploiting redundancy. For instance, alternative operation can be used to replace the invocation of a deprecated method with the invocation of an up-to-date method.

Finally, an environmental change can be performed to react to the failures caused by problems in the environment. It works by replaying the failed interaction after having modified the environment in a way that may prevent the failure from occurring again. For instance, an environmental change can be used to create the missing folders that cause an application to fail.

When an uncaught exception is raised, multiple healing patterns might be eligible to react to the failure. The Planner is responsible for organising the applicable patterns in a pipeline. Healing connectors do not require special effectors, but they simply take advantage of the API of the monitored program. If necessary, depending on the failure, they may incorporate actions to compensate the effect of a failure so that the failed interaction can be safely re-executed.

Sample-Based Approaches. Sample-based approaches exploit the *knowledge* of how failures have been (manually) handled in the past to automatically react to new occurrences of the same failures [29]. They thus rely on the assumption that a repository of failures and corresponding countermeasures is available.

Sample-based approaches are failure-specific because they can only address the failures that have been observed in the past. However, the set of supported failures is changed every time by simply providing a different set of samples to learn from, potentially increasing the generality of the technique.

Comparing actual failures to sample failures is challenging because failures caused by the same problem are never exactly the same. The same failures may occur in many different circumstances, such as different states of the system, different inputs, and in different environment conditions. To match a pair of failures, sample-based approaches distill a signature that characterises a failure regardless of the specific circumstances in which it occurred. In their work, Ding et al. [29] apply concept and contrast analysis to the log files collected during failures to produce signatures that characterise failures in terms of the key events reported in the log files.

Signatures are derived for both the sample failures and the newly observed failures. When an observed failure has a signature matching the signature of a failure in the repository, an appropriate reaction can be automatically extracted from the repository and executed.

Reactions that have been taken in the past necessarily refer to a specific situation. For instance, they may concern rebooting specific machines and changing specific configurations. When the same failure is observed, it might occur in slightly different circumstances, which may require slightly different reactions. For instance, if in the past machine `hostA` has been rebooted because it stopped responding, and in the actual execution the machine that is not responding is machine `hostB`, the reboot operation

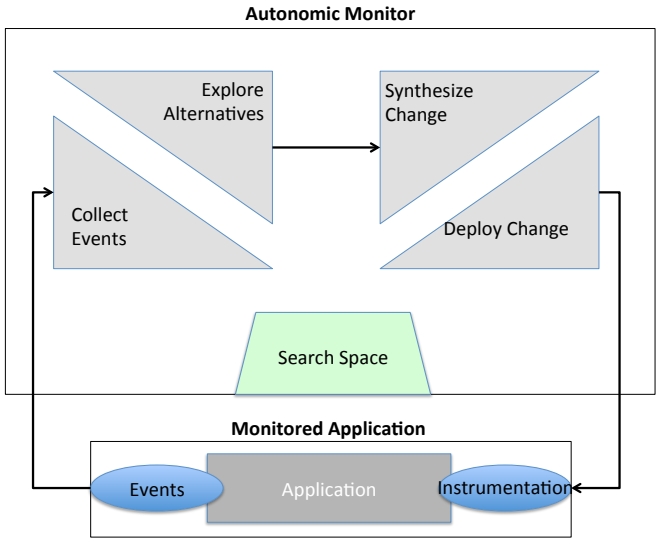


Fig. 5: General architecture for reacting to failures by exploring variants

should be executed on `hostB` and not on `hostA`. The approach described in [29] can achieve this capability by executing an operation called contextualisation of the reaction. Contextualisation extracts the parameters used in the sample reaction (e.g., the name of the machine), matches the observed failure with the sample failure in the repository identifying the actual value for all the extracted parameters (e.g., the actual name of the machine that is not responding), and executes the reaction replacing parameters with actual values.

This strategy has been mostly experienced with large service-based systems to turn the manual reactions executed in the past by the operators into automatic reactions, reducing maintenance cost and increasing system reliability.

4.3 Techniques Exploring Variants

Techniques that react to failures by exploring variants are not usually explicitly tied to any class of failures. The general intuition is that these techniques may try to replay a failing execution many times until finding a change that might be operated on the monitored program to prevent the failure without breaking the other functionalities of the system. The change might be either on the configuration [90] or in the code [46] of the monitored program. Of course, a suitable environment is needed to replay an execution many times regardless of the side-effects that might be introduced by a failing execution. For this reason, these approaches must have access to a protected environment where a copy of the application can be executed many times until finding a solution that can be deployed on the real instance of the program.

These techniques do not require any specific knowledge about the failures that may occur on the target system, but they require to know how to explore the space of the possible variants. For instance, they need to know what the space of the possible configurations looks like, to be able to systematically execute a program with different configurations, or they need to know how the code of a program can be modified, to explore the space of the code changes that might fix a faulty program.

The general architecture of techniques reacting to failures by looking for variants is shown in Figure 5. The sensor sends *events* from the monitored application to the Observer. The collected events usually consist of the inputs received by the monitored program and failure signals. When a failure is detected, the control is transferred to the Analyser that exploits the knowledge of the *search space* to *explore alternatives* that might prevent the occurrence of the failure. Each alternative is checked by replaying the observed failure.

Alternatives might consist of different configurations of the monitored applications or even program variants. When a suitable alternative is identified, the Planner *synthesises a change* that the Executor can *deploy* on the monitored program. The Effectors consist of mechanisms that allow the modification of either the program or its environment.

In the following, we present two techniques, one that reacts to failures by exploring alternative configurations and the other that reacts to failures by synthesising code changes.

Alternative Configurations. The assumption made by approaches exploring the space of the possible configurations of a program is that there may exist a configuration under which a failed functionality may run correctly. The strategy to find a workaround consists of transferring the control to a separate instance of the monitored program running in a sandboxed environment to replay the failed execution several times for many different configurations, until finding a configuration that makes the program pass. In order to apply this process, the knowledge of the autonomic monitor has to incorporate information about the shape of the space of all the legal configurations. Such a space must be sampled efficiently to quickly find a solution to an observed problem.

REFRACT implements this strategy using a feature model as representation of the configuration space [90]. In practice, when a failure is observed, REFRACT replays the failed execution in a separate environment and samples the configuration space described by the feature model according to three possible strategies: n-hops sampling, random sampling, and covering array sampling.

The n-hops sampling strategy systematically investigates all the configurations that can be obtained from the configuration of the monitored program by changing n options. The random sampling generates a completely random configuration. Covering array sampling considers a set of configurations that include every possible combination of values for up to t configuration options. If a configuration that prevents the failure is detected, the configuration is further modified using the delta-debugging algorithm [93] to minimise the set of changes that must be operated on the current configuration to workaround the failure.

The new configuration can then be deployed to the monitored program. If the execution in the monitored program could be suspended while waiting for a better config-

uration, the monitored program may immediately benefit from the new configuration, otherwise only future occurrences of the failure will be prevented by the deployment of the updated configuration.

Alternative Implementations. When a failure is observed, alternative implementations that may include a fix to the problem that caused the failure can be generated using automatic program repair techniques. While these techniques have been originally designed to assist developers when fixing programs, they can also be exploited to automatically react to failures, as proposed in GenProg [46]. The idea is that automatic program repair can be used to generate many tentative program fixes that are deployed and tested in a separate instance of the monitored program. The separate instance runs in a sandboxed environment to prevent the generation of any harmful side-effect. If a fix can be found, it is deployed on the original instance of the monitored program to prevent future occurrences of the same failure. If the execution in the original instance can be suspended, the fix can also be exploited to turn the currently failing execution into a correct one.

To synthesise fixes automatically, the knowledge must include information on how to change a monitored program and how to verify the correctness of the tentative fixes. In GenProg, the synthesis of the fixes is driven by a genetic programming algorithm that modifies the program exploiting single-point crossover and three mutation operators. The mutation operators can change a program by deleting a statement, adding a statement copied elsewhere from the program, and replacing a statement with another statement copied elsewhere from the program. The locations where the mutant operators should be applied to are identified using spectrum-based fault localisation [55], which can automatically assign to each statement a score representing its likelihood to be faulty. To increase the probability to produce mutations that can affect the faults in a program, the probability to mutate a statement is proportional to its suspiciousness, so that the statements that are more likely to be faulty are more likely to be modified. The verification strategy simply runs the available test suite to check the correctness of a fix, that is, a fix that passes all the available test cases is assumed to be correct.

GenProg has been exploited to react to failures produced by programs that respond to http requests (e.g., a Web server) [46]. The idea is that the monitored system can be extended with anomaly detection techniques to detect if an untrusted input causing a suspicious execution has been received. When an anomalous execution is detected, the program is suspended and the control is transferred to GenProg, which runs an automatic repair process in a separate machine. If GenProg can find a fix, that is, a change in the program that prevents the anomalous execution without causing the failure of any of the available test cases, the fix is deployed on the original program and the execution resumed. This strategy may prevent the immediate failure of the program, but also prevent any similar failure in the future.

5 Open challenges

There are still several open challenges to achieve effective failure prevention and reaction. In this section, we discuss some of the main open challenges. We note that some of these challenges apply more broadly to runtime verification.

Gap between models and software Enforcement models might be difficult to implement into the corresponding autonomic monitors because they may require a strong adaptation and instrumentation effort resulting from the gap between the abstraction used in the model and the concrete behaviour of the software. Solutions that can reduce this gap to make monitors easier to implement and reuse are necessary to make runtime enforcement more practical.

Property specification Usually the languages proposed by the tools are rather simple and do not permit to describe complex properties. Effort should be done in designing formalism in order to describe and manage more complex properties in an intuitive way.

Distributed and multi-threaded systems Nowadays, many systems are distributed and need to be observed and controlled in several points. The generation of distributed enforcement mechanisms, communicating together in a minimal way, in order to ensure a global property is still an open challenge. A similar challenge is present in the case of multi-threaded programs, where the effect of the monitors on multiple partially-independent threads must be coordinated and controlled. In both cases, it will require means to decentralise enforcement mechanisms. For this purpose, one can inspire from the decentralisation of runtime verification monitors [7,27], the monitoring of decentralised specifications [33,32] and the decentralised enforcement of document lifecyles [47].

The oracle problem In order to react to a failure, it is necessary to recognise that a failure has happened. While some failures are trivial to detect (e.g., crashes), other failures (e.g., wrong results) require a thorough and detailed knowledge of the system to be recognised. Unfortunately this knowledge is seldom available, and when it is available it is typically expensive to encode in a machine-processable form. Researchers have investigated how to automatically extract this knowledge from software artefacts produced for other purposes, but despite these early attempts, how to systematically extract and exploit such knowledge to detect non-trivial failures is still an open challenge.

Specific vs general Solutions Techniques for reacting to failures might be defined to be either general, that is, to be able to potentially address a large family of failures, or specific, that is, to be able to address a restricted family of software failures. While general approaches might be frequently useful, since they cover a broad range of situations, their effectiveness is intrinsically limited by their generality. In practice, general approaches can hardly react to a failure in an optimal way because their strategies are designed to be broadly useful. On the other hand, failure-specific approaches are useful in a limited number of cases, but they can be extremely effective when applicable. Finding a good compromise between generality and specificity in designing techniques that may optimally address an extensive number of cases is still a challenge.

Non-intrusiveness Both techniques for preventing and reacting to failures work in the field directly in the end-user environment. Any operation that is performed in the field

in the attempt to prevent or react to a failure may potentially cause even more serious consequences than the failure itself to the user data and processes. Although there are several environments providing a degree of isolation (e.g., virtual machines and containers), how to employ them in a resource-constrained environment for preventing and reacting to failures is still an open challenge. More in general, it is hard to design techniques that can prevent and react to failures providing the guarantee of not affecting the user.

Provably-correct monitoring To ensure a better confidence in enforcement mechanisms, or more generally, in the mechanisms protecting the system from faults, it is desirable to ensure that the monitoring code conforms to the property or security policy at hand. This check can then be performed (using a proof checker) by a third-party who does not necessarily trust the monitoring process. Preliminary work has been carried out on this topic in [89] to verify the soundness and transparency of SAs, in [14] to check the transition function of monitors generated from regular expressions, and in [87] to verify the lack of interference between enforcers.

6 Conclusions

Society demands for highly-dependable, large and dynamic systems that can serve citizens in their daily operations. Such systems are increasingly difficult to verify in-house due to their size, complexity and dynamic nature. Runtime techniques, in particular enforcement and healing solutions, can be exploited in the field to compensate the validation and verification activities performed in-house. The joint collaboration of enforcement techniques, which can prevent failures, and healing techniques, which can overcome an observed failure, can significantly increase the dependability of software systems.

This chapter discusses some of the achievements in these related areas, providing an overview of the available solutions. The material presented in this chapter can represent a valuable starting point for researchers interested in enforcement and healing solutions.

Acknowledgment. The authors would like to thank Antoine El-Hokayem, Raphaël Khoury, and Srinivas Pinisetty for commenting on the section related to runtime enforcement. The authors warmly thank the reviewers for their comments on a preliminary version of this chapter.

References

1. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: O'Boyle, M.F.P., Pingali, K. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 259–269. ACM (2014)

2. Avizienis, A.: The n-version approach to fault-tolerant software. IEEE Transactions on Software Engineering (TSE) 11(12), 1491–1501 (1985)
3. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Shin, Y.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering (TSE) 41(5), 507–525 (May 2015)
4. 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 international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. International Journal on Software Tools for Technology Transfer (Apr 2017)
5. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E.: Enforceable security policies revisited. ACM Trans. Inf. Syst. Secur. 16(1), 3:1–3:26 (Jun 2013), http://doi.acm.org/10.1145/2487222.2487225
6. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
7. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design 48(1-2), 46–93 (2016)
8. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Proceedings of the Workshop on Foundations of Computer Security (FCS'02), Copenhagen, Denmark (2002)
9. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Sarkar, V., Hall, M.W. (eds.) Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005. pp. 305–314. ACM (2005)
10. Beauquier, D., Cohen, J., Lanotte, R.: Security policies enforcement using finite and pushdown edit automata. Int. J. Inf. Sec. 12(4), 319–336 (2013), http://dx.doi.org/10.1007/s10207-013-0195-8
11. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? - edited automata revisited. Int. J. Inf. Sec. 10(4), 239–254 (2011)
12. Bielova, N., Massacci, F.: Predictability of enforcement. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6542, pp. 73–86. Springer (2011)
13. Bielova, N., Massacci, F.: Iterative enforcement by suppression: Towards practical enforcement theories. Journal of Computer Security 20(1), 51–79 (2012)
14. Blech, J.O., Falcone, Y., Becker, K.: Towards certified runtime verification. In: Aoki, T., Taguchi, K. (eds.) Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7635, pp. 494–509. Springer (2012)
15. Bloem, R., Könighofer, B., Könighofer, R., Wang, C.: Shield synthesis: - runtime enforcement for reactive systems. In: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 533–548 (2015)
16. Bonakdarpour, B., Finkbeiner, B.: Runtime verification for hyperltl. In: Falcone, Y., Sánchez, C. (eds.) Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10012, pp. 41–45. Springer (2016)
17. Bultan, T., Sen, K. (eds.): Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017. ACM (2017)

18. Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., Pezzè, M.: Automatic recovery from runtime failures. In: Proceedings of the International Conference on Software Engineering (ICSE). pp. 782–791. IEEE Press (2013)
19. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds: Exploiting the intrinsic redundancy of web applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(3), 16 (2015)
20. Chabot, H., Khoury, R., Tawbi, N.: Generating in-line monitors for rabin automata. In: Jøsang, A., Maseng, T., Knapskog, S.J. (eds.) Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems, NordSec 2009, Oslo, Norway, 14-16 October 2009. Proceedings. Lecture Notes in Computer Science, vol. 5838, pp. 287–301. Springer (2009), http://dx.doi.org/10.1007/978-3-642-04766-4
21. Chang, E., Manna, Z., Pnueli, A.: The Safety-Progress Classification. Tech. rep., Stanford University, Dept. of Computer Science (1992)
22. Chang, H., Mariani, L., Pezzè, M.: In-field healing of integration problems with COTS components. In: Proceedings of the International Conference on Software Engineering (ICSE) (2009)
23. Chang, H., Mariani, L., Pezzè, M.: Exception handlers for healing component-based systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 22(4), 30 (2013)
24. Charafeddine, H., El-Harake, K., Falcone, Y., Jaber, M.: Runtime enforcement for component-based systems. In: Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1789–1796. ACM (2015)
25. Chen, F., d'Amorim, M., Roşu, G.: Checking and correcting behaviors of java programs at runtime with java-mop. Electronic Notes in Theoretical Computer Science 144(4), 3–20 (2006)
26. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, 23-25 June 2008. pp. 51–65. IEEE Computer Society (2008)
27. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design 49(1-2), 109–158 (2016), https://doi.org/10.1007/s10703-016-0251-x
28. Cuppens, F., Cuppens-Boulahia, N., Ramard, T.: Availability enforcement by obligations and aspects identification. In: Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on. pp. 10–pp. IEEE (2006)
29. Ding, R., Fu, Q., Lou, J.G., Lin, Q., Zhang, D., Shen, J., Xie, T.: Healing online service systems via mining historical issue repositories. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 318–321. IEEE (2012)
30. Dolzhenko, E., Ligatti, J., Reddy, S.: Modeling runtime enforcement with mandatory results automata. International Journal of Information Security 14(1), 47–60 (Feb 2015)
31. 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. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8803, pp. 239–253. Springer (2014)
32. El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Bultan and Sen [17], pp. 125–135
33. El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Bultan and Sen [17], pp. 372–375

34. 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.) Proceedings of the 1999 Workshop on New Security Paradigms, Caledon Hills, ON, Canada, September 22-24, 1999. pp. 87–95. ACM (1999)
35. 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 International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)
36. Falcone, Y., Currea, S., Jaber, M.: Runtime verification and enforcement for Android applications with RV-Droid. In: Qadeer and Tasiran [80], pp. 88–95
37. Falcone, Y., Fernandez, J.C., Mounier, L.: Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In: Sekar, R., Pujari, A. (eds.) Information Systems Security, Lecture Notes in Computer Science, vol. 5352, pp. 41–55. Springer Berlin Heidelberg (2008)
38. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? International Journal on Software Tools for Technology Transfer 14(3), 349–382 (2012)
39. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. International Journal on Software Tools for Technology Transfer pp. 1–25 (2016)
40. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S.: Runtime enforcement of regular timed properties by suppressing and delaying events. Systems & Control Letters 123, 2–41 (2016)
41. Falcone, Y., Marchand, H.: Runtime enforcement of k-step opacity. In: Proceedings of the 52nd IEEE Conference on Decision and Control, CDC 2013, December 10-13, 2013, Firenze, Italy. pp. 7271–7278. IEEE (2013)
42. Falcone, Y., Marchand, H.: Enforcement and validation (at runtime) of various notions of opacity. Discrete Event Dynamic Systems 25(4), 531–570 (2015), http://dx.doi.org/10.1007/s10626-014-0196-4
43. Falcone, Y., Mounier, L., Fernandez, J., Richier, J.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
44. Fong, P.W.L.: Access control by tracking shallow execution history. In: 2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA. pp. 43–55. IEEE Computer Society (2004)
45. Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., Tonella, P.: Search-based synthesis of equivalent method sequences. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (2014)
46. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering (TSE) 38(1), 54–72 (2012)
47. Hallé, S., Khoury, R., Betti, Q., El-Hokayem, A., Falcone, Y.: Decentralized enforcement of document lifecycle constraints. Information Systems (2017)
48. Hallé, S., Khoury, R., El-Hokayem, A., Falcone, Y.: Decentralized enforcement of artifact lifecycles. In: Matthes, F., Mendling, J., Rinderle-Ma, S. (eds.) 20th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2016, Vienna, Austria, September 5-9, 2016. pp. 1–10. IEEE Computer Society (2016)
49. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified in-lined reference monitoring on .net. In: Sreedhar, V.C., Zdancewic, S. (eds.) Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS 2006, Ottawa, Ontario, Canada, June 10, 2006. pp. 7–16. ACM (2006)

50. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (TOPLAS) 28(1), 175–205 (2006)
51. Hosek, P., Cadar, C.: Safe software updates via multi-version execution. In: Proceedings of the International Conference on Software Engineering (ICSE) (2013)
52. Humphrey, L., Könighofer, B., Könighofer, R., Topcu, U.: Synthesis of admissible shields. In: Bloem, R., Arbel, E. (eds.) Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10028, pp. 134–151 (2016)
53. IEEE: Systems and Software Engineering - Vocabulary. Tech. Rep. ISO/IEC/IEEE 24765, IEEE International Standard (2010)
54. 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.) Proceedings of the 6th Asia-Pacific Workshop on Systems, APSys 2015, Tokyo, Japan, July 27-28, 2015. pp. 16:1–16:7. ACM (2015)
55. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the International Conference on Automated software engineering (ASE) (2005)
56. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
57. Khoury, R., Hallé, S.: Runtime enforcement with partial control. In: García-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9482, pp. 102–116. Springer (2015)
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 (Jul 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. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M.: Computational analysis of runtime monitoring - fundamentals of java-mac. Electr. Notes Theor. Comput. Sci. 70(4), 80–94 (2002)
61. 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 International Conference, Miami, FL, USA, November 1-3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9419, pp. 458–472. Springer (2015)
62. Ligatti, J., Bauer, L., Walker, D.: Enforcing non-safety security policies with program monitors. In: di Vimercati, S.D.C., Syverson, P.F., Gollmann, D. (eds.) Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3679, pp. 355–373. Springer (2005)
63. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12(3), 19:1–19:41 (Jan 2009)
64. Ligatti, J., Reddy, S.: A theory of runtime enforcement, with results. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6345, pp. 87–100. Springer (2010)
65. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. Electr. Notes Theor. Comput. Sci. 179, 31–46 (2007), http://dx.doi.org/10.1016/j.entcs.2006.08.029

66. 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 International Workshop, STM 2016, Heraklion, Crete, Greece, September 26-27, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9871, pp. 64–81. Springer (2016)
67. Martinelli, F., Matteucci, I., Saracino, A., Sgandurra, D.: Remote policy enforcement for trusted application execution in mobile environments. In: Bloem, R., Lipp, P. (eds.) Trusted Systems - 5th International Conference, INTRUST 2013, Graz, Austria, December 4-5, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8292, pp. 70–84. Springer (2013)
68. Martinelli, F., Matteucci, I., Saracino, A., Sgandurra, D.: Enforcing mobile application security through probabilistic contracts. In: Joosen, W., Martinelli, F., Heyman, T. (eds.) Proceedings of the 2014 ESSoS Doctoral Symposium co-located with the International Symposium on Engineering Secure Software and Systems (ESSoS 2014), Munich, Germany, February 26, 2014. CEUR Workshop Proceedings, vol. 1298. CEUR-WS.org (2014)
69. Martinelli, F., Mori, P., Saracino, A.: Enhancing android permission through usage control: a BYOD use-case. In: Ossowski [70], pp. 2049–2056
70. Ossowski, S. (ed.): Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. ACM (2016)
71. Owicki, S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Transaction Programming Languages and Systems 4(3), 455–495 (1982)
72. Pavlich-Mariscal, J., Michel, L., Demurjian, S.: A formal enforcement framework for role-based access control using aspect-oriented programming. In: Model Driven Engineering Languages and Systems, pp. 537–552. Springer (2005)
73. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H.: Runtime enforcement of parametric timed properties with practical applications. In: Lesage, J., Faure, J., Cury, J.E.R., Lennartson, B. (eds.) 12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014. pp. 420–427. International Federation of Automatic Control (2014)
74. 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.) Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014. pp. 1279–1286. ACM (2014)
75. 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 International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9333, pp. 306–320. Springer (2015)
76. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.: Runtime enforcement of timed properties revisited. Formal Methods in System Design 45(3), 381–422 (2014)
77. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.L.: Runtime enforcement of timed properties. In: Qadeer and Tasiran [80], pp. 229–244
78. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. In: Ossowski [70], pp. 1628–1633
79. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Formal Methods in System Design pp. 1–46 (2017)
80. Qadeer, S., Tasiran, S. (eds.): Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7687. Springer (2013)
81. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM journal on control and optimization 25(1), 206–230 (1987)

82. Ramadge, P.J., Wonham, W.M.: The control of discrete event systems. Proceedings of the IEEE 77(1), 81–98 (1989)
83. Renard, M.: GREP. https://github.com/matthieurenard/GREP (2017)
84. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H.: Optimal enforcement of (timed) properties with uncontrollable events. Mathematical Structures in Computer Science pp. 1–46 (2017)
85. 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 International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 542–560. Springer (2015)
86. Renard, M., Rollet, A., Falcone, Y.: Runtime enforcement using Büchi games. In: Proceedings of Model Checking Software - 24th International Symposium, SPIN 2017, Co-located with ISSTA 2017, Santa Barbara, USA. pp. 70–79. ACM (July 2017)
87. Riganelli, O., Micucci, D., Mariani, L., Falcone, Y.: Verifying policy enforcers. In: Proceedings of the International Conference on Runtime Verification (RV) (2017)
88. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000)
89. Sridhar, M., Hamlen, K.W.: Flexible in-lined reference monitor certification: Challenges and future directions. In: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification. pp. 55–60. PLPV '11 (2011)
90. Swanson, J., Cohen, M.B., Dwyer, M.B., Garvin, B.J., Firestone, J.: Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (2014)
91. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Inf. Comput. 206(2-4), 158–184 (2008), http://dx.doi.org/10.1016/j.ic.2007.07.009
92. 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 International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9690, pp. 65–81. Springer (2016)
93. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering (TSE) 28(2) (Feb 2002)
94. Zhang, X., Leucker, M., Dong, W.: Runtime verification with predictive semantics. In: Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7226, pp. 418–432. Springer (2012)